UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

AQUA – ATIVIDADES DE QUALIDADE NO CONTEXTO ÁGIL

LUIZ CAVAMURA JÚNIOR

São Carlos/SP

Fevereiro/2008

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

C377aa

Cavamura Júnior, Luiz.
Aqua - atividades de qualidade no contexto ágil / Luiz Cavamura Júnior. -- São Carlos : UFSCar, 2008.
164 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2008.

1. Engenharia de software. 2. Métodos ágeis. 3. Qualidade de processos. 4. Técnicas de inspeção. 5. Software – testes. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

"AQUA - Atividades de Qualidade no Contexto Ágil"

LUIZ CAVAMURA JÚNIOR

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Profa. Dra. Sandra Camargo P. Ferraz Fabbri
(Orientadora – DC/UFSCar)

Profa. Dra. Simone do Rocio Senger de Souza
(ICMC/USP)

Profa. Dra. Luciana Andréia F. Martimiano
(UEM)

São Carlos
Fevereiro/2008

Ao meu filho
Luiz Gustavo,

que, com a sua chegada, durante essa caminhada,
me encheu de felicidade,

dedico este trabalho.

AGRADECIMENTOS

À Deus, por tudo.

À *Profa. Dra. Sandra Camargo P. F. Fabbri*, pela oportunidade, confiança, paciência, ajuda e orientação.

À toda minha *Família* e A*migos*, em especial à minha esposa *Nadia* e meu filho *Luiz Gustavo*, pela compreensão, auxílio e incentivo durante essa caminhada.

À *EDRA*, pela oportunidade e incentivo.

Ao *Centro Estadual de Educação Tecnológica Paula Souza*, em especial à *ETEC Prof. Armando Bayeux da Silva*, pelo auxílio.

SUMÁRIO

LISTA DE FIGURAS .... **iv**

LISTA DE TABELAS .... **vi**

LISTA DE ABREVIATURAS .... **viii**

RESUMO .... **ix**

ABSTRACT .... **x**

CAPÍTULO 1 - INTRODUÇÃO **1**

1.1. Considerações iniciais 1

1.2. Motivação e Objetivo 2

1.3. Organização do Trabalho 3

CAPÍTULO 2 – MÉTODOS ÁGEIS **4**

2.1. Considerações iniciais 4

2.2. Metodologia Ágil 5

2.3. Onde está a diferença? 7

2.4. Métodos Ágeis 10

a) Extreme Programming (XP) 10

b) Scrum 15

c) Dinamic Systems Development Method (DSDM) 19

d) Feature Driven Development (FDD) 24

e) Adaptive Software Development (ASD) 29

f) Família de Métodos Crystal 34

2.5 Considerações finais 39

CAPÍTULO 3 – ATIVIDADES DE GARANTIA DE QUALIDADE DE SOFTWARE **40**

3.1. Considerações iniciais 40

3.2. Qualidade de Software 41

3.3. Teste 44

3.3.1. Técnicas de Teste 47

3.4. Inspeção 54

3.4.1. Técnicas de Leitura para o Processo de Inspeção 56

3.5. Planejamento 60

3.5.1. Técnica de Pontos de Casos de Uso 62

3.6. Considerações Finais 66

CAPÍTULO 4 - TESTE E INSPEÇÃO NO CONTEXTO ÁGIL 67
4.1. Considerações Iniciais 67
4.2. A Atividade de Teste no contexto Ágil. 67
4.2.1. Teste de Unidade 68
4.2.2. Teste de Aceitação 70
4.3. Automação dos Testes de Unidade e de Aceitação 71
4.3.1. Automação de Testes de Unidade com o JUnit 72
4.3.2. Automação de Testes de Aceitação com Selenium 76
4.4. O Testador no Contexto Ágil 84
4.5. Inspeção no Contexto Ágil 85
4.6. Considerações Finais 89
CAPÍTULO 5 - AQUA - ATIVIDADES DE QUALIDADE NO CONTEXTO ÁGIL 90
5.1 Considerações Iniciais 90
5.2 Caracterização dos Métodos Ágeis Estudados 91
5.2.1. Artefatos e Processo 91
5.2.2. Métodos Ágeis e o Paradigma Orientado a Objetos 94
5.3. AQUA – Atividades de Qualidade no contexto Ágil 95
5.3.1. A Estratégia de Aplicação das atividades da AQUA 97
5.3.2. Atividade IFC – Inspeção das Funcionalidades junto ao Cliente 101
5.3.3. Atividade IFA – Inspeção das Funcionalidades nos Artefatos 106
5.3.4. Atividade ECT – Elaboração dos Cartões de Teste 109
5.3.5. Atividade ETI – Estimativa de Tempo para as Iterações 113
5.4. Considerações Finais 116
CAPÍTULO 6 – EXEMPLO DE APLICAÇÃO 117
6.1. Considerações iniciais 117
6.2. Descrição do Sistema para Clínica Veterinária (SCV) 118
6.3. Aplicação da atividade IFA 121
6.4. Aplicação da atividade ECT 123
6.5. Aplicação da atividade ETI 130
6.6. Considerações finais 136
CAPÍTULO 7 - CONCLUSÕES 137
7.1. Contribuições 139
7.2. Trabalhos Futuros 140
REFERÊNCIAS BIBLIOGRÁFICAS 141

APÊNDICE A **151**
APÊNDICE B... **157**
APÊNDICE C **161**

LISTA DE FIGURAS

Figura 2. 1 - Custo da mudança - projetos com um modelo não-ágil (AMBLER, 2004). .. 9
Figura 2. 2 - Custo da mudança - projetos com metodologias ágeis (BECK, 2004).......... 9
Figura 2. 3 - Processo do XP (adaptado de ABRAHAMSSON et al., 2002).................... 11
Figura 2. 4 - Visão geral do processo Scrum (adaptado de SCHWABER, 2004)............ 16
Figura 2. 5 - Processo DSDM (adaptado de ABRAHAMSSON et al., 2002). 20
Figura 2. 6 - Processo FDD (adaptado de ABRAHAMSSON et al., 2002)...................... 25
Figura 2. 7 - O projeto e construção por funcionalidades do FDD (adaptado de ABRAHAMSSON et al.,2002). 25
Figura 2. 8 - Processo ASD (adaptado de ABRAHAMSSON et al., 2002)...................... 30
Figura 2. 9 - Métodos Crystal e suas dimensões (ABRAHAMSSON et al., 2002). 34
Figura 2. 10 - Uma iteração do Crystal Orange (adaptado de ABRAHAMSSON et al., 2002).. 37
Figura 3. 1 - Limites de uma partição de equivalência (FABBRI, 2005)......................... 50
Figura 3. 2 - Notação para construção do grafo (adaptado de MYERS, 2004)................ 52
Figura 3. 3 - Notação de Restrições (MEYERS, 2004).. 52
Figura 3. 4 - Grafo causa-efeito completo.. 53
Figura 3. 5 - Processo de inspeção (adaptado de SHULL, 1998).................................... 55
Figura 4. 1- Resultados de testes bem sucedidos no JUnit.. 75
Figura 4. 2 – Teste independente da interface do usuário (adaptado de MARTIN, 2005).76
Figura 4. 3 - Isolando os testes de interface dos testes das regras de negócio (adaptado de MARTIN, 2005). 77
Figura 4. 4 - Trabalhando com a base de dados (adaptado de MARTIN, 2005).............. 77
Figura 4. 5 - Interface da ferramenta Selenium IDE. 79
Figura 4. 6 - Comandos open e type e seus argumentos. 80
Figura 4. 7 - Operações registradas no Selenium... 81
Figura 4. 8 - Caso de teste bem sucedido. 82
Figura 4. 9 - Caso de teste que falhou. 83
Figura 4. 10 - PI-XP baseado nos requisitos – abstração do processo (TOMA, 2004)..... 88
Figura 4. 11 - Detalhe da atividade de desenvolvimento do XP (TOMA, 2004)............. 89
Figura 5. 1 - Processo genérico dos métodos ágeis... 94
Figura 5. 2 – Atividades da abordagem AQUA 96

Figura 5. 3 - Estratégia de aplicação da abordagem AQUA. 98
Figura 5. 4 - Especificação da customização a ser feita no sistema de gestão empresarial. 102
Figura 5. 5 - Funcionalidades descritas pelo proprietário da empresa "B". 104
Figura 5. 6 – Cartão de Teste proposto.. 112
Figura 6. 1 - Diagrama de casos de uso do sistema para Clínica Veterinária (SCV). 118

LISTA DE TABELAS

Tabela 2. 1 - Papéis e responsabilidades do XP (BECK, 2004; ABRAHAMSSON et al., 2002). 13
Tabela 2. 2 - Práticas do XP (BECK, 2004; LARMAN, BASILI, 2003; ABRAHAMSSON et al., 2002). 14
Tabela 2. 3 - Papéis e responsabilidades do Scrum (ABRAHAMSSON et al., 2002). 18
Tabela 2. 4 - Práticas adotadas pelo Scrum (ABRAHAMSSON et al., 2002). 18
Tabela 2. 5 - Papéis e responsabilidades do DSDM (ABRAHAMSSON et al., 2002). 23
Tabela 2. 6 - Práticas do DSDM (CLIFTON et al.,2003). 24
Tabela 2. 7 - Papéis e responsabilidades do FDD (ABRAHAMSSON et al.,2002). 28
Tabela 2. 8 - Práticas do FDD (ABRAHAMSSON et al., 2002). 29
Tabela 2. 9 - Práticas do ASD (ABRAHAMSSON et al., 2002). 33
Tabela 2. 10 - Papéis e responsabilidades do método Crystal Clear (ABRAHAMSSON et al., 2002). 36
Tabela 2. 11 - Papéis e responsabilidades do método Crystal Orange (COCKBURN, 2006). 38
Tabela 2. 12 - Práticas da metodologia Crystal (COCKBURN, 2006). 39
Tabela 3. 1 - Tabela de decisão exemplo. 54
Tabela 3. 2 - Papéis do processo de inspeção (LAITENBERGER, 2001). 55
Tabela 3. 3 - Caracterização das técnicas de leitura (LAITENBERGER, 2001). 59
Tabela 3. 4 - Peso dos atores (KARNER, 1993). 62
Tabela 3. 5 - Características técnicas (KARNER, 1993). 65
Tabela 3. 6 - Recursos disponíveis (KARNER, 1993). 65
Tabela 4. 1 - Casos de teste conforme a especificação (adaptado de MYERS, 2004). 73
Tabela 4. 2 - Automação dos testes de aceitação (MARTIN, 2005). 76
Tabela 4. 3 - Caso de teste de aceitação. 78
Tabela 4. 4 - Operações a serem realizadas para o caso de teste #1 da Tabela 4.3. 81
Tabela 5. 1 – Artefatos usados nos métodos ágeis estudados. 92
Tabela 5. 2 - Roteiro da atividade IFA 108
Tabela 6. 1 - Especificação do caso de uso "Fazer agendamento". 119
Tabela 6. 2 - Especificação do caso de uso "Cancelar agendamento". 120
Tabela 6. 3 - Especificação do caso de uso "Consultar agendamento". 120

Tabela 6. 4 - Classes de equivalência da funcionalidade "Fazer Agendamento"......... 124
Tabela 6. 5 – Cartão de Teste da funcionalidade “Fazer Agendamento”..................... 125
Tabela 6. 6 - Classes de equivalência da funcionalidade "Cancelar Agendamento". .. 126
Tabela 6. 7 - Situações da Funcionalidade “Cancelar Agendamento”que tendem a erros 126
Tabela 6. 8 – Cartão de Teste da funcionalidade “Cancelar Agendamento” 127
Tabela 6. 9 - Classes de equivalência da funcionalidade “Consultar Agendamento”.. 128
Tabela 6. 10 – Situações da funcionalidade “Consultar Agendamentos” que tendem a erros. 128
Tabela 6. 11 – Cartão de Teste da funcionalidade “Consultar Agendamento”............ 129
Tabela 6. 12 - Atores do sistema. 131
Tabela 6. 13 - Funcionalidades da iteração. 132
Tabela 6. 14 - Pontos não ajustados da iteração. 132
Tabela 6. 15 - Características técnicas.. 133
Tabela 6. 16 - Recursos do projeto. 134

LISTA DE ABREVIATURAS

AQUA	Atividades de Qualidade no Contexto Ágil
ASD	Adaptive Software Development
AUCP	Valor de Pontos por Caso de Uso Ajustado
DOT	Desenvolvimento Orientado por Testes
DSDM	Dinamic Systems Development Method
ECT	Elaboração do Cartão de Teste
EF	Fator de Ambiente
ES	Engenharia de Software
ETI	Estimativa de Tempo para as Iterações
FDD	Feature Driven Development
IFA	Inspeção das Funcionalidades nos Artefatos
IFC	Inspeção das Funcionalidades junto ao Cliente
Lag1	Leitura Ágil 1
Lag2	Leitura Ágil 2
Lag3	Leitura Ágil 3
MA	Métodos Ágeis
MADS	Metodologia Ágil de Desenvolvimento de Software
OO	Orientado a Objetos
PI-XP	Processo de Inspeção – XP
QS	Qualidade de Software
SCV	Sistema Clínica Veterinária
TCF	Fator de Complexibilidade Técnica
UAW	Peso Total dos Atores não Ajustado
UUCP	Peso Total não Ajustado
UUCW	Peso Total dos Casos de Uso não Ajustado
VV&T	Validação, Verificação e Testes
XP	Extreme Programming

RESUMO

Cenário: O termo “agilidade” vem adquirindo cada vez mais destaque na Engenharia de Software pela crescente necessidade de executar e entregar sistemas de software em prazos cada vez menores. A metodologia ágil de desenvolvimento de software propõe um desenvolvimento rápido e iterativo, buscando atender todas as necessidades do cliente e obter bons resultados durante o processo de desenvolvimento. Embora essa metodologia tenha alcançado destaque, verifica-se que seus métodos, mesmo os mais citados na literatura, não apresentam explicitamente diretrizes detalhadas para sua utilização. Algumas referências mencionam detalhes das atividades realizadas, das práticas utilizadas e das informações que devem ser registradas. No entanto, não se encontram estudos de caso ou exemplos de utilização, que mostrem, na prática, do início ao fim, o processo do método adotado. ***Objetivo:*** Este trabalho tem como objetivo contribuir para a melhoria da qualidade dos processos ágeis, propondo a abordagem denominada AQUA – Atividades de Qualidade no contexto Ágil. Essa abordagem envolve a aplicação de atividades de garantia de qualidade de software, como inspeção, teste e planejamento que podem ser utilizadas independentemente do método ágil adotado, nas fases iniciais de levantamento dos requisitos. ***Método:*** Foram estudados os métodos ágeis mais citados na literatura, a partir do que foi extraído um processo genérico que os caracteriza e um conjunto de artefatos utilizados por eles, o que deu apoio à definição da abordagem proposta. ***Resultados:*** A abordagem foi aplicada utilizando os requisitos de um sistema desenvolvido durante uma disciplina de pós-graduação da Universidade Federal de São Carlos, e os resultados dão indícios de sua contribuição para a melhoria da qualidade do processo. ***Conclusão:*** Embora o exemplo utilizado tenha sido pequeno, a utilização da abordagem não dificultou a utilização do método ágil em questão e mostrou pontos positivos para a melhoria da qualidade do processo.

ABSTRACT

Background: The term "agility" has increasingly received attention in Software Engineering, since there is a growing demand for executing and delivering software systems in a shorter and shorter period of time. Agile software development proposes a rapid and iterative development aiming at meeting all of clients' needs and, as a consequence, achieving good results throughout development process. Although this methodology is well-known, it is important to notice that its methods, even the ones which are the most quoted in the scientific literature, do not establish any detailed guidelines so that they can be effectively adopted. Some examples encompass details of activities performed, practices and procedures used and some piece of information which must be documented. However, there are not any case studies which show, practically, the whole process of the adopted method. ***Aim:*** This work aimed at contributing in order to achieving improvements in quality of the agile processes by introducing an approach named AQUA - Quality Activities in Agile Context. This approach takes into account the application of software quality assurance activities such as inspection, test and planning activities which may be used along with any agile method adopted, in the early stages of requirements elicitation. ***Method:*** Only the most quoted methods in the literature were analyzed, from which a generic process was extracted besides a group of artifacts utilized by these methods, supporting, thereby, the definition process of the strategy proposed. ***Results:*** This approach was applied by using software requirements developed during a graduate course in Federal University of São Carlos, and the results give insights of its contribution to the improvement in process quality. ***Conclusion:*** Although this example had been small, the use of the proposed approach did not make it difficult the application of the agile method adopted and presented benefits with regard to improvements of process quality.

CAPÍTULO 1

INTRODUÇÃO

1.1. Contexto

A cada dia o mercado atual encontra-se cada vez mais competitivo e acirrado. A necessidade de processos cada vez mais ágeis, confiáveis, eficientes e eficazes é comum em todas as empresas seja qual for o segmento. Nesse contexto, a automação de processos através de sistemas de software é um dos principais fatores para que a empresa se destaque e se torne mais competitiva, visando sempre estar um passo à frente de seus concorrentes. A busca por tais objetivos é constante e vital para a empresa. Dessa forma, o desenvolvimento de tais sistemas deve atender às necessidades da empresa em tempo hábil e com qualidade.

O desenvolvimento de software não é uma atividade simples como outras atividades profissionais em que os métodos e os processos não sofrem mudanças constantes. A Engenharia de Software (ES) é evolutiva e acompanha o avanço da tecnologia. Esses fatos são constatados dados os inúmeros métodos de desenvolvimento de software existentes, que surgiram nas últimas duas décadas e que hoje são considerados métodos tradicionais e mais pesados em termos de documentação. Tal documentação, embora estabeleça uma sistemática visando obter qualidade no processo, demanda um tempo significativo, principalmente quando a equipe de desenvolvimento é pequena. Nesse caso, o tempo é um fator importante de competitividade dada a crescente necessidade de se executar e entregar sistemas de software com qualidade em prazos cada vez menores.

Com o objetivo de agilizar o desenvolvimento de software, surgiu o Movimento Ágil (MANIFESTO,2001) que propõe um novo paradigma de desenvolvimento, no qual é priorizada a produção de código ao invés de extensa documentação, além de várias outras características. A partir de então, a Metodologia Ágil de Desenvolvimento de

Software (MADS) tem se tornado uma opção de desenvolvimento de software, em contrapartida às metodologias tradicionais.

A MADS propõe um desenvolvimento rápido, valorizando o contato com o cliente e o desenvolvimento em iterações, estando preparada para modificações necessárias que vierem a ocorrer durante o processo de desenvolvimento, registrando somente o que é essencial ao processo.

Considerando o cenário nacional, que é fortemente composto de pequenas e médias empresas, esse paradigma de desenvolvimento é muito adequado, uma vez que ele é mais apropriado para equipes e projetos pequenos. No entanto, o maior problema com essa forma de desenvolvimento é que, por ser uma alternativa ainda nova, há pouca informação e avaliação de seu uso, faltando diretrizes e maiores detalhes para sua efetiva aplicação na prática.

1.2. Motivação e Objetivo

Os Métodos Ágeis (MA) têm se tornado cada vez mais populares e têm sido cada vez mais usados. No entanto, como foi dito anteriormente, ainda não se encontram diretrizes detalhadas de como aplicá-los, com segurança, na prática. Mesmo tendo suas fases bem definidas, as atividades que devem ser executadas durante o processo e os artefatos utilizados, não são declarados com precisão.

Atividades de garantia de Qualidade de Software (QS), como inspeção, teste e planejamento que são essenciais, independentemente do paradigma utilizado, embora sejam citadas no processo desses métodos, não possuem nenhuma diretriz que indique como executá-las ou documentá-las. Mesmo que muitas vezes, na literatura, fique subentendido que as informações referentes a elas devam ser armazenadas de alguma forma, não fica claro como fazer isso.

Assim, considerando a relevância das atividades de garantia de qualidade citadas, este trabalho tem o objetivo de contribuir para a melhoria do processo dos MA definindo e sugerindo uma maneira de registrar detalhes sobre atividades de inspeção, teste e planejamento, que possam ser aplicadas independentemente do método ágil

utilizado. Essas atividades atuam nas fases iniciais do desenvolvimento, quando os requisitos do sistema estão sendo definidos e tratados, momento este crucial no paradigma de desenvolvimento ágil.

1.3. Organização do Trabalho

Este trabalho está organizado em sete capítulos, além da seção referente às referências bibliográficas e dos Apêndices A, B e C.

Os capítulos estão organizados da seguinte forma: no Capítulo 2 são apresentados os MA mais citados na literatura e abordados neste trabalho, explorando seus princípios, processos, práticas e os papéis neles existentes; no Capítulo 3 comentam-se as atividades de inspeção, teste e planejamento, de garantia de QS; no Capítulo 4 são abordadas as atividades de teste e inspeção no contexto ágil, apresentando os conceitos, características e práticas existentes; no Capítulo 5 é apresentada a abordagem AQUA – Atividades de Qualidade no contesto Ágil, proposta neste trabalho; no Capítulo 6 é mostrado um exemplo de aplicação dessa proposta e no Capítulo 7 encontram-se as conclusões.

No Apêndice A apresenta-se o código de um programa utilizado como exemplo no Capítulo 4; no Apêndice B estão as técnicas de leitura do PI-XP (Processo de Inspeção – XP) citadas no Capítulo 4, que serviram de subsídio para algumas definições deste trabalho; e no Apêndice C são mostrados os formulários que devem ser usados na proposta deste trabalho.

CAPÍTULO 2

MÉTODOS ÁGEIS

2.1. Considerações iniciais

Agilidade, adaptação, qualidade e atenção ao cliente são fatores fundamentais para atender o mercado atual, independentemente da área de negócios em que se atua, principalmente pela competitividade existente.

Para se manter ativa no mercado, no qual inovações são constantes, os prazos são cada vez mais curtos e as exigências do cliente são cada vez maiores, as empresas de desenvolvimento de software estão buscando sistemáticas que promovam esses fatores em seus projetos.

Nesse contexto, a MADS vem adquirindo importância e reconhecimento na área de desenvolvimento de software, pois adotam como princípio a adaptação à mudanças, a presença constante do cliente e a implementação iterativa, permitindo entregas freqüentes.

Embora os MA já possuam reconhecimento na comunidade da ES, fato verificado dada à quantidade de artigos publicados abordando esse contexto, ainda é escasso os indicadores de desempenho de tais métodos, como produtividade e eficiência em relação a métodos tradicionais (MELNIK, MAURER, 2004; PANCUR et al., 2003; REICHLMAYR, 2003). Dada a sua importância no desenvolvimento de software, neste capítulo são apresentados os fundamentos dessa metodologia e os MA mais populares.

Este capítulo está organizado da seguinte maneira: na Seção 2.2 são apresentados os princípios e fundamentos da metodologia ágil; na Seção 2.3 são apresentados os pontos divergentes entre os princípios da metodologia ágil e as metodologias tradicionais de desenvolvimento; na Seção 2.4 são apresentadas as características de seis MA e na Seção 2.5 encontram-se as considerações finais deste capítulo.

2.2. Metodologia Ágil

A MADS se destacou na indústria do software em fevereiro de 2001, através da criação da "*Agile Software Development Alliance*", ou simplesmente "*Aliança Ágil*", organização sem fins lucrativos que impulsiona o movimento e os conceitos do desenvolvimento ágil de software (AMBLER, 2004; ROOIJEN, 2006). A organização nasceu da participação de 17 especialistas em desenvolvimento de software em uma reunião realizada em Utah - Estados Unidos da América, a qual tinha por finalidade discutir e criar uma alternativa às metodologias tradicionais de desenvolvimento (AMBLER, 2004).

O passo inicial da "*Aliança Ágil*" foi a publicação do "*Manifesto Ágil*" (MANIFESTO, 2001; ROOIJEN, 2006) que denota toda a filosofia do desenvolvimento ágil de software. A essência do Manifesto está em:

- Indivíduos e interações são mais importantes que processos e ferramentas.
- O software funcionando é mais importante que extensa documentação.
- Colaboração do cliente tem maior valor do que negociação de contrato.
- Resposta rápida às mudanças tem maior valor que seguir os planos.

Compreende-se por esses princípios:

- O foco nas pessoas, a troca de conhecimentos e o relacionamento entre elas são fatores chave para que o projeto obtenha sucesso. De nada adiantará ferramentas e recursos se os membros da equipe trabalharem individualmente, voltados somente para seus interesses.
- A documentação deve ser utilizada somente em casos em que há a necessidade de tomar decisões importantes e imediatas no projeto. Essa documentação ainda deve ser curta e relatar somente o essencial para o projeto. Ressalta-se que a definição da documentação essencial varia de acordo com o contexto no qual se está trabalhando e envolve várias variáveis, tais como equipe, cliente, ambiente, porte do projeto, riscos, entre outros.

- O cliente é fundamental para o sucesso do projeto. É necessário que haja uma constante troca de informações, idéias e pensamentos entre a equipe de desenvolvimento e o cliente.

- Ter a habilidade de assimilar todo tipo de modificação durante a execução do projeto, desde mudanças no levantamento de requisitos, como mudanças na equipe de desenvolvimento e tecnologia a ser utilizada.

Os quatro valores do Manifesto Ágil, citados anteriormente, originaram doze princípios que sustentam o desenvolvimento ágil. São eles:

1- A maior prioridade é satisfazer o cliente através da entrega rápida e contínua de software de valor.

2- Mudanças nos requisitos do sistema são bem vindas, mesmo que de última hora.

3- Entregar versões do software funcionando freqüentemente, em poucas semanas.

4- Cliente e desenvolvedor devem trabalhar juntos ao longo de todo o projeto.

5- Trabalhe com pessoas motivadas, disponibilize o ambiente e o suporte necessário, confie no potencial da equipe.

6- O método mais eficiente para colher informações sobre o projeto é através da conversa "*cara-a-cara*" com o cliente.

7- A principal medida de progresso no projeto é trabalhar sobre software funcionando, e não em documentação.

8- Processos ágeis promovem um desenvolvimento sustentável. Cliente e desenvolvedor devem descobrir o ritmo de trabalho e mantê-lo constantemente.

9- Atenção contínua à excelência técnica e um bom projeto promovem a agilidade.

10- A simplicidade é essencial.

11- As melhores arquiteturas, requisitos e projeto provêm de equipes auto-organizadas.

12- Em intervalos regulares, a equipe refletirá em como se tornar mais eficiente e, segundo isto, poderão ajustar o seu comportamento conforme as necessidades.

Percebe-se atualmente que, embora tal metodologia venha adquirindo destaque em comunidades que estudam a ES, ela tem maior evidência somente em ambientes acadêmicos, normalmente através da realização de estudos de caso (ANDREA, 2003; REICHLMAYR, 2003). Com a falta de informações relativas à experiência de utilização desses métodos em ambientes não acadêmicos, os MA encontram resistência por parte de profissionais da área que não se encontram em ambientes acadêmicos.

2.3. Onde está a diferença?

A metodologia ágil difere da tradicional em vários aspectos, não somente em regras de como produzir e conduzir processos, mas também no próprio ambiente de desenvolvimento e nos integrantes da equipe, introduzindo a eles uma nova forma de pensar e desenvolver software.

A principal diferença entre as metodologias ágeis e as metodologias tradicionais está na mudança de paradigma de como desenvolver software, ou seja, adotar uma nova forma de pensar e adotar novos princípios no desenvolvimento de software. Canós et al. (2003) relacionaram algumas diferenças entre essas metodologias:

- As metodologias ágeis são baseadas em dados estatísticos levantados a partir do histórico de implementação de código, enquanto os métodos tradicionais se baseiam em normas contidas nos padrões seguidos pelo ambiente de desenvolvimento.
- As metodologias ágeis são preparadas para aceitar mudanças no decorrer do projeto enquanto as metodologias tradicionais oferecem resistência à mudanças.

- Metodologias ágeis têm sua forma de trabalho imposta internamente, pela própria equipe, enquanto as tradicionais seguem uma metodologia imposta externamente, ou seja, baseada em modelos existentes, seguindo os processos neles definidos.

- Metodologias ágeis exercem pouco controle aos processos, enquanto as metodologias tradicionais impõem muito mais controle a seus processos, com inúmeras normas e políticas a serem respeitadas.

- Normalmente não há contrato, mas quando ele é necessário, é o mais flexível possível, reforçando o princípio de que as mudanças são bem vindas e resguardando os interesses da equipe de desenvolvimento e do cliente. As metodologias tradicionais se fundam em contratos rígidos.

- O cliente faz parte da equipe de desenvolvimento, auxiliando e indicando o rumo da implementação. Nas metodologias tradicionais o cliente interage com a equipe de desenvolvimento somente através de reuniões, porém, sem poder de decisão no que diz respeito à forma de implementação do projeto.

- O desenvolvimento através de metodologias ágeis normalmente é feito por uma equipe pequena, normalmente não ultrapassando dez indivíduos trabalhando no mesmo local. As metodologias tradicionais se ocupam de vários integrantes.

- A arquitetura nas metodologias tradicionais é essencial e adota o uso de modelos para o desenvolvimento. As metodologias ágeis não enfatizam o uso de modelos, mas sim a documentação considerada essencial pela equipe.

Verifica-se que os MA compreendem um processo de desenvolvimento baseado nas entregas parciais do produto, permitindo que o cliente obtenha, rapidamente, o retorno do investimento feito. Essas entregas parciais são obtidas através do desenvolvimento em iterações, tendo como foco atender as necessidades do cliente e não nas regras de contrato, mantendo-se contato direto com o cliente durante todo o processo. Esses conceitos promovem a aceitação de mudanças no projeto durante o processo de desenvolvimento.

Um exemplo da diferença entre a metodologia ágil e os métodos tradicionais pode ser observado quando se analisa o fator custo proveniente de mudanças no projeto durante suas fases. Considerando-se o modelo não-ágil (metodologias tradicionais), por exemplo, pode-se até entender a resistência às mudanças pois, conforme a mudança, é necessária percorrer novamente todas as fases do modelo, desde o levantamento de requisitos. Segundo Ambler (2004), a Figura 2.1 representa o custo das modificações em cada fase de um modelo não-ágil. À medida que a alteração do projeto se faz necessária próxima do seu encerramento, seu custo cresce exponencialmente (PRESSMAN, 2006).

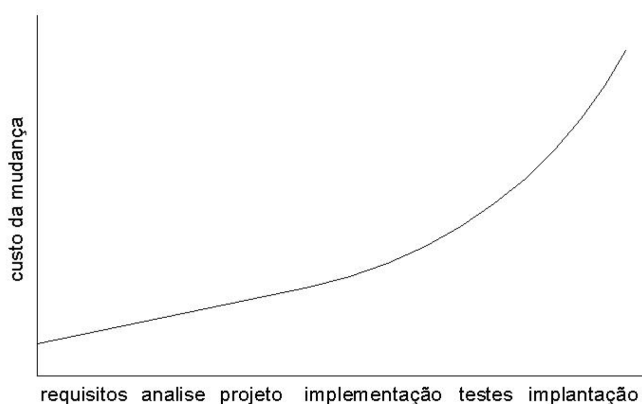

Figura 2. 1 - Custo da mudança - projetos com um modelo não-ágil (AMBLER, 2004).

Ao contrário das metodologias tradicionais, as metodologias ágeis estão preparadas para aceitar mudanças no projeto devido aos seus princípios, uma vez que está focada nas pessoas e não em processos e modelos de desenvolvimento, os quais denotam rígido controle. Segundo Beck (2004), a Figura 2.2 representa o custo de modificações no projeto utilizando-se metodologias ágeis. Percebe-se, através das Figuras 2.1 e 2.2, que o custo utilizando-se metodologias ágeis não cresce muito mesmo estando próximo do encerramento do projeto.

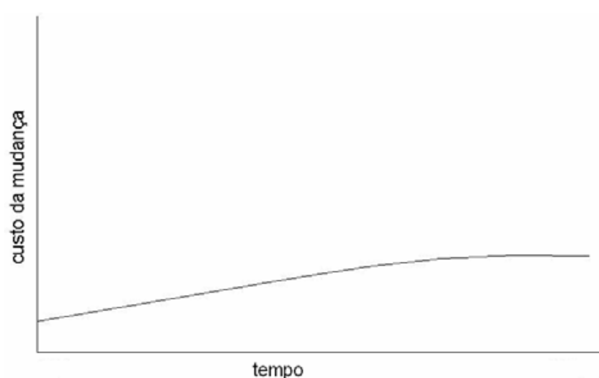

Figura 2. 2 - Custo da mudança - projetos com metodologias ágeis (BECK, 2004).

O custo da mudança em metodologias ágeis se mantém menor pelo fato de não estar amarrada a contratos, documentações e a modelagem em cada atividade (SOARES, 2004). Outro motivo pelo qual o custo se mantém linear é a iteratividade, possibilitando que o sistema seja produzido incrementalmente (BECK, 2004).

Embora tenha suas vantagens, a adoção de metodologias ágeis no desenvolvimento de software está vinculada ao contexto no qual o software se aplica. Software cuja precisão e confiabilidade são fatores de decisão e risco (sistema metroviário, aeroviário, cirúrgicos, entre outros) não pode dispensar características implícitas das metodologias tradicionais, tais como extensa documentação.

2.4. Métodos Ágeis (MA)

Os MA implementam os princípios da metodologia ágil. No contexto deste trabalho, são abordados os MA com maior destaque na literatura, no que diz respeito aos seus princípios, os processos que os compõem, seus papéis e responsabilidades e as práticas que são adotadas em sua aplicação. São eles o *Extreme Programming* (XP), *Scrum*, *Dinamic Systems Development Method* (DSDM), *Feature Driven Development* (FDD), *Adaptive Software Development* (ASD) e os métodos *Crystal Clear* e *Orange* da metodologia *Crystal*.

a) Extreme Programming (XP)

Princípios

O método *Extreme Programming* (XP) surgiu como decorrência de problemas originados do longo ciclo de vida dos modelos de desenvolvimento tradicionais. Ele se aplica principalmente quando são encontradas as seguintes situações (TELES, 2004):

- Projetos cujos requisitos não estão claros (passíveis de modificação).
- Desenvolvimento utilizando o paradigma orientado a objeto (OO).
- Equipe com pequeno número de integrantes.

O XP visa o desenvolvimento rápido e a satisfação do cliente. Seu princípio de comunicação busca o melhor relacionamento possível entre o cliente e equipe de desenvolvimento e entre os próprios integrantes da equipe. A implementação deve ser feita com código simples, com menor número de classes e métodos possível, não possuindo funções desnecessárias (BECK, 2004; TELES, 2004; ABRAHAMSSON et al., 2002).

O *feedback* constante do cliente se dá através das freqüentes validações de uma parte do software totalmente funcional. O cliente constantemente sugere novas características e funcionalidades aos desenvolvedores.

Processo

O processo do XP é composto por seis fases (BECK, 2004; ABRAHAMSSON et al., 2002; MARTIN et al., 2003). São elas:

- Exploração
- Planejamento
- Iterações para entregas
- Produção
- Manutenção
- Morte

A Figura 2.3 representa o processo do XP mostrando o relacionamento entre as fases do desenvolvimento, as quais são comentadas a seguir, de acordo com Beck (2004) e Abrahamsson et al.(2002).

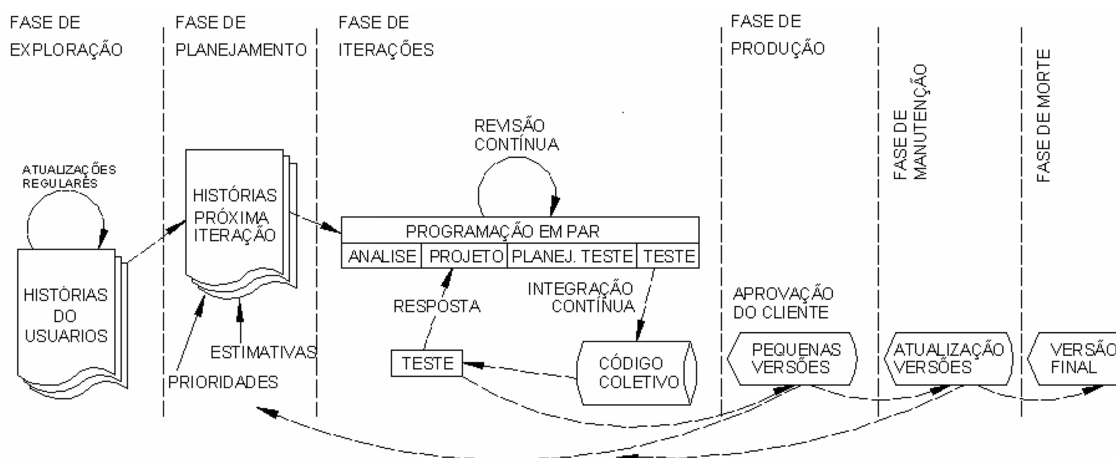

Figura 2. 3 - Processo do XP (adaptado de ABRAHAMSSON et al., 2002)

Exploração

Na fase de exploração é feito o levantamento de requisitos. O cliente escreve as características do sistema para a primeira versão, sejam elas funcionais ou não funcionais. Esses dados são registrados em fichas de papel, denominadas histórias do usuário. Paralelamente ao levantamento dos requisitos, a equipe de desenvolvimento se familiariza com a tecnologia que será utilizada no projeto. A fase de pesquisa pode durar semanas ou alguns meses, dependendo da complexibilidade do sistema e do grau de familiarização dos desenvolvedores com a tecnologia adotada.

Planejamento

O conteúdo da primeira versão é definido com base nas prioridades dos requisitos relacionados pelo cliente. A equipe de desenvolvimento estima o tempo de trabalho para implementar cada requisito, definindo o prazo de entrega da primeira versão. As histórias do usuário são transcritas em cartões de tarefa. Esses cartões são criados para dividir a implementação da história do usuário em pequenas partes tal que tempo dessas partes seja menor. Um cartão de tarefas pode conter tarefas relacionadas à mais de uma história.

Iterações

São feitas várias iterações de análise, projeto, implementação e testes antes da entrega da versão do sistema. O foco da primeira iteração deve estar em construir a arquitetura do sistema, ou seja, desenvolver o sistema da forma mais simples possível, de forma que sejam implementados os requisitos essenciais para seu funcionamento. À medida que versões são entregues, o cliente determina a prioridade de liberação das histórias restantes a serem implementadas.

Ao final de cada iteração são executados os testes funcionais descritos pelo cliente e um incremento é entregue.

Produção

Essa fase é iniciada após a última iteração. Os testes e verificações antes que o sistema seja entregue e implantado. Uma vez que detectada necessidade de modificações no sistema, tais modificações são submetidas a avaliação junto ao cliente,

verificando se deverão ser implementadas para a liberação, ou poderão ser implementadas na manutenção. Caso seja definido que as modificações devam ser implementadas posteriormente, elas devem ser documentadas para posterior implementação.

Manutenção

São realizados reparos necessários decorrentes de falhas encontradas, além da produção de novas iterações caso necessário e para futuras versões do sistema.

Morte

É a fase final de todo o processo. Essa fase ocorre quando não há mais histórias a serem implementadas, ou seja, todas as necessidades do cliente foram satisfeitas. É nessa fase que a documentação necessária é feita, sem que haja modificações na arquitetura, projeto e código.

Papéis e Responsabilidades

Os papéis do XP são definidos na Tabela 2.1.

Tabela 2. 1 - Papéis e responsabilidades do XP (BECK, 2004; ABRAHAMSSON et al., 2002).

Papéis	Responsabilidades
1. Programador	Escreve o código do sistema.
2. Cliente	Determina os requisitos do sistema, suas respectivas prioridades e escreve os casos de testes.
3. Testador	Auxilia o cliente na elaboração dos casos de teste funcionais. Regularmente executa os testes e divulga os resultados.
4. Rastreador	É o responsável por dar um feedback para toda a equipe. Ele verifica se a realidade traduz as estimativas, ou seja, verifica as estimativas corretas e tempo utilizado, visando melhorar as futuras estimativas.
5. Treinador	É o responsável pelo processo no geral. Deve elaborar regras para que a equipe utilize as práticas do XP e siga o processo corretamente.
6. Consultor	Não faz parte da equipe. É o responsável por dar suporte e ter conhecimento específico de alguma regra do negócio a ser implementado. Eventualmente pode ser consultado para auxílio.
7. Gerente	É o responsável por todo o projeto. Toma as decisões necessárias durante o processo e mantém relacionamento direto com a equipe para determinar o status do projeto, identificando as dificuldades e deficiência do mesmo.

Práticas XP

O XP adota e aplica as práticas contidas na Tabela 2.2 para orientação e auxílio ao processo de desenvolvimento durante seu ciclo de vida.

Tabela 2. 2 - Práticas do XP (BECK, 2004; LARMAN, BASILI, 2003; ABRAHAMSSON et al., 2002).

Prática	Descrição
1. Planejamento	A equipe faz as estimativas de tempo para implementação dos requisitos e com isso define um cronograma de entrega de cada iteração. A comunicação com o cliente é direta e constante, auxiliando nas decisões do projeto.
2. Entregas freqüentes	O foco estando no cliente, deve-se gerar entregas freqüentes das funcionalidades para que o cliente possa utilizá-lo rapidamente, atendendo suas necessidades em um fluxo constante.
3. Metáfora	O sistema é definido através de uma metáfora ou um conjunto de metáforas definidas pela equipe de desenvolvimento e o cliente. A metáfora é uma descrição de como deverá funcionar o sistema. Essa descrição define um conjunto de nomes que atuam como um dicionário a ser utilizado no domínio do problema no qual está se atuando, auxiliando na nomenclatura das classes e métodos do sistema.
4. Simplicidade	Procura-se projetar uma solução simples e funcional.
5.Desenvolvimento guiado por Testes	Toda a implementação é submetida a testes unitários. Esses testes são estabelecidos antes de qualquer implementação e são executados constantemente em cada modificação do sistema.
6. Refatoração	Constante reestruturação do código sem alterar sua funcionalidade, procurando otimiza-lo, removendo duplicação de código, melhorando seu entendimento e buscando simplicidade.
7. Programação em pares	Permite uma revisão constante do código que está sendo gerado, uma vez que há dois programadores juntos escrevendo o código.
8. Código de propriedade coletiva	Qualquer programador pode modificar o código em qualquer momento sem necessidade de autorização por parte do elaborador.
9. Integração continua	Cada funcionalidade implementada é integrada ao sistema. A cada integração todos os testes devem ser executados para assegurar que a atualização do sistema não tenha ocasionado algum problema.
10. 40 horas semanais	Deve-se trabalhar no máximo 40 horas semanais. A ocorrência de horas-extras significa que há um problema no decorrer do processo e o mesmo deve ser verificado e tratado. A ocorrência de trabalhos-extras desmotiva os integrantes da equipe.
11. Cliente presente	O cliente deve estar presente e disponível, participando do processo de desenvolvimento. O *feedback* do cliente é de extrema importância para o processo. O cliente presente esclarece dúvidas à medida que elas surgem no desenvolvimento e prioriza os requisitos a serem implementados.
12. Código padrão	Visando o entendimento do código por qualquer programador, é necessário que a codificação siga a padrões de programação.
13. Ambiente de trabalho	O ambiente de trabalho deve proporcionar uma boa comunicação entre os integrantes da equipe e facilitar a prática do desenvolvimento de software. A organização da equipe afeta diretamente a execução das práticas adotadas pelo XP.
14. Regras	A equipe pode definir suas próprias regras para seguir. Tais regras devem ter o consentimento de toda a equipe. A cada regra estabelecida é necessário avaliar suas conseqüências dentro do projeto. As regras podem ser modificadas a qualquer momento.

b) Scrum

Princípios

O método Scrum dá ênfase ao gerenciamento do projeto, não definindo práticas de ES. Suas práticas, voltadas para o gerenciamento e controle do processo, estão baseadas em teorias e experiências de controle de processos industriais (BACH, 1995; ABRAHAMSSON et al., 2002; SCHWABER, BEEDLE, 2001; MARTIN et al., 2003; HIGHSMITH, 2002; LARMAN, BASILI, 2003; ROOIJEN, 2006).

Assim como na metodologia XP, o Scrum tem como base equipes pequenas e projetos cujos requisitos tendem a ser modificados constantemente.

O Scrum considera que o processo de desenvolvimento de software possui várias variáveis como requisitos, tecnologia e recursos. O Scrum proporciona uma forma de trabalho flexível dentro de um ambiente no qual há constantes alterações nessas variáveis durante todo o projeto (BACH, 1995). À medida que a complexibilidade de tais variáveis aumenta, a probabilidade de sucesso no projeto diminui (BACH, 1995).

Sendo flexível e estando preparado para constantes alterações nessas variáveis, o Scrum faz com que a probabilidade de sucesso em sistemas com alto nível de complexibilidade seja maior (BACH, 1995).

O processo de desenvolvimento no Scrum é dividido em iterações denominadas *sprints*. Cada iteração tem duração de aproximadamente trinta dias. Toda a equipe trabalha sobre os requisitos definidos no início de cada iteração (ABRAHAMSSON et al., 2002; ROOIJEN, 2006).

O Scrum adota a realização de reuniões diárias para acompanhamento do projeto. Essas reuniões abordam o previsto e o realizado desde a ocorrência da última reunião. Durante esta reunião, as dificuldades encontradas são identificadas e resolvidas.

Processo

O processo Scrum, mostrado na Figura 2.4, é composto por três fases (ABRAHAMSSON et al., 2002; BACH, 1995; JONSSON, 2002; HIGHSMITH, 2002):

- Pré-planejamento.
- Desenvolvimento.
- Pós-planejamento.

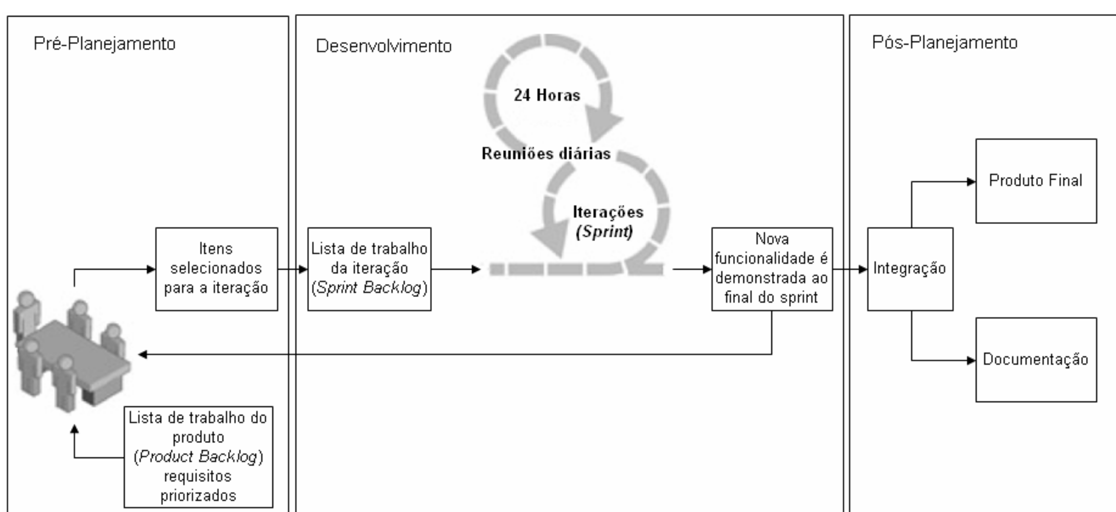

Figura 2. 4 - Visão geral do processo Scrum (adaptado de SCHWABER, 2004).

Pré-planejamento (Pré-game phase)

Esta fase é composta por duas etapas: planejamento e arquitetura.

Planejamento: Nessa etapa, os requisitos do sistema são informados pelo cliente e registrados em um documento denominado lista de trabalho do produto (*Product Backlog*). Após o levantamento dos requisitos, os mesmos são priorizados e a equipe faz estimativas para a implementação de cada requisito. A definição da equipe, ferramentas a serem utilizadas no projeto, treinamento e taxas de risco também são abordadas. As alterações de requisito que vierem a ocorrer são registradas na lista de trabalho do produto juntamente com os possíveis riscos decorrentes de tais modificações.

Arquitetura: Nessa etapa, o propósito é identificar detalhes dos requisitos e o que será requerido para que ele seja implementado. Nesse momento padrões, convenções, tecnologias e recursos necessários são definidos para as iterações (*sprints*).

Desenvolvimento (Game phase)

Durante a fase de desenvolvimento, as variáveis técnicas e de ambiente são constantemente controladas e observadas para garantir a flexibilidade em atender as mudanças que vierem a ocorrer.

O software é desenvolvido em iterações (*sprints*), nos quais novas funcionalidades são adicionadas à versão atual. Cada iteração implementa uma lista de trabalho da iteração (*sprint backlog*). Essa lista contém os requisitos selecionados para a iteração, os quais são extraídos da lista de trabalho do produto (*product backlog*). O desenvolvimento de cada iteração ocorre da forma tradicional, envolvendo os métodos de análise, projeto, implementação e testes. O desenvolvimento de cada iteração deve durar um período de uma semana até um mês.

Pós-planejamento (Post-game phase)

Após a fase de desenvolvimento, são feitas reuniões sobre o projeto e a versão atual é apresentada ao cliente. Nessa fase são feitos os testes na versão final e a documentação necessária é elaborada.

Papéis e Responsabilidades

O Scrum define seis papéis (ABRAHAMSSON et al., 2002) os quais estão descritos na Tabela 2.3.

Práticas Scrum

O Scrum não define práticas específicas de ES como o XP, por exemplo, mas possui práticas voltadas ao gerenciamento do processo. As práticas são apresentadas na Tabela 2.4.

Tabela 2. 3 - Papéis e responsabilidades do Scrum (ABRAHAMSSON et al., 2002).

Papel	Responsabilidade
1. Mestre Scrum (*Scrum máster*)	Líder do projeto, responsável por interagir com o cliente e a equipe de desenvolvimento, gerenciando essa comunicação durante todo o projeto. É também responsável por garantir que as práticas, valores e regras do Scrum sejam adotados e executados.
2. Proprietário do produto (*Product owner*)	Ele gerencia, controla e faz com que a lista de trabalho do produto (*product backlog*) seja respeitada. Ele é escolhido pelo mestre Scrum e pelo cliente.
3. Equipe scrum (*Scrum team*)	É a equipe de desenvolvimento que possui autoridade para se organizar e decidir suas ações para alcançar as metas de cada *sprint*, estando diretamente envolvida nas estimativas.
4. Cliente (*Customer*)	Participa das tarefas relacionadas à implementação da lista de trabalho do produto (*product backlog*).
5. Administrador (Administrator)	É responsável pelas decisões fundamentais e críticas.
6. Usuário (*User*)	Usuário do sistema.

Tabela 2. 4 - Práticas adotadas pelo Scrum (ABRAHAMSSON et al., 2002).

Prática	Descrição
1. Lista de trabalho do produto (*Product backlog*)	Consiste em uma lista que descreve os requisitos do software, funcionais ou não-funcionais, ordenada pela prioridade de cada requisito.
2. Estimar o esforço	Realizar uma análise mais precisa sobre a lista de trabalho do produto, estimando o esforço necessário para implementá-lo.
3. Iteração (*sprint*)	São as iterações realizadas durante o desenvolvimento. Cada iteração corresponde a um ciclo composto por variáveis de ambiente as quais são modificados a cada iteração.
4. Reunião de planejamento da iteração	Essa reunião é realizada antes de cada iteração e é composta por duas etapas. Na primeira etapa, a equipe, juntamente com o cliente, define os objetivos e funcionalidades da próxima iteração. Na segunda etapa, o mestre Scrum e a equipe definem como será feita a implementação da iteração. Esta reunião é organizada pelo mestre Scrum.
5. Lista de trabalho da iteração (*sprint backlog*)	É o ponto inicial de cada iteração (*sprint*). Consiste em uma lista contendo itens selecionados da lista de trabalho do produto (*product backlog*). Esta lista define o que será implementado em cada iteração. Esta lista é definida na reunião de planejamento da iteração.
6. Reunião diária	A reunião diária tem como objetivo verificar o progresso da equipe scrum durante o projeto. Nessa reunião são discutidas as realizações do dia anterior e o que será feito até a próxima reunião. A reunião diária tem duração de aproximadamente quinze minutos.
7. Reunião de revisão da iteração	Essa reunião é realizada no último dia de cada iteração, na qual o mestre Scrum e a equipe Scrum apresentam aos demais papéis os resultados obtidos na iteração. Essa reunião pode dar origem a novos itens na lista de trabalho do produto (*product backlog*), mesmo que para isso seja necessário alterar o planejamento do desenvolvimento.

c) Dinamic Systems Development Method (DSDM)

Princípios

O DSDM procura executar o projeto ajustando os requisitos do sistema ao tempo e aos recursos disponíveis, ao contrário do convencional, que tendo um conjunto de requisitos, procura ajustar o tempo e os recursos para alcançar a funcionalidade desejada.

O método inicia-se com um estudo de viabilidade e um estudo das regras de negócio. O estudo de viabilidade verifica se o DSDM é adequado ao projeto em questão. O estudo das regras de negócio é uma pequena série de *workshops* para entender e compreender o domínio sobre o qual os programadores irão trabalhar. Esse estudo também propõe esboços da arquitetura do sistema e um plano de projeto (DSDM; HIGHSMITH, 2002).

Processo

O processo DSDM é composto por cinco fases (ABRAHAMSSON et al., 2002; CLIFTON, DUNLAP, 2003; FOWLER, 2003; HIGHSMITH, 2002):

- Estudo de viabilidade.
- Estudo do negócio.
- Iteração do Modelo funcional.
- Iteração do Projeto e construção.
- Implementação.

A representação do ciclo de vida do DSDM pode ser vista na Figura 2.5.

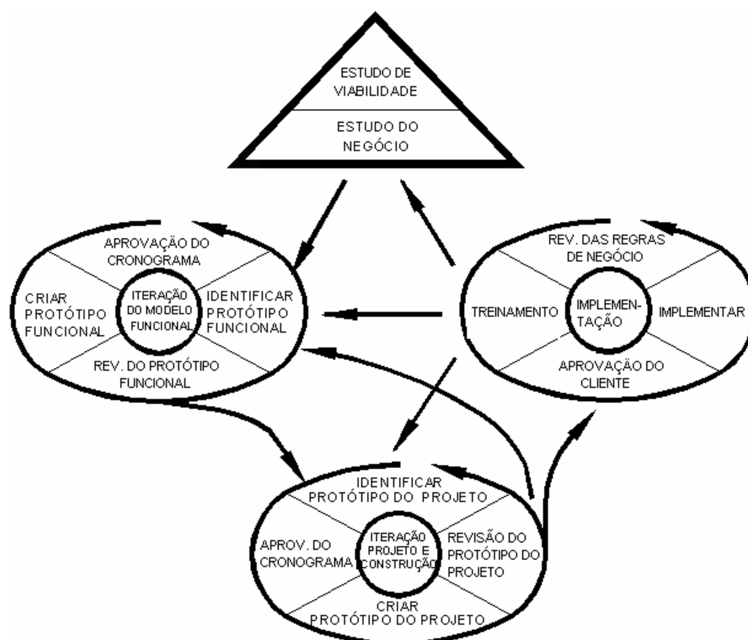

Figura 2. 5 - Processo DSDM (adaptado de ABRAHAMSSON et al., 2002).

As duas primeiras fases são seqüenciais. As demais fases são incrementais e iterativas.

Nas fases iterativas e incrementais, o DSDM utiliza o conceito de *timeboxes*, que é um período de tempo pré-estabelecido pela equipe com duração de alguns dias ou semanas. As iterações são associadas aos *timeboxes*, assim a duração e o resultado esperado de uma iteração ficam determinados nas fases iniciais.

As cinco fases do processo DSDM são descritas a seguir.

Estudo de Viabilidade

Essa fase consiste em verificar se o DSDM atende aos requisitos do projeto, ou seja, é verificada a viabilidade de uso do DSDM (FOWLER, 2003; ABRAHAMSSON et al., 2002). Também são verificadas as possibilidades técnicas, inclusive os riscos para o desenvolvimento do projeto. Nesse estudo são gerados o relatório de viabilidade e um esboço do planejamento do desenvolvimento (ABRAHAMSSON et al., 2002; HIGHSMITH, 2002).

Estudo do negócio

É feita a análise das regras de negócio juntamente com a tecnologia envolvida. É recomendada a realização de *workshops*, com o cliente, para levantar e discutir os pontos relevantes do projeto, bem como as prioridades de desenvolvimento. Nessa fase é definida a arquitetura a ser utilizada no desenvolvimento e é gerado um esboço do planejamento do protótipo a ser desenvolvido.

Iteração do Modelo Funcional

O Modelo funcional é a primeira fase iterativa e incremental. Essa fase pode ser subdivida em quatro etapas:

- *Identificar o protótipo funcional*: consiste em levantar os requisitos que serão implementados no protótipo da iteração.
- *Aprovar o cronograma das tarefas*: consiste em estabelecer prazos e critérios para implementação dos requisitos definidos para a iteração.
- *Desenvolver o protótipo funcional*: consiste em desenvolver o protótipo da iteração.
- *Revisão do protótipo funcional*: consiste em buscar a melhoria contínua, procurando por correções necessárias através de técnicas e critérios de teste.

Nessa fase é feita atualização no levantamento de requisitos, sendo que os requisitos já implementados são identificados e atribuem-se prioridades aos requisitos restantes.

Artefatos gerados nessa fase:

- Lista de prioridades dos requisitos que deverão ser atendidos até o fim da iteração.
- Documentos de revisão do modelo funcional, registrando e analisando os comentários dos usuários sobre o protótipo funcional para serem trabalhados nas fases subseqüentes.

- Lista de requisitos não-funcionais, principalmente os que forem tratados na próxima fase.

- Análise de risco para os novos desenvolvimentos, a qual auxiliará a identificar possíveis problemas na próxima fase, tornando-se mais fácil, caso ocorra algum erro, encontrar sua origem.

Iteração do Projeto e Construção

O objetivo principal dessa fase é integrar as funções implementadas na fase anterior, gerando um sistema que atenda as necessidades do cliente. Como na fase anterior, essa fase também é subdividida em quatro etapas:

- *Identificar o protótipo do projeto*: consistem em identificar os requisitos funcionais e não funcionais do sistema como um todo, ou seja, integrado. Tais requisitos serão utilizados nas atividades de teste.

- *Aprovar o cronograma de tarefas*: consiste em estabelecer prazos e critérios para a implementação dos requisitos levantados.

- *Desenvolvimento do protótipo do projeto*: consiste em gerar o sistema, integrando as funcionalidades implementadas, de modo que ele possa ser disponibilizado em ambiente de produção ao cliente.

- *Revisão do protótipo de projeto*: o sistema é revisado detalhadamente, utilizando as atividades de teste e inspeção.

Ao final dessa fase o protótipo do produto é apresentado ao cliente para avaliação e testes das funcionalidades.

Implementação

A fase de implementação consiste em implantar o sistema em ambiente de produção, ou seja, implantá-lo para sua utilização por parte do cliente. Nessa fase ocorrem os treinamentos necessários aos usuários e ela é subdividida em quatro etapas:

- *Aprovação do cliente*: consiste na aprovação do cliente para que as próximas etapas possam ser realizadas. São criados os roteiros para utilização do sistema e para a etapa de implementação desta fase.

- *Implementação*: consiste em transferir as bases, ou seja, o sistema sai do ambiente de desenvolvimento e é implantando em ambiente de produção.

- *Revisão das regras de negócio*: consiste em analisar o funcionamento do sistema na organização cliente, verificando se ele atende os requisitos especificados nas primeiras fases do processo. Essa análise define se o projeto segue para a próxima fase, denominada Pós-Projeto, ou na ocorrência de falhas com relação aos requisitos, retorna para as fases anteriores.

- *Treinamento*: consiste em treinar todos os usuários do sistema.

Ao final da fase de implementação, o sistema deve estar pronto para que o cliente comece a utilizá-lo. A documentação de utilização do sistema deve ser entregue.

Papéis e Responsabilidades

O DSDM define quinze papéis dos quais os principais estão relacionados na Tabela 2.5.

Tabela 2. 5 - Papéis e responsabilidades do DSDM (ABRAHAMSSON et al., 2002).

Papel	Descrição
1. Desenvolvedor e desenvolvedor sênior	São os únicos papéis vinculados ao desenvolvimento. O papel de desenvolvedor sênior é definido com base na experiência do desenvolvedor e nas tarefas a serem executadas.
2. Coordenador técnico	É responsável pela arquitetura do sistema e pela qualidade técnica do projeto.
3. Usuário embaixador	É o responsável por fazer o elo entre a equipe de desenvolvimento e demais usuários. É o representante dos usuários, fornecendo informações dos mesmos e os informando do status do projeto.
4. Usuário conselheiro	É responsável por determinados requisitos considerados importantes para o projeto.
5. Visionário	Possui maior conhecimento e domínio dos objetivos que o sistema deve atender, tendo em mente todo o seu funcionamento.
6. Diretor responsável	Responsável financeiro e por responsabilidades do lado do cliente.
7. Lideres de equipe	Lideram as equipes de desenvolvimento
8. Treinadores	Treinam

Práticas DSDM

O DSDM é dotado de dez práticas as quais são relacionadas na Tabela 2.6.

Tabela 2. 6 - Práticas do DSDM (CLIFTON et al., 2003).

Prática	Descrição
1. Manter o usuário ativo	O usuário deve estar presente e interagir durante o desenvolvimento do software.
2. Tomada de decisões por parte da equipe	A equipe, quando necessário, deve tomar decisões rápidas e divulgá-las, sem exigências de burocracia.
3. Realizar liberações freqüentes	A equipe deve estar preparada para liberações freqüentes do produto. Essa prática permite que os usuários utilizem o sistema e relatem os fatos ocorridos.
4. Promover desenvolvimento iterativo guiado pelos relatos do usuário	O desenvolvimento é realizado em iterações. Durante o desenvolvimento o contato com o cliente é fundamental. A cada iteração novas funcionalidades são adicionadas ao sistema.
5. Possibilitar mudanças reversíveis	Havendo problemas após mudanças no projeto, deve ser possível desfazê-las rapidamente.
6. Acompanhamento de requisitos complexos	Os requisitos definidos como complexo são analisados no início do projeto, antes de qualquer codificação.
7. Conhecer o objetivo do negócio	Entender as necessidades do negócio, bem como ter domínio sobre suas regras, é mais importante que a perfeição na parte técnica.
8. Testes integrados	Os testes são realizados a cada iteração, garantindo que, a cada implementação, o software não possui problemas.
9. Colaboração e cooperação	A colaboração e cooperação entre ambas as partes, equipe de desenvolvimento e cliente, são essenciais para o sucesso do projeto. Ambos devem trabalhar para alcançar o objetivo do negócio.
10. Regra 80% / 20%	O DSDM considera que um sistema perfeito não é obtido na primeira versão. O DSDM assume que 80% de todo o projeto pode ser realizado com 20% do tempo necessário para obter o projeto perfeito. O DSDM está focado nesses 80%, deixando os 20% restantes para revisões posteriores. O DSDM assume também que existem requisitos desconhecidos no início do projeto, dessa forma, os 20% restantes do projeto provavelmente serão falhos.

d) Feature Driven Development (FDD)

Princípios

O FDD é um método ágil de desenvolvimento de software focado nas fases de projeto e construção (implementação). Diferentemente de outros MA, o FDD dá suporte a sistemas complexos de grande porte (ABRAHAMSSON et al., 2002). Esse método provê um desenvolvimento iterativo, promovendo entregas freqüentes e um acompanhamento detalhado do progresso do projeto.

Processo

O FDD é composto por cinco fases, como mostra a Figura 2.6.

- Criação do modelo geral
- Construção da lista de funcionalidades
- Planejamento por funcionalidades
- Projeto por funcionalidades
- Construção por funcionalidades

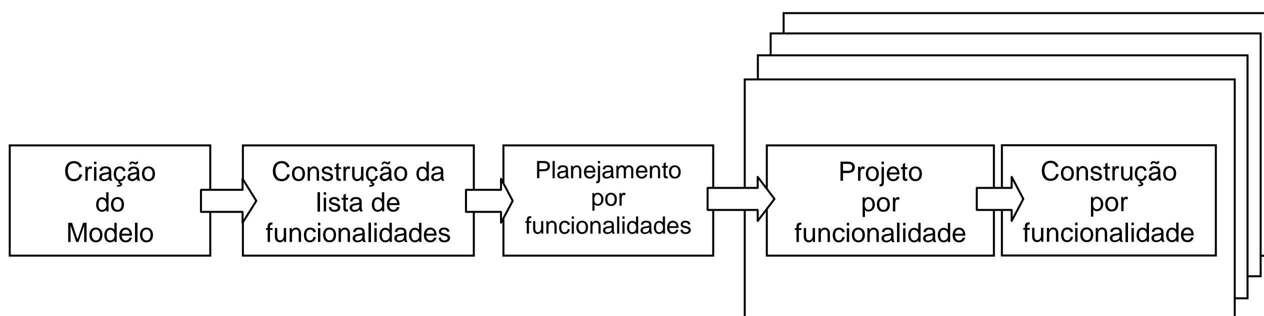

Figura 2. 6 - Processo FDD (adaptado de ABRAHAMSSON et al., 2002).

Na Figura 2.6, nota-se que as duas últimas fases, projeto e construção por funcionalidades, são iterativas. A iteratividade dessas fases é mostrada na Figura 2.7.

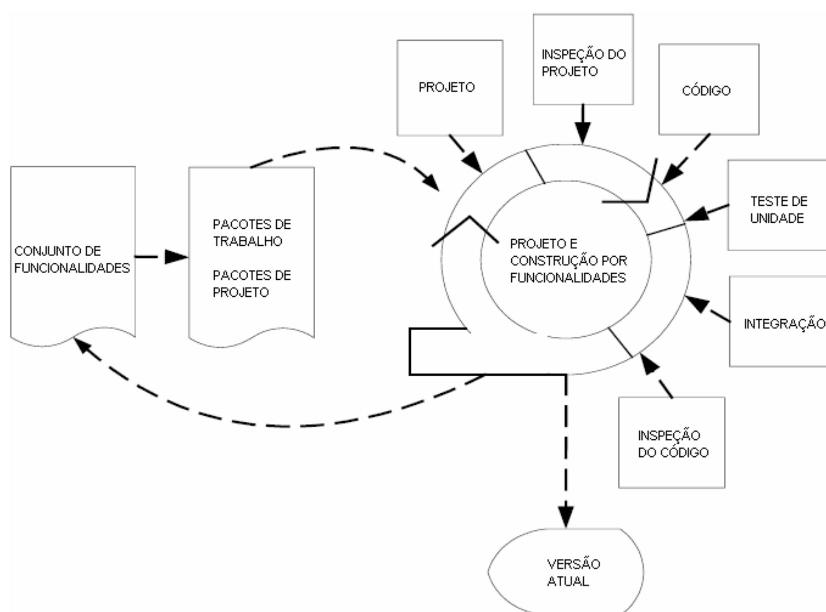

Figura 2. 7 - O projeto e construção por funcionalidades do FDD (adaptado de ABRAHAMSSON et al.,2002).

A seguir são descritas as cinco fases do processo FDD, mostradas nas Figuras 2.6 e 2.7.

Criação do modelo geral

Nessa fase é construído um modelo geral, o qual é composto pelas classes do sistema e o relacionamento que há entre elas (HIGHSMITH, 2002). Para sistemas de grande porte, são criados modelos por áreas do domínio, que são posteriormente, durante essa mesma fase, integrados. O responsável pela criação desses modelos é o arquiteto chefe.

Durante essa fase são levantados os requisitos do sistema, podendo existir o diagrama de casos de uso ou as especificações funcionais do sistema (ABRAHAMSSON et al., 2002). Os requisitos do sistema são então refinados, tornando-se funcionalidades no próximo processo.

Construção da lista de funcionalidades

Essa fase gera, com base na modelagem do processo anterior, uma lista com todas as funcionalidades do produto. Essas funcionalidades são semelhantes às histórias do usuário adotadas no XP (HIGHSMITH, 2002). A partir da modelagem da fase anterior, são identificadas as regras do negócio. Essas regras são decompostas em ações menores. Cada ação é uma funcionalidade do sistema. Essas funcionalidades, também denominadas *funções de valor para o cliente*, devem ser implementadas em até dez dias. Caso o tempo estimado ultrapasse esse período, essa ação deve ser decomposta, gerando outras funcionalidades (HIGHSMITH, 2002). Esse processo é gerenciado pelo programador-chefe, e o feedback ao cliente é constante.

Planejamento por funcionalidades

Nesse momento é estabelecida a ordem em que as funcionalidades serão desenvolvidas, a qual é gerada com base em fatores como dependências, riscos e complexibilidade.

A responsabilidade de cada regra de negócio é atribuída ao programador-chefe e a responsabilidade de cada classe é atribuída a um programador que exerce o papel do

proprietário de classe. Cada funcionalidade, consequentemente, é nomeada ao programador-chefe responsável pela regra de negócio.

Projeto por funcionalidades

O programador-chefe, responsável por um conjunto de funcionalidades, após analisar essas funcionalidades e as classes do sistema, cria pacotes de trabalho. Esses pacotes compreendem então um conjunto de funcionalidades que pertencem a um mesmo contexto técnico.

Com base nos pacotes de trabalho, o programador-chefe cria os pacotes de projeto, que são compostos por informações de projeto, tais como o diagrama de classe refinado e o de seqüência.

Construção por funcionalidades

Nessa fase, a equipe implementa cada pacote de projeto de acordo com os seguintes passos:

- Implementação das classes e métodos
- Inspeção do código
- Teste de unidade
- Integração das funcionalidades
- Atualizar a versão atual

As fases de projeto e construção por funcionalidades, citadas anteriormente, são iterativas. Essa iteração dura enquanto as funcionalidades selecionadas são implementadas (ABRAHAMSSON et al., 2002).

Papéis e Responsabilidades

Os papéis exercidos no FDD são classificados como papéis principais, de apoio e adicionais. Esses papéis são mostrados na Tabela 2.7.

Tabela 2. 7 - Papéis e responsabilidades do FDD (ABRAHAMSSON et al.,2002).

Papel	Responsabilidade	Categoria
1. Gerente do Projeto	Líder administrativo e financeiro do projeto. Deve garantir as condições de trabalho para a equipe de desenvolvimento.	Principal
2. Arquiteto-chefe	É responsável por todo o projeto do sistema, tomando decisões finais sobre qualquer assunto do projeto.	Principal
3. Gerente de desenvolvimento	Gerencia diariamente as atividades de desenvolvimento. É o responsável por resolver qualquer conflito que venha a ocorrer na equipe.	Principal
4. Programador-chefe	Lidera pequenas equipes na análise, projeto e desenvolvimento de novas funcionalidades. É responsável pela seleção das funcionalidades que serão desenvolvidas nas iterações do FDD.	Principal
5. Proprietário de Classe	Está subordinado ao programador chefe, atuando nas tarefas de projeto, codificação, testes e documentação. É o responsável pelo desenvolvimento das classes que foram atribuídas a ele. Os proprietários de classe formam a equipe de funcionalidades.	Principal
6. Especialistas do Domínio	São as pessoas que detêm maior conhecimento nas regras de negócio do sistema.	Principal
7. Gerente do Domínio	Lidera os especialistas do domínio. É responsável por resolver qualquer conflito de opiniões que possa ocorrer.	Apoio
8. Gerente de Liberação	Acompanha o progresso dos processos através dos relatórios do programador chefe. É o responsável por relatar o desempenho dos processos ao gerente de projeto.	Apoio
9. Especialista de linguagem	Corresponde ao membro da equipe com maior conhecimento na linguagem adotada para o projeto.	Apoio
10. Engenheiro de Construção	É o responsável por manter e realizar o processo de desenvolvimento, tendo como tarefas o controle de versões e a publicação da documentação.	Apoio
11. Ferramenteiro	Tem a responsabilidade de construir ferramentas de desenvolvimento, testes e migração de dados.	Apoio
12. Administrador do Sistema	É o responsável pela infra-estrutura da equipe. Ele deve configurar e gerenciar servidores, rede, estações de trabalho e ambientes de teste.	Apoio
13. Testador	Verifica se o produto desenvolvido está de acordo com os requisitos do cliente.	Adicional
14. Instaladores	É responsável por adequar dos dados existentes ao formato requerido pelo novo sistema. Esse papel pode ser assumido por um membro da equipe de desenvolvimento ou não.	Adicional
15. Escritor Técnico	É o responsável por preparar e redigir a documentação do sistema para os usuários, também podendo ser um membro da equipe de desenvolvimento ou não.	Adicional

Práticas FDD

O FDD adota as práticas relacionadas na Tabela 2.8.

Tabela 2. 8 - Práticas do FDD (ABRAHAMSSON et al., 2002).

Prática	Descrição
1. Modelagem do Objeto do Domínio	Exploração e explicação dos requisitos do sistema, nos quais funcionalidades podem ser incluídas.
2. Desenvolvimento por funcionalidade	Desenvolvimento e rastreabilidade através de uma lista de pequenas funcionalidades de valor ao cliente.
3. Propriedade de Classe individual	Cada classe do sistema possui seu responsável.
4. Equipes de funcionalidades	Consistem em montar pequenas equipes de proprietários de classe.
5. Inspeção	Adotar as técnicas de inspeção mais adequadas ao projeto.
6. Freqüência de versões	Garantir a disponibilidade de um demonstrativo do sistema funcionando.
7. Gerenciamento de Configuração	Garantir a identificação e rastreabilidade das versões de cada código-fonte finalizado.
8. Relatório de Progresso	Registrar o progresso dos processos. Essa informação é importante para todo o projeto.

e) Adaptive Software Development (ASD)

Princípios

O método ASD está voltado para o desenvolvimento de sistemas complexos e de grande porte (ABRAHAMSSON et al., 2002). Esse método procura trabalhar as mudanças, decorrentes de ambientes turbulentos, ao invés de confrontá-las. É iterativo e incremental, tendo o cliente sempre presente. Suas práticas estão focadas nessa adaptação às mudanças (HIGHSMITH, 2002).

Processo

O processo do ASD, classificado como um processo orientado a mudanças (HIGHSMITH, 2002; ABRAHAMSSON et al. 2002, HIGHSMITH, 2000), é composto pelas seguintes fases.

- Especulação
- Colaboração
- Aprendizagem

A *Especulação* procura enfatizar as possíveis mudanças no planejamento, uma vez que o planejamento foi elaborado com base na análise e investigação de um problema.

A *Colaboração* denota a importância da equipe para a adaptação às mudanças durante o processo e a simultaneidade na construção dos componentes.

A *Aprendizagem* denota dois sentidos, a importância do reconhecimento e reflexão sobre os erros e a conscientização de que os requisitos são alterados durante o processo de desenvolvimento. As análises e revisões são feitas em grupo, com foco nos interesses do cliente.

A Figura 2.8 mostra o processo ASD.

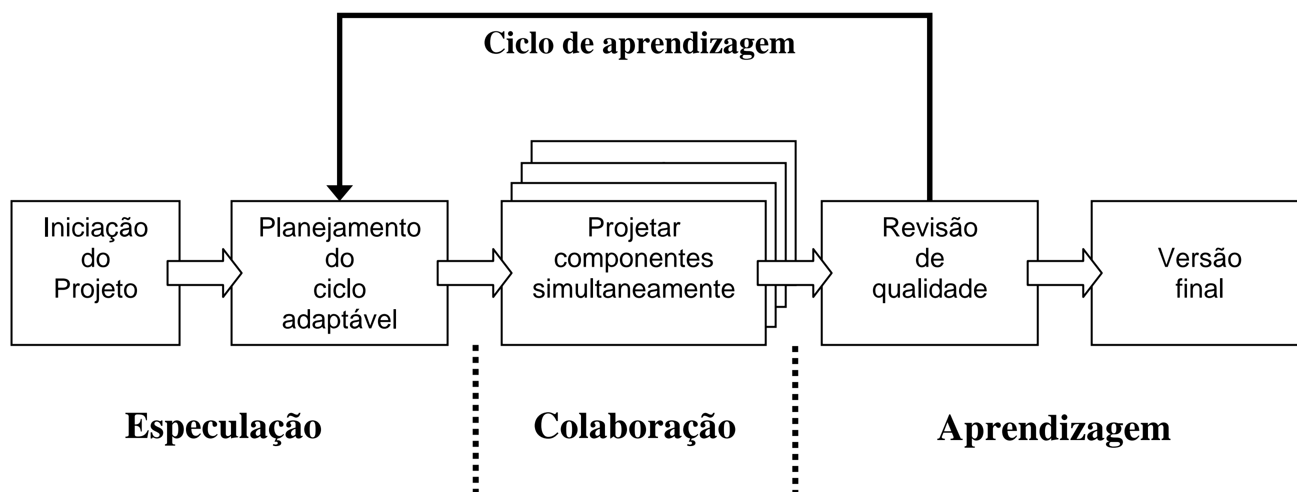

Figura 2. 8 - Processo ASD (adaptado de ABRAHAMSSON et al., 2002).

Através da Figura 2.8, notam-se as três fases principais das iterações (*especulação*, *colaboração* e *aprendizagem*), e nelas distribuídas as etapas do processo, as quais são descritas a seguir (HIGHSMITH, 2002; ABRAHAMSSON et al. 2002):

Iniciação do projeto: Nessa etapa são definidos pontos iniciais do projeto, tais como a missão e objetivo, organização do projeto, limitações, identificação dos requisitos, riscos e esforço necessário.

Para projetos de pequeno e médio porte, esse processo dura entre dois e cinco dias. Para projetos de grande porte, esse processo pode consumir de duas a três semanas.

Planejamento do ciclo adaptável: Nesse planejamento define-se um prazo (*time-boxed*) para todo o sistema. Também é definida a quantidade de iterações para o projeto e, para cada iteração, é definido seu prazo, o qual varia entre quatro e oito semanas de acordo com o porte do projeto. Para cada iteração define-se um objetivo.

Os desenvolvedores e os clientes são os responsáveis por determinar os componentes de cada iteração. O critério mais importante para essa seleção é o fato de que a iteração deve entregar um conjunto de componentes, cujo resultado deve ser visível ao cliente. É o cliente que define a prioridade dos componentes a serem produzidos.

Cada iteração entrega um conjunto de componentes (pedaços da aplicação) visíveis ao cliente, para que esse possa validá-lo. A integração deve ser freqüente para que o produto possa ser mostrado à equipe e ao cliente.

Os testes fazem parte do desenvolvimento das funcionalidades e não são executados somente ao fim da implementação.

Projetar componentes simultaneamente: Os gerentes de projeto facilitam a colaboração e as atividades de desenvolvimento simultâneas. Em projetos nos quais a equipe toda de desenvolvimento está localizada no mesmo espaço físico, conversas no corredor e até um quadro podem ser considerados como colaboração. Quando a equipe está distribuída em vários locais distantes um do outro, devem ser adotadas atividades e ferramentas para gerenciar essa comunicação.

Revisão de qualidade: Ao fim de cada iteração deve ser verificado:

- A qualidade do produto do ponto de vista do cliente
- A qualidade do produto do ponto de vista técnico
- O progresso da equipe e as práticas adotadas pela equipe
- O status do projeto

Essas informações promovem uma verificação do andamento do projeto, atuando como um embasamento para as próximas iterações. Critérios e práticas adequadas devem ser adotados.

Versão final: Não é propriamente uma fase, mas sim um estágio no qual são realizados os testes na versão atual e também são registrados os *feedbacks* do cliente. Nessa etapa é decidido se as atividades de teste sobre a versão atual devem continuar ou não. Caso seja necessário, uma nova iteração pode ser iniciada.

Nos processos apresentados, seis características devem estar presentes (HIGHSMITH, 2002). São elas:

- **Foco na missão**: A missão define o objetivo do projeto. Sem uma missão, ou seja, um objetivo principal, um ciclo de vida iterativo, que tem por objetivo uma produção constante, passa a ser improdutivo, ou seja, emprega-se esforço ao projeto, porém esse não traz resultados.

- **Base nos componentes**: O ciclo de vida ASD é voltado aos resultados obtidos através do método e não da execução de tarefas. Esses resultados são identificados como componentes da aplicação, ou seja, pedaços da aplicação, os quais compreendem as funcionalidades.

- **Iteratividade**: Os métodos tradicionais são bem sucedidos em ambientes bem definidos, não sujeito as mudanças durante os processos. Em ambientes turbulentos, o foco deve estar na refatoração e na adaptação às mudanças que venham a ocorrer.

- **Intervalo de tempo definido (time-boxed)**: A utilização de desses prazos forçam os membros da equipe a tomar as decisões críticas e inevitáveis no início do processo, procurando resolver essas situações o mais rápido possível, procurando evitar que tais mudanças afetem o projeto.

- **Orientação a Riscos**: O desenvolvimento de funcionalidades críticas, ou seja, de alto risco, deve ser iniciado assim que possível. Esse procedimento auxilia na adaptação aos possíveis problemas que possam ocorrer.

- **Tolerância a mudanças**: Mudanças são constantes no desenvolvimento de software, logo a adaptação a essas mudanças agrega mais ao projeto do que tentar controlar o surgimento delas.

Papéis e responsabilidades

O método ASD cita apenas como papéis o responsável executivo e os participantes de sessões (*workshops*). O responsável executivo é o responsável geral do processo, e os participantes das sessões (workshops) de desenvolvimento correspondem ao facilitador, o escriba e o cliente. O Facilitador é responsável pelo planejamento dos workshops, o Escriba é o responsável por fazer as anotações e o cliente está sempre presente. Esse método não define detalhadamente os papéis e responsabilidades que atuam no projeto (ABRAHAMSSON et al., 2002).

Práticas

As práticas identificadas no processo ASD são mostradas na Tabela 2.9.

Tabela 2. 9 - Práticas do ASD (ABRAHAMSSON et al., 2002).

Prática	Descrição
1. Desenvolvimento iterativo	Promover a iteratividade com adaptação às mudanças que venham a ocorrer.
2. Planejamento com base em componentes	Planejar o projeto com base em componentes, ou seja, pedaços da aplicação final.
3. Análise em grupo com foco no cliente	As análises e revisões são realizadas em grupo tendo o cliente sempre presente. Essas atividades focam os interesses do cliente.

f) Família de Métodos Crystal

Princípios

A Família *Crystal* corresponde ao seguinte conjunto de métodos:

- *Crystal Clear*
- *Crystal Yellow*
- *Crystal Orange*
- *Crystal Red*

A escolha de qual método aplicar varia de acordo com cada projeto, dependendo de seu tamanho e sua criticidade (COCKBURN, 2006; ABRAHAMSSON et al., 2002; HIGHSMITH, 2002). A Figura 2.9 ilustra como é feita essa verificação.

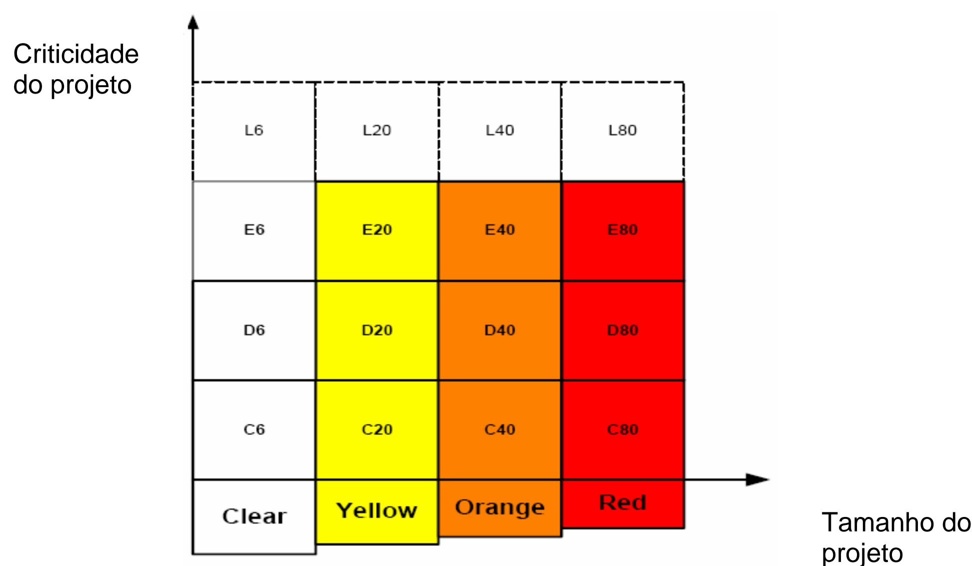

Figura 2. 9 - Métodos *Crystal* e suas dimensões (ABRAHAMSSON et al., 2002).

Cada elemento entre os eixos de criticidade e tamanho do projeto, conforme a Figura 2.9, corresponde a uma categoria a qual é identificada através de sua criticidade e quantidade máxima de pessoas que envolvem o projeto. A criticidade do projeto é classificada conforme os itens:

- Conforto (C)
- Baixo Custo (D)

- Alto Custo (E)
- Risco de vida (L)

Através dos elementos contidos na Figura 2.9, determina-se o método que mais se aplica ao projeto. Tomando como exemplo o elemento D6, esse é interpretado como sendo classificado com risco D e com seis pessoas envolvendo o projeto.

Pontos comuns aos métodos da família *Crystal* (COCKBURN, 2006):

- A iteratividade com desenvolvimento incremental
- Tempo de duração de cada ciclo, normalmente até quatro meses
- Ênfase na comunicação e cooperação entre pessoas
- Não impõem práticas, ferramentas ou artefatos, ou seja, alta tolerância.

A seguir, são abordados os métodos *Crystal Clear* e *Crystal Orange*. Os métodos *Crystal Yellow* e *Crytal Red* não são descritos porque, no estudo realizado, não foram encontradas referências da utilização desses dois métodos.

Método *Crystal Clear*

O método *Crystal Clear* é destinado a projetos em que a equipe é composta por, no máximo, seis pessoas e seu risco é classificado como D (baixo custo). Cada incremento gerado no projeto, executado sob esse método, tem duração de dois a três meses. As especificações e o projeto são feitos informalmente, através de esboços e anotações.

Processos

As características da metodologia *Crystal* no método *Crystal Clear* são (COCKBURN, 2006; HIGHSMITH, 2002):

- **Políticas**: As políticas adotadas no método *Crystal Clear* compreendem a entrega incremental a cada dois ou três meses, participação direta do cliente, testes automatizados, participação de dois usuários para validação dos incrementos e revisão dos processos através de workshops, o processo é

rastreado por pontos que consistem em entrega de versões ou decisões importantes.

- **Artefatos**: Os artefatos adotados são os planos de liberação de incrementos e do sistema para validação e entrega, cronograma para a verificação dos usuários e entrega, esboço do projeto e da interface do sistema bem como notas necessárias, código executável, casos de teste, casos de uso e descrição das funcionalidades e código de migração caso necessário.
- **Responsabilidades**: Os papéis do método *Crystal Clear* são apresentados a seguir.

Papéis e Responsabilidades

O *Crystal Clear* possui quatro papéis principais (CORCKBURN, 2006). Esses papéis são descritos na Tabela 2.10.

Tabela 2. 10 - Papéis e responsabilidades do método *Crystal Clear* (ABRAHAMSSON et al., 2002).

Papel	Descrição
1. Responsável do Projeto	Responsável administrativo e financeiro do projeto. Deve garantir as condições de trabalho para a equipe de desenvolvimento e gerenciar todos os processos
2. Projetista-programador sênior	É o líder dos programadores. O papel de desenvolvedor sênior é definido com base na experiência do desenvolvedor.
3. Projetista-Programador	Executa as atividades pertinentes a ES (análise, projeto, codificação e outros)
4. Usuário	Usuário do sistema.

Método *Crystal Orange*

O método *Crystal Orange* abrange projetos que não envolvem risco de vida e são formados por mais de uma equipe, as quais são organizadas de acordo com as responsabilidades adotadas (planejamento, tecnologia e outros). O projeto envolve entre dez e quarenta pessoas. Cada incremento gerado tem duração de dois a quatro meses e o ciclo de vida total do projeto dura de uma a dois anos.

Processos

No método *Crystal Orange*, as características da metodologia são (COCKBURN, 2006; ABRAHAMSSON et al., 2002):

- **Políticas**: As políticas do *Crystal Orange* são as mesmas do *Crystal Clear*.
- **Artefatos**: O método *Crystal Orange* adota como artefatos o plano de liberação detalhado, seqüências de entrega, cronograma, documentação do projeto de interface, modelo de objetos, manual, código-fonte, casos de teste, plano de migração, documento de requisitos, documentação de especificações internas da equipe e relatórios de status do projeto.
- **Negócios Locais**: Correspondem a procedimentos da metodologia *Crystal* que precisam ser aplicados, compreendendo o uso de templates para os produtos do trabalho.
- **Ferramentas**: Corresponde às ferramentas utilizadas durante o processo de desenvolvimento.
- **Padrões**: São os padrões de notação, convenções do projeto e padrões de qualidade adotados no projeto.
- **Atividades**: São as atividades executadas durante o processo de desenvolvimento.

A Figura 2.10 mostra as atividades que envolvem um incremento do processo do *Crystal Orange*.

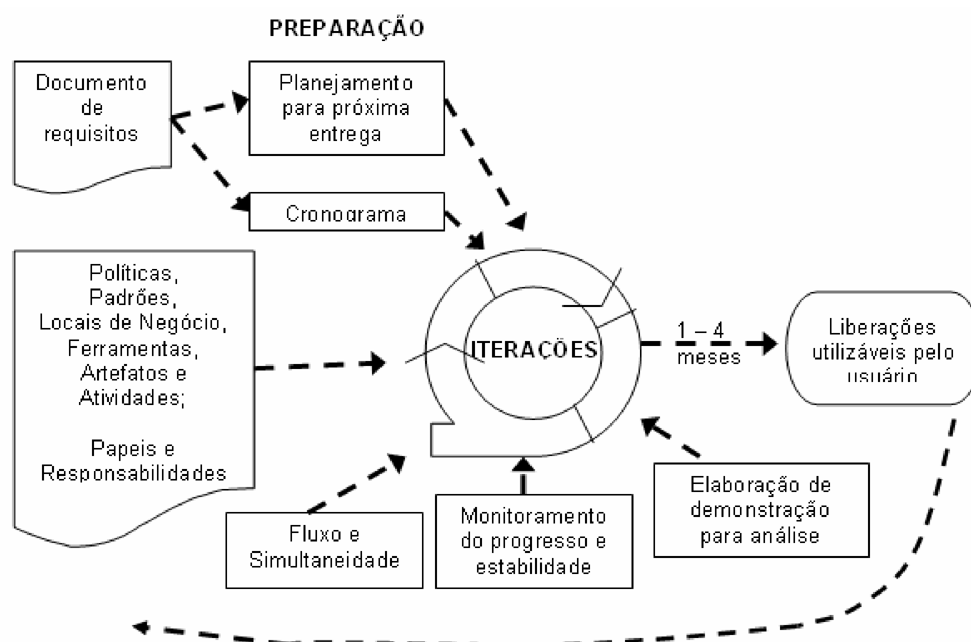

Figura 2. 10 - Uma iteração do *Crystal Orange* (adaptado de ABRAHAMSSON et al., 2002).

Papéis e Responsabilidades

O método *Crystal Orange* adota os mesmos papéis do método *Crystal Clear* e adicionalmente os relacionados na Tabela 2.11.

Tabela 2. 11 - Papéis e responsabilidades do método *Crystal Orange* (COCKBURN, 2006).

Papel	Descrição
1. Projetista de interface	Responsável pelas interfaces do projeto
2. Projetista da base de dados	Responsável pela base de dados do projeto
3. Especialista do domínio	Responsável que detêm maior conhecimento nas regras de negócio do sistema
4. Facilitador técnico	Lidera e planeja workshops quando necessário. Esses workshops podem ser relativos a tecnologia ou às regras de negócio.
5. Analista/projetista das regras do negócio	mantem contato com o usuário, especificando os requisitos do sistema ou possíveis revisões no projeto
6. Arquiteto	É responsável por todo o projeto do sistema, tomando decisões finais sobre qualquer assunto do projeto.
7. Responsável pelo reuso	Gerencia o reuso de componentes de software e a aquisição de novos componentes
8. Redator	Responsável por redigir documentos externos, como o manual do usuário
9. Testador	Responsável pelos casos de teste e automação dos casos de teste
10. Gerente do Projeto	Gerente da equipe.

Práticas adotadas nos métodos *Crystal Clear* e *Crystal Orange*

Ambos os métodos *Crystal Clear* e *Crystal Orange* não definem práticas especificas. Práticas adotadas em outros métodos podem ser adotadas quando necessário (ABRAHAMSSON et al., 2002).

As práticas da Tabela 2.12 são identificadas nos métodos *Crystal Clear* e *Crystal Orange*, respeitando-se as características de cada método, como por exemplo o tempo das iterações.

Tabela 2. 12 - Práticas da metodologia *Crystal* (COCKBURN, 2006).

Prática	Descrição
1. Preparação	Planejamento das iterações, ou seja, um cronograma para produzir uma versão utilizável. A equipe seleciona as funcionalidades a serem implementados na iteração e, sobre elas, montam o cronograma.
2. Monitoramento	O progresso do projeto é monitorado. Verifica-se o que a equipe foi capaz de realizar. O monitoramento é feito em pontos de medição (inicio, revisão1, revisão 2, teste, entrega).
3. Revisão e Análise	Cada incremento ao projeto constitui um conjunto de iterações sendo que, nessas iterações, são realizadas as atividades de implementação, demonstração e revisão dos objetivos do incremento.
4. Fluxo e simultaneidade	No uso de várias equipes (*Crystal Orange*), elas podem proceder com máximo de paralelismo, permitindo o trabalho de várias equipes ao mesmo tempo.
5.*Holistic Diversity Strategy*	Consiste em elaborar estratégias para que a equipe seja multifuncional.
6. Refinamento de Metodologia	Uso de entrevistas e workshops para compreender e instanciar uma metodologia do crystal. A idéia central é que, a cada incremento, seja possível fixar e melhorar o processo de desenvolvimento.
7. Visões do usuário	Consiste em ter o usuário participando do desenvolvimento, analisando e validando cada entrega realizada.
8. Workshops de análise (reflexão)	São realizados antes e depois do incremento

2.5 Considerações finais

A metodologia ágil, embora muito evidenciada somente em trabalhos acadêmicos (MELNIK, MAURER, 2005), ganha cada vez mais destaque na ES. Sua utilização está diretamente relacionada ao contexto do projeto, sendo que, em determinadas situações, os métodos tradicionais prevalecem (WILLOUGHBY, 2005).

Neste capítulo foram abordados a MADS e os MA mais evidenciados no estudo realizado. A essência da Metodologia ágil foi apresentada assim como os pontos que a diferem dos métodos tradicionais. Para os métodos abordados, identificaram-se suas principais características e forma como esses trabalham o desenvolvimento de software.

Nos próximos capítulos, são abordadas as atividades de garantia de QS, envolvendo as atividades de testes, inspeção e planejamento.

CAPÍTULO 3

ATIVIDADES DE GARANTIA DE QUALIDADE DE SOFTWARE

3.1. Considerações iniciais

Como o próprio nome diz, as Atividades de Garantia de QS são aquelas que devem ser aplicadas ao longo de todo desenvolvimento, com o objetivo de garantir qualidade em cada fase do desenvolvimento. Dentre elas, as atividades de VV&T (Validação, Verificação e Teste) e o Planejamento do software são atividades muito importantes e são tratadas no contexto deste trabalho.

A ocorrência de erros, sejam eles relacionados diretamente ao produto ou relacionados às tarefas organizacionais, prejudica o cliente que está adquirindo tal produto e a empresa que o está fornecendo. Esses prejuízos são tanto financeiros quanto em relação à confiança entre cliente e organização.

Na ES, o custo gerado por um erro no produto aumenta à medida que esse erro se propaga no ciclo de vida do software, ou seja, quanto mais tarde é descoberto, maior será o prejuízo.

Os erros durante o desenvolvimento de software ocorrem independentemente do paradigma de desenvolvimento adotado. Os erros estão propícios porque há pessoas interagindo com o projeto, ferramentas adotadas podem estar com *bugs* e até mesmo fatores externos podem ocasionar erros durante o processo.

Neste capítulo são abordadas atividades de garantia de QS como atividades de inspeção, teste e planejamento, pois essas atividades compõem a abordagem proposta neste trabalho.

Este capítulo está organizado da seguinte forma: na Seção 3.2 são apresentados os conceitos de QS; na Seção 3.3 é explorado o conceito de testes no desenvolvimento de software; Na Seção 3.4, assim como na Seção 3.3, são explorados os conceitos de inspeção de software; na Seção 3.5 são abordados o conceito de planejamento no desenvolvimento de software e a técnica de pontos por caso de uso (KARNER, 2003), que é utilizada para estimar o esforço a ser empregado no projeto; na Seção 3.6 encontram-se as considerações finais.

3.2. Qualidade de Software

As atividades de garantia de qualidade são essenciais no processo de desenvolvimento de software.

Qualidade de software (QS) é estar em conformidade com os requisitos, tanto funcionais quanto não funcionais, tendo as características implícitas e explícitas do sistema atendidas (PRESSMAN, 2006). Nesse contexto, três fatores são destacados:

- A ocorrência de não-conformidade com os requisitos é falta de qualidade.
- O desenvolvimento “alheio”, não seguindo critérios e métodos, resultará em falta de qualidade.
- O software que não atende aos requisitos implícitos, ou seja, o que está subentendido, tem sua qualidade suspeita.

A QS é composta por um conjunto de fatores que variam de acordo com o contexto no qual se está trabalhando. Para um sistema on-line, o tempo de processamento da informação é requisito fundamental e pode afetar diretamente a qualidade do sistema, enquanto um sistema de gestão pode não ter o tempo de processamento como um fator que irá afetar sua qualidade diretamente (PRESSMAN, 2006). A QS pode ser analisada conforme a perspectiva com a qual é vista (FALBO, 2005):

Visão do usuário: software com qualidade é o que atende suas necessidades, é confiável e fácil de usar.

Visão do desenvolvedor: software com qualidade é aquele que foi desenvolvido de forma que sua manutenção seja rápida, eficaz e eficiente, atendendo as necessidades de seu cliente.

Visão do cliente: software com qualidade é o que agrega valor à organização e proporciona o retorno do investimento nele aplicado.

Mesmo sendo subjetivo e dependente diretamente do contexto, a QS possui três fatores que também devem ser considerados para proporcionar qualidade (FILHO,RIOS, 2003):

- O número e a gravidade dos defeitos residuais do processo de testes devem ser aceitáveis pela organização.
- O software deve ser entregue dentro do prazo e custo previstos, atendendo aos requisitos e expectativas do cliente e do desenvolvedor.
- O software deve ser projetado de forma que, quando identificado um defeito após sua implantação, sua manutenção seja realizada de forma eficiente.

Um sistema em pleno funcionamento é resultado do trabalho conjunto de todos os envolvidos em seu processo de desenvolvimento. Dessa forma, cada envolvido não pode ver a QS como algo que atenda somente as suas próprias necessidades, mas sim como um objetivo comum, que todos se empenham para alcançá-lo (MOLINARI, 2005).

Uma maneira de promover a QS é elaborar o planejamento do projeto a ser realizado e adotar atividades de Validação, Verificação e Testes (VV&T). Essas práticas (planejamento e atividades de VV&T) ajudam a mostrar e eliminar possíveis riscos, de qualquer natureza, aos quais o processo de desenvolvimento de software está suscetível. Iftikhar (2004) cita que as atividades de VV&T proporcionam qualidade ao software, maior estabilidade nos requisitos e tornam o plano de desenvolvimento mais eficaz.

Um fato decisivo para o sucesso de todo o projeto é garantir um mínimo de qualidade ao software. O custo para prover tal garantia, através de atividades VV&T, varia entre 50 e 75% do custo total do desenvolvimento do produto (HAILPERN; SANTHANAM, 2002). A QS depende do investimento feito no controle da qualidade (FILHO; RIOS, 2003).

Segundo Pressman (2006), o custo e investimento relacionados à QS são rateados em prevenção, avaliação e falhas. O custo com a prevenção disponibiliza recursos para as atividades que abrangem o planejamento, revisões técnicas, treinamentos e equipamentos para teste. A avaliação compreende a análise do produto na execução do projeto pela primeira vez. A falha compreende o custo gerado para encontrar a natureza de um erro e corrigi-lo.

As atividades de VV&T são atividades de garantia da qualidade, vitais para o sucesso do processo de desenvolvimento (IFTIKHAR, 2004). Assim como as atividades de VV&T, o planejamento do projeto também está relacionado à garantida de QS. Entende-se por Garantia da Qualidade um conjunto de atividades para auditar, relatar e avaliar as atividades de controle da qualidade, que por sua vez, envolve atividades de revisão, teste e inspeção, durante o processo de desenvolvimento (PRESSMAN, 2006). Vários autores (SOMMERVILLE, 2004; PRESSMAN, 2006; HETZEL, 1987) definem VV&T da seguinte maneira:

Validação: avalia se o software desenvolvido atende às expectativas do cliente.

Verificação: avalia se o software desenvolvido está em conformidade com os padrões previamente estabelecidos e está sendo construído corretamente.

Teste: examina se o comportamento do software está correto, através de sua execução.

Nas próximas seções apresentam-se os principais conceitos sobre as atividades de teste, inspeção e planejamento no contexto de desenvolvimento de software.

3.3. Teste

Hetzel (1987) define a atividade de teste como um processo utilizado para adquirir confiança em um software, mediante suas ações e os resultados que dele se espera. O mesmo autor comenta a definição de Myers, a qual considera que as atividades de teste têm como objetivo encontrar erros na execução do software.

Pressman (2006) e Sommerville (2004) citam a definição de Myers e reafirmam que o objetivo principal da atividade de teste é encontrar erros, e um bom teste é o que tem grande probabilidade de encontrá-los. Os erros estão implícitos ao desenvolvimento de software (MOLINARI, 2005).

As atividades de testes fazem parte do processo de desenvolvimento de software e portanto, não podem deixar de serem realizadas (TELES, 2004). No entanto, tais atividades, embora essenciais, muitas vezes acabam não sendo realizadas adequadamente ou até mesmo não realizadas. Isso é decorrente de:

- As atividades de teste demandam tempo, o qual normalmente é escasso quando se trata de desenvolvimento de software.
- Para uma atividade de teste eficaz exigem-se integrantes da equipe voltados para tal atividade.
- A equipe não domina o conceito, técnicas e critérios de teste envolvidos.
- A atividade não é valorizada por natureza. A equipe só valoriza as atividades de teste quando os problemas começam a aparecer com o sistema já em funcionamento no cliente.
- O atraso no cronograma da equipe faz com que as atividades de testes, em geral previstas para o fim do desenvolvimento, acabem por não serem feitas.

Quando ocorre um erro na aplicação, automaticamente é gerado custo tanto para o desenvolvedor quanto para o cliente. Para o desenvolvedor, pois terá que desprender recursos para descobrir a origem do problema e corrigi-lo, sem mencionar a falta de credibilidade que é gerada junto ao cliente. Para o cliente, pois ocorre atraso na ferramenta de apoio aos negócios da empresa.

Quanto mais próximo do fim do ciclo de vida do processo de desenvolvimento um problema é encontrado, maior será o custo para corrigi-lo. O relatório "*The Economic Impact of Inadequate Infraestruture for Software Testing*" (RTI, 2002) mostra essa realidade. Esse relatório apresenta informações referentes ao custo gerado para correção de erros encontrados no desenvolvimento de software. Erros encontrados no software já em utilização pelo cliente, geram um custo de correção que pode chegar a ser trinta vezes maior que o custo que seria gerado caso esses erros fossem encontrados nas primeiras fases do desenvolvimento. O custo de correção de um erro está relacionado ao momento em que o erro é encontrado. Um erro encontrado em fases posteriores à fase em que foi gerado, tem o seu custo de correção aumentado conforme as fases que se passaram, ou seja, quanto mais tardia é a descoberta do erro, maior será o custo gerado (RTI, 2002).

Para que qualquer tarefa seja bem realizada, ela deve ser planejada e organizada antes de sua execução. Com a atividade de teste não é diferente. A atividade de teste possui seu próprio ciclo de vida o qual corresponde a quatro etapas (PRESSMAN, 2006; CRESPO et al., 2004; MALDONADO et al.,1998; FALBO, 2005):

Planejamento de teste: Elaboração de um plano de trabalho detalhado envolvendo a definição das atividades, investimentos necessários e objetivos. Nesta etapa também são definidos o tipo de teste, o contexto em que os testes serão aplicados e a técnica, com os respectivos critérios, a ser utilizada (CRESPO et al.,2004).

Projeto de casos de teste: Nessa etapa são gerados os casos de teste. Um caso de teste é um par ordenado (*e*,*s*), sendo *e* o dado de entrada e *s* a saída esperada. Os casos de teste são gerados com base nas técnicas e critérios de teste (FILHO;RIOS, 2003; MOLINARI, 2005).

Execução dos testes: Compreende a execução dos casos de teste, coletando seus resultados e os registrando adequadamente.

Avaliação dos resultados obtidos: Analisar os resultados provenientes dos casos de teste. Na ocorrência de falhas, deve-se descobrir sua origem e corrigi-las. Na ausência da mesma, os casos de teste devem ser avaliados.

Para a geração dos casos de teste, três fatores devem ser verificados: o tipo de teste a ser adotado, o contexto em que serão aplicados e a técnica, com seus respectivos critérios, que será utilizada para gerar os casos de teste.

O tipo de teste define quais fatores de qualidade do software serão testados (PRESSMAN, 2006). São tipos de teste: teste de funcionalidade; teste de carga; teste de segurança; teste de interface; teste de usabilidade; teste de desempenho e teste de volume.

O contexto compreende em quais níveis da implementação os testes serão aplicados. Esse contexto pode ser: teste de unidade; teste de integração, teste de sistema e teste de aceitação. A seguir, com relação ao contexto de aplicação, são abordados os testes de unidade e aceitação. Em seguida, na seção subseqüente, as técnicas de teste e seus respectivos critérios serão apresentados.

Teste de Unidade

Teste de Unidade no contexto Procedimental

Consiste em projetar e criar casos de teste para os componentes do software. Esses componentes podem ser funções, procedimentos ou até mesmo "trechos" de código que executam uma determinada ação (PRESSMAN, 2006; MOLINARI, 2005). Esta estratégia tem foco na estrutura interna da implementação. Sua execução pode ser feita logo que o código-fonte tenha sido gerado. Os casos de teste podem ser projetados com base nas técnicas de teste e seus respectivos critérios.

Teste de Unidade no contexto Orientado a Objetos

No contexto Orientado a Objetos (OO) o teste de unidade se difere do contexto convencional. Uma classe ou objeto instanciado não pode ser igualado aos componentes de software definidos no teste de unidade para o contexto convencional. Isso se dá pelas características da Orientação a Objeto como encapsulamento, abstração e herança, nas quais uma operação ou método pode não estar definido isoladamente em uma única classe. Assim, essas operações ou

métodos devem ser testados nos vários contextos em que estão sendo definidos (PRESSMAN, 2006).

Teste de Aceitação

Nesse nível, os casos de teste são baseados em critérios que procuram evidenciar que o software não está atendendo aos requisitos que foram especificados.

O teste de aceitação ocorre no final do processo de desenvolvimento, tão logo se tenham funcionalidades implementadas. Os erros encontrados neste estágio decorrentes de falhas nos requisitos implicam em alto custo para sua solução, uma vez que o software está para ser entregue ao cliente. Para evitar maiores problemas, o software é liberado e esses erros são tratados posteriormente, na fase de manutenção (PRESSMAN, 2006).

3.3.1. Técnicas de Teste

São três as principais técnicas de teste e o que as diferenciam são a fonte na qual são estabelecidos os requisitos de teste:

- **Teste Funcional**, também conhecido como Teste Caixa-Preta, estabelece os requisitos de teste na especificação do sistema (PRESSMAN, 2006; SOMMERVILLE, 2004)
- **Teste Estrutural**, também conhecido como Teste Caixa-Branca, estabelece os requisitos de teste em uma determinada implementação, utilizando como recurso básico para a aplicação de seus critérios o grafo de fluxo de controle (PRESSMAN, 2006; SOMMERVILLE, 2004; MALDONADO et al., 1998)
- **Teste Baseado em Erros,** estabelece os requisitos de teste também no código, no qual são explorados os erros típicos cometidos pela equipe de desenvolvedores (PRESSMAN, 2006; MALDONADO et al., 1998).

Para cada técnica existem critérios, que compreendem um conjunto de regras de como proceder para criar e realizar, de forma eficiente, os casos de teste. Os casos de

teste devem executar as regras do software definidas pelo critério (CRESPO et al., 2004).

Como no contexto deste trabalho a técnica de teste abordada é a Funcional, essa técnica será apresentada com mais detalhes em seguida.

Teste Funcional ou Teste Caixa-Preta

Nessa técnica, os casos de teste são gerados com base nos requisitos levantados, procurando encontrar erros nas funcionalidades que o software está executando. Por essa técnica estar voltada para a funcionalidade, utilizando somente as entradas e saídas do software, e não a forma como o software foi implementado, ela é chamada de teste funcional ou caixa-preta. O termo caixa-preta provém do fato que o "conteúdo" do software não é relevante para essa técnica, ou seja, é "desconhecido" (PRESSMAN, 2006; SOMMERVILLE, 2004). Dessa forma, essa técnica procura por erros de interface, comportamento e desempenho, funções incorretas ou omitidas, de acesso à base de dados e de inicialização e término (PRESSMAN, 2006).

Exemplos de critérios dessa técnica: *Particionamento por classes de equivalência*; *Análise de valor-limite*; *Análise causa-efeito* e "*Error Guessing*". Esses critérios são apresentados a seguir.

Particionamento por classes de equivalência

Esse critério utiliza o domínio de entradas do software o dividindo em classes, conforme os requisitos, equivalentes (PRESSMAN, 2006; SOMMERVILLE, 2004). Os casos de teste são gerados a partir das classes de equivalência identificadas. As classes de equivalência podem ser identificadas através das seguintes diretrizes (MYERS, 2004; MOLINARI, 2005):

- Quando uma condição de entrada especifica um intervalo são identificadas três classes de equivalência, uma classe de equivalência válida (entrada válida) e duas não válidas (entradas não válidas), das quais, uma refere-se a um valor não válido abaixo do intervalo e a outra se refere a um valor não válido acima do intervalo.

- Quando uma condição de entrada especifica uma quantidade de valores válidos são identificadas três classes de equivalência, uma classe de equivalência válida e duas não válidas, das quais, uma refere-se à quantidade igual a zero e a outra se refere a uma quantidade a baixo ou acima do especificado.

- Quando uma condição de entrada especifica um conjunto de valores válidos é identificada uma classe válida para cada elemento do conjunto e uma classe inválida que se refere a um elemento que não pertence ao conjunto.

- Quando uma condição de entrada contém uma situação "deve ser" (restrição) são identificadas duas classes de equivalência, uma classe de equivalência válida que atenda a restrição e uma inválida que não atenda a restrição.

A utilização do critério de Particionamento de Equivalência é caracterizada por gerar casos de teste unicamente através dos requisitos do software. Essa característica faz com que se tenha melhor resultado em processos de desenvolvimento quando as variáveis podem ser identificadas facilmente e seus valores sejam bem definidos. A forma como o critério de particionamento de equivalência trabalha com as entradas descritas na especificação do software, faz com que o domínio de entrada se torne limitado. Com base nas classes de equivalência, os casos de teste são gerados. Os casos de teste devem cobrir as classes válidas e inválidas identificadas.

A seguir é mostrado um exemplo de utilização do critério de particionamento por classes de equivalência.

Considere o requisito:

- Deve ser atribuída uma nota entre 0 e 10 ao aluno.

A condição de entrada denota um intervalo, ou seja, são identificadas três classes de equivalência, uma classe válida e duas classes inválidas.

Classe válida: atribuir nota 5.

Classes inválidas: atribuir nota -5; atribuir nota 15.

Estando as classes de equivalências identificadas, os casos de teste são gerados de forma que cubram as classes de equivalência válidas e inválidas.

Análise do Valor Limite

O critério de Análise do Valor Limite é derivado do critério de Particionamento de Equivalência (PRESSMAN, 2006). Esse critério testa os limites da classe de equivalência, sendo os dados de entrada extraídos dos extremos de uma classe de equivalência, e não qualquer elemento que represente a classe de equivalência (MYERS, 2004), conforme mostrado na Figura 3.1.

Para identificar as situações de teste, as diretivas a seguir podem ser utilizadas (MYERS, 2004):

- Quando a condição de entrada especificar um intervalo, são gerados os casos de teste que tratam o início e o fim do intervalo (extremos) e também os casos de teste que tratam os valores inválidos imediatamente após as extremidades do intervalo.

- Quando a condição de entrada especificar um dado número de valores, são gerados casos de teste que tratam o valor mínimo e máximo e casos de teste tratam valores inválidos imediatamente após o mínimo e após o valor máximo.

- Quando a entrada ou saída de um programa especificar um conjunto ordenado, os casos de teste devem tratar o primeiro e o último elemento do conjunto.

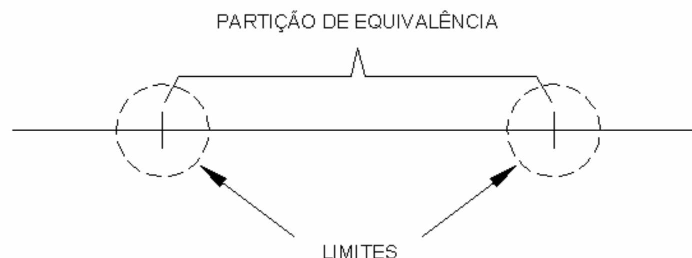

Figura 3. 1 - Limites de uma partição de equivalência (FABBRI, 2005).

A seguir é mostrado um exemplo do critério análise do valor limite:

Considera-se o requisito:

- Deve ser atribuída uma nota entre 0 e 10 ao aluno.

Como o critério análise do valor limite é derivado do particionamento por classes de equivalência, inicialmente verifica-se a condição de entrada corresponde a um intervalo, sendo assim, as classes de equivalência são geradas considerando-se os limites do intervalo, e não qualquer elemento que represente a classe de equivalência. Dessa forma são identificadas as seguintes classes:

Classes válidas: atribuir nota 0; atribuir nota 10.

Classes inválidas: atribuir nota -1; atribuir nota 11.

Análise Causa-Efeito (MYERS, 2004)

Esse critério consiste em identificar causas e efeitos na especificação do programa e, a partir disso, gerar casos de teste. A "causa" corresponde a uma condição de entrada ou a uma classe de equivalência de uma condição de entrada, enquanto o "efeito" é uma resposta (ação) às condições de entrada. Esse critério permite testar combinações entre as condições de entrada. É utilizado o grafo causa-efeito para representar os dados de entrada, com suas possíveis combinações, e as respectivas saídas. A partir do grafo causa-efeito é gerada a tabela de decisão, em que cada coluna, também chamada de regra, representa um caso de teste. Para utilizar esse critério é necessário que a especificação, da qual são extraídos as causas e efeitos, seja dividida em partes menores porque o controle do grafo causa-efeito se torna difícil em grandes especificações.

A notação básica para a construção do gráfico compreende os operadores: *identidade*, *e*, *ou*, *negação*. A Figura 3.2 mostra exemplos da notação desses operadores.

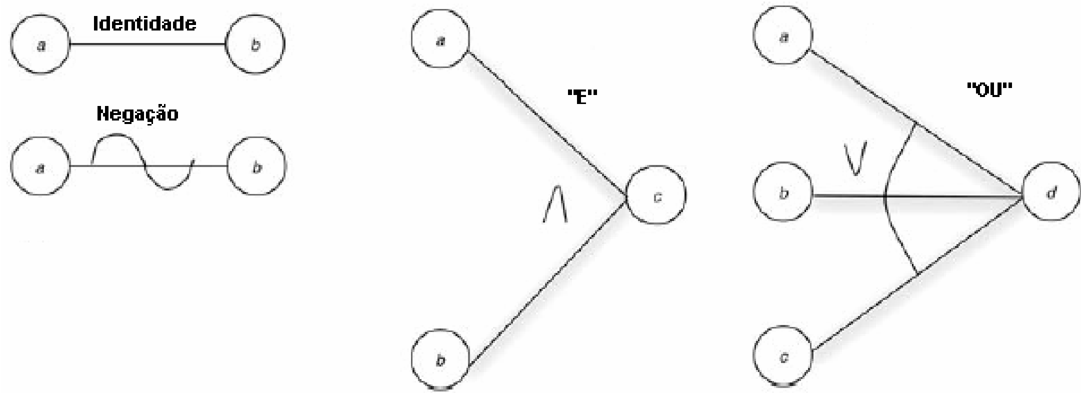

Figura 3. 2 - Notação para construção do grafo (adaptado de MYERS, 2004).

Para representar no grafo combinações de causas que são impossíveis de ocorrer, também é utilizada uma notação. A Figura 3.3 mostra essa notação:

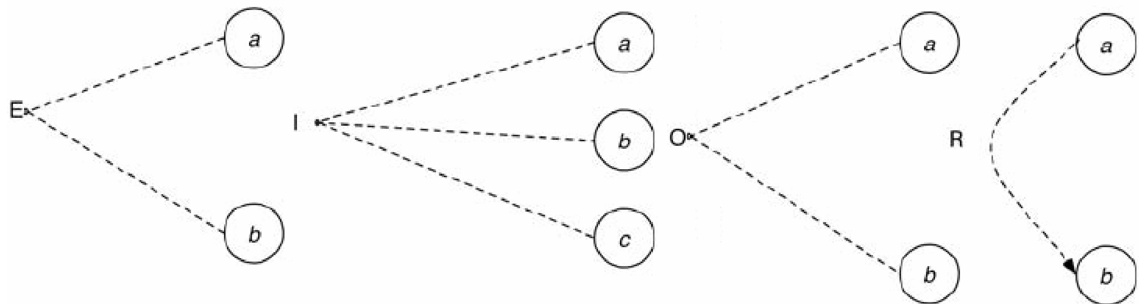

Figura 3. 3 - Notação de Restrições (MEYERS, 2004).

As notações, mostradas na Figura 3.3, correspondem as restrições:

E: Os estados *a* e *b* não poderão ser verdadeiros simultaneamente.

I: Os estados *a*, *b* e *c* não poderão ser falsos simultaneamente.

O: Somente um dos estados pode ser verdadeiro.

R: Para que o estado *a* seja verdadeiro, *b* também deve ser verdadeiro. O estado *a* não pode ser verdadeiro se o estado *b* for falso.

A seguir é mostrado um exemplo de utilização do critério análise causa-efeito:

Consideram-se os seguintes requisitos:

- Se o cliente é classificado com risco "E" e o valor do pedido for igual ou inferior a 1000, então permitir a emissão do pedido de venda.

- Se o cliente é classificado com risco "E" e o valor do pedido for superior a 1000, então não permitir a emissão do pedido de venda.

- Se o cliente é classificado como tipo "A", permitir a emissão do pedido de venda.

Causas:

C1 - O tipo é "E".

C2 - O tipo é "A".

C3 - O valor é menor que ou igual a 1000.

C4 - O valor é superior a 1000.

Efeitos:

E1 - Permitir a emissão do pedido de venda.

E2 - Não permitir a emissão do pedido de venda.

A Figura 3.4 mostra o grafo elaborado.

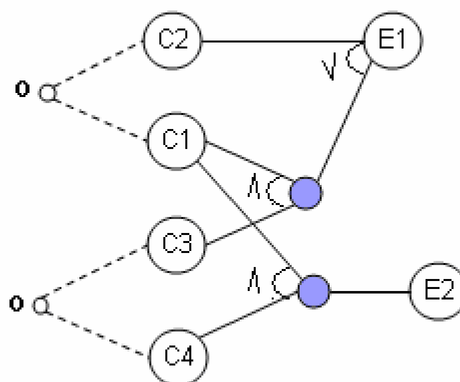

Figura 3. 4 - Grafo causa-efeito completo.

Após a elaboração do grafo causa-efeito, a Tabela 3.1 representa a tabela de decisão obtida.

Tabela 3. 1 - Tabela de decisão exemplo.

Item	R1	R2	R3	R4
C1	**F**	**F**	**V**	**V**
C2	**V**	**V**	**F**	**F**
C3	**V**	**F**	**V**	**F**
C4	**F**	**V**	**F**	**V**
E1	√	√	√	x
E2	x	x	x	√

Cada coluna da Tabela 3.1 correspondente a uma regra (R) gera um caso de teste.

Error Guessing

Esse critério consiste em identificar e listar possíveis erros, ou situações que tendem a erros, em uma dada especificação, supondo alguns tipos prováveis de erro. Com base nisso, são extraídos os casos de teste. Essa lista é criada sem a utilização de diretrizes, sendo assim totalmente intuitiva, ou seja, é um processo ad-hoc. Dessa forma, esse critério está relacionado à experiência do testador.

3.4. Inspeção

O processo de inspeção é composto por atividades de verificação estática, as quais revisam os artefatos gerados durante o processo de desenvolvimento com o objetivo de encontrar e corrigir possíveis defeitos antes do software ter sido implementado (SOMMERVILLE, 2004).

Laitenberger (2001) estabelece papéis dentro do processo de inspeção. Esses papéis são relacionados na Tabela 3.2.

Tabela 3. 2 - Papéis do processo de inspeção (LAITENBERGER, 2001).

Papel	*Responsabilidade*
1. ***Organizador***	Responsável pelo planejamento das atividades do processo de inspeção.
2. ***Moderador***	É o responsável pela equipe. O moderador garante o comprometimento dos integrantes da equipe com suas respectivas responsabilidades e a execução das atividades previamente estabelecidas.
3. ***Inspetor***	Responsável por realizar a inspeção propriamente dita. Procura defeitos, inconsistências, omissões e ambigüidades nos artefatos a serem inspecionados.
4. ***Autor***	Criador e responsável pelo artefato a ser inspecionado.
5. ***Leitor***	Apresenta à equipe os artefatos a inspecionar, informando e explicando o objetivo esperado de cada artefato.
6. ***Coletor***	Registra os defeitos encontrados pelas atividades de inspeção caso a reunião de inspeção não seja feita.
7. ***Redator***	Registra todos os defeitos encontrados e apresentados durante a reunião de inspeção.

Conforme a Figura 3.5, o processo de inspeção compreende quatro fases:

Planejamento: nesta fase, o organizador tem a responsabilidade de planejar e organizar todo o processo de inspeção, incluindo a definição da equipe, a responsabilidade de cada membro e os prazos para a execução das atividades.

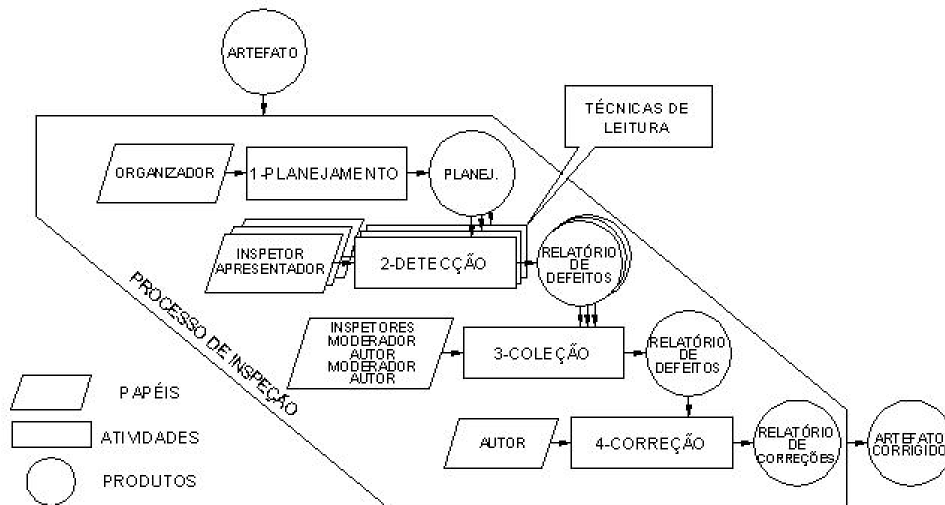

Figura 3. 5 - Processo de inspeção (adaptado de Shull, 1998).

Detecção: Nesta fase ocorre a inspeção propriamente dita. Laitenberger (2001) menciona que esta fase gera argumentações entre vários autores. O fato responsável

pelas argumentações é se a descoberta de um defeito deve ser uma atividade individual, ou é uma atividade coletiva, na qual tal atividade seria apoiada por reuniões de inspeção. Fagan (1986), citado em Laitenberger (2001), defende a realização da atividade de inspeção em grupo, pois a maioria dos defeitos é encontrada quando o artefato é inspecionado em grupo.

Coleção: Todos os problemas encontrados pelos inspetores são analisados em reuniões que ocorrem dentro desta fase. Nesta reunião é definido se o problema encontrado é realmente um defeito e se o artefato originador de tal defeito precisará ser re-inspecionado. Todos os defeitos devem ser registrados.

Correção: O autor, de posse do registro dos defeitos encontrados, executa o retrabalho, solucionando todos os defeitos de seu artefato.

O sucesso de uma inspeção sobre um artefato depende exclusivamente do inspetor a qual está submetido. Segundo Knight e Myers (1993), citado em Mafra e Travassos (2005), até o início dos anos 90 a atividade de inspeção era executada tendo como base na experiência do revisor e seu modo próprio de executar tal atividade, dessa forma, a garantia de que a atividade tenha sido realizada com qualidade era apenas a confiança no inspetor que a realizou. As técnicas de leitura procuram eliminar esse tipo de problema.

3.4.1. Técnicas de Leitura para o Processo de Inspeção

Basili, citado em Shull (1998), define técnica de leitura como uma técnica de análise individual de um artefato de software textual, que permite ao inspetor alcançar a compreensão necessária para a realização da tarefa especificada.

As técnicas de leitura envolvem três fatores importantes (SHULL, 1998):

- *Uma série de passos*: A técnica deve fornecer instruções ao inspetor de como proceder a leitura.

- *Análise individual*: O entendimento individual deve ser tratado pelas técnicas de leitura.

- *O entendimento necessário para uma tarefa em particular*: As técnicas de leitura devem garantir um nível de entendimento satisfatório para os aspectos do artefato inspecionado.

A seguir são apresentados Técnicas de Leitura Ad-hoc, CheckList e Leitura Baseada em Perspectiva (LAITENBERGER,2001).

Ad-hoc

Esta técnica não fornece uma sistemática para leitura. Os inspetores inspecionam os documentos com base em seu próprio conhecimento. Esta característica ocasiona desvantagens como:

- Dependência da experiência do inspetor, pois cada inspetor adota sua maneira de inspecionar o documento.
- Não é repetível, ou seja, a cada inspeção são adotados critérios diferentes de inspeção por depender do inspetor que esta executando a revisão.

CheckList

O checklist adota como sistemática a utilização de um formulário de questões que o inspetor deve responder à medida que executa a verificação do documento inspecionado. Esta técnica fornece ao inspetor diretrizes que o auxiliam a executar a inspeção.

O formulário de questões é elaborado com base em resultados de atividades de inspeção anteriores, nos requisitos definidos para o software e defeitos característicos do ambiente de desenvolvimento.

Leitura Baseada em Perspectiva

A técnica de Leitura Baseada em Perspectiva fornece ao revisor instruções de como inspecionar o produto baseado em diferentes aspectos. Tais aspectos são as perspectivas pelas quais o documento pode ser inspecionado, como por exemplo, a perspectiva de projetista, de um testador ou de um usuário.

À cada inspetor é entregue um "cenário", o qual consiste na criação de um modelo do documento a ser inspecionado tendo como base as características da perspectiva que se está utilizando.

Esta técnica permite atribuir a cada inspetor uma responsabilidade específica.

A Tabela 3.3 faz uma caracterização de técnicas de leituras, incluindo-se as apresentadas neste trabalho e outras as quais não foram citadas.

A caracterização foi feita sob os seguintes pontos (LAITENBERGER,2001):

- **Contexto da Aplicação**

Indica a qual artefato de software a técnica de leitura pode ser aplicada.

- **Usabilidade**

Indica se a técnica de leitura proporciona instruções precisas de como examinar o artefato de software em busca de defeitos.

- **Repetibilidade**

Indica se os resultados, bem como os defeitos detectados, são iguais independentemente da pessoa que está realizando a inspeção.

- **Adaptável**

Indica se a técnica de leitura se adapta a aspectos particulares de um determinado domínio.

- **Cobertura**

Indica se todos as propriedades de qualidade comuns ao desenvolvimento de um software, tais como exatidão e integridade, são verificados pela inspeção.

- **Treinamento**

Indica se a técnica de leitura requer treinamento.

- **Validação**

Informa como a técnica de leitura foi validada.

Os campos da tabela 3.3 identificados por "indefinidos" significam que não foi possível obter um resultado preciso e confiável (LAITENBERGER, 2001).

Tabela 3. 3 - Caracterização das técnicas de leitura (LAITENBERGER, 2001).

Técnica de Leitura	*Características*						
	Contexto de aplicação	*Usabi-lidade*	*Repeti-bilidade*	*Adaptabi-lidade*	*Cobertura*	*Treinamento Requerido*	*Validação*
Ad-hoc	Todos os produtos	Não	Não	Não	Não	Não	Prática industrial
Checklist	Todos os produtos	Não	Não	Sim	Depende do Caso	Não	Prática industrial
Leitura baseada em Perspectiva	Todos os produtos, requisitos, projeto e código.	Sim	Sim	Sim	Alto	Sim	Validação Experime ntal e uso industrial
Leitura por Abstração Gradativa	Todos os produtos que permitam abstração e código funcional	Sim	Sim	Não	Alto para defeitos específicos	Sim	Aplicado em projetos *cleanroom*
Revisão ativa de Projeto	Projeto	Sim	Sim	Não	Indefinido	Indefinido	Estudo de Caso inicial
Leitura baseada em defeitos	Todos os produtos, requisitos	Sim	Depende do Caso	Sim	Alto	Sim	Validação experimen tal
Leitura baseada em pontos de função	Todos os produtos, requisitos	Sim	Depende do Caso	Sim	Indefinido	Sim	Validação experimen tal

É possível verificar pela tabela 3.3, que a técnica de leitura Baseada em Perspectiva atende todas as características apresentadas e possui o maior contexto de aplicação. A técnica de leitura ad-hoc, embora sendo aplicada a todos os produtos, é a técnica que não atende a maioria das características apresentadas.

Na próxima seção, no contexto de desenvolvimento de software, o conceito de planejamento é explorado. Nessa seção a técnica de pontos por caso de uso é apresentada.

3.5. Planejamento

O Planejamento constitui uma atividade de preparação para o trabalho a ser executado, tanto no âmbito organizacional como no técnico. Nessa preparação, procura-se identificar os possíveis riscos e definir ações a serem tomadas caso um deles aconteça, levantar os produtos do trabalho a serem produzidos, levantar os recursos necessários disponíveis e os não disponíveis a serem utilizados durante a execução do trabalho, elaborar um cronograma contendo todas as atividades a serem realizadas (PRESSMAN, 2006). O planejamento fornece diretrizes a serem seguidas para se atingir as metas e alcançar os objetivos esperados.

Uma prática simples que fornece uma visão macro do planejamento, independentemente do porte e segmento do projeto, é o 5W2H de Bohem (1996), citado por Pressman (2006). Essa prática consiste em obter respostas a sete questões que são fundamentais a qualquer planejamento. São elas: "*O quê?*"; "*Por quê?*"; "*Quando?*"; "*Quem?*"; "*Onde?*"; "*Como?*" e "*Quanto?*". A sigla 5W2H provém das iniciais dessas perguntas na língua inglesa.

Pressman (2006) descreve um conjunto de tarefas, genérico, para o planejamento do processo de desenvolvimento de software. Essas tarefas são:

Reavaliar o escopo do projeto: é necessário identificar o contexto em que o software a ser construído irá atuar, assim como os produtos de saída que esse software fornecerá ao cliente, os quais devem satisfazer as suas expectativas. Os dados de entrada para gerar os produtos de saída e também as funções que serão desempenhadas pelo software, que transformaram dos dados de entrada em produtos de saída, também devem ser identificadas.

Avaliar os riscos: os riscos os quais o projeto está sujeito devem ser verificados, assim como as conseqüências que serão ocasionadas caso um desses riscos ocorra.

Desenvolver e/ou refinar cenários de usuário: os cenários, que contextualizam as funcionalidades do software, frequentemente visualizados através dos casos de uso, devem ser criados e refinados.

Definir funções e características técnicas que formem a infra-estrutura do software: A modelagem da análise, demonstrando ações e decisões decorrentes de funções ou atores, com base nos cenários, deve ser elaborada.

Agrupar funções e características por prioridade do cliente;

Criar um plano de projeto geral: Um cronograma geral é elaborado e o número de incrementos do software é definido juntamente com as respectivas datas de entrega.

Criar um plano de refinado para a iteração atual: Nessa tarefa, as atividades específicas para a construção do incremento são definidas, tais como o esforço necessário para a iteração, as responsabilidades e os produtos de trabalhos a serem gerados.

Acompanhar o progresso regularmente.

Com relação ao plano de projeto, esse é detalhado em um conjunto de tarefas, também estabelecido por Pressman (2006). Essas tarefas são:

- Estabelecer o escopo do projeto
- Determinar a viabilidade
- Analisar os riscos
- Definir recursos necessários
- Determinar recursos humanos necessários
- Definir recursos reusáveis de software
- Identificar recursos ambientais
- Estimar custo e esforço
- Decomponha o problema
- Desenvolva duas ou mais estimativas usando tamanho, pontos por função, tarefas de processo, ou casos de uso

- Harmonize as estimativas
- Desenvolva um cronograma.

Neste trabalho, no contexto de planejamento de projeto, mais especificamente na tarefa de estimar custo e esforço, será abordada uma técnica de estimativa de esforço baseada no diagrama de casos de uso, chamada técnica de pontos por caso de uso (KARNER, 1993).

3.5.1. Técnica de Pontos de Casos de Uso

Definida por Gustav Karner (1993), essa técnica consiste em estimar projetos orientados a objetos com base no modelo de casos de uso. As etapas a serem realizadas são descritas a seguir:

Contagem dos Atores: o Peso total dos Atores Não Ajustado (*UAW*) corresponde ao somatório do produto de cada Tipo de Ator pelo respectivo Peso (ator simples (*As*) com peso 1; ator médio (*Am*) com peso 2 e ator complexo (*Ac*) com peso 3), de acordo com a fórmula (1).

$$UAW = Aws * \sum As + Awm * \sum Am + Awc * \sum Ac \qquad (1)$$

Na qual, *Aws*, *Awm* e *Awc* correspondem ao peso do *As*, peso do *Am* e peso do *Ac*, respectivamente.

A Tabela 3.4 mostra os tipos de atores as regras que os definem.

Tabela 3. 4 - Peso dos atores (KARNER, 1993).

Tipo	Representação	Peso
Simples	Outro sistema interagindo através de uma API.	1
Médio	Outro sistema interagindo remotamente através de protocolos de comunicação	2
Complexo	Usuário interagindo através de uma interface gráfica	3

Contagem dos Casos de Uso: o Peso total dos Casos de Uso Não Ajustado (*UUCW*) é calculado de acordo com a complexidade envolvida, conforme as seguintes regras:

- *UCs*: um Caso de Uso simples contém uma entidade de banco de

dados, ou contém quatro ou menos cenários principais ou caminhos de execução, ou envolve quatro ou menos objetos de análise. Nesse caso, pode-se considerar um peso simples (*UCWs*) igual a cinco, para o caso de uso.

- *UCm*: um Caso de Uso médio contém duas ou mais entidades de banco de dados, ou contém de cinco a oito cenários principais ou caminhos de execução, ou envolve de cinco a dez objetos de análise. Nesse caso, pode-se considerar um peso médio (*UCWm*) igual a dez, para o caso de uso.
- *UCc*: um Caso de Uso complexo contém três ou mais entidades de banco de dados, ou contém oito ou mais cenários principais ou caminhos de execução, ou envolve mais que dez objetos de análise, então, nesse caso, pode-se considerar um peso complexo (*UCWc*) igual a quinze, para o caso de uso.

A contagem dos Casos de Uso é obtida pela fórmula (2):

$$UUCW = UCWs * \sum Ucs + UCWm * \sum Ucm + UCWc * \sum Ucc \qquad (2)$$

O cálculo do Peso Total Não Ajustado (*UUCP*) corresponde ao somatório entre os pesos de Atores e Casos de Uso calculados pela fórmula (3):

$$UUCP = UAW + UUCW \qquad (3)$$

O cálculo do Fator de Ajuste possui duas etapas: determinar o Fator de Complexibilidade Técnica (*TCF*) e determinar o Fator de Ambiente (*EF*). Estes dois tipos de fatores geram multiplicadores distintos e devem ser aplicados ao *UUCP*.

O *TCF* é obtido a partir da relação de características técnicas (Tabela 3.5). São atribuídos os níveis de influência de cada característica no projeto. A escala do nível de

influência varia de zero a cinco, sendo que zero significa que a característica não exerce influência no projeto e cinco significa que a característica tem grande influência no projeto. Dados os níveis de influência a cada característica técnica, é obtido o fator técnico (*Tfactor*) que corresponde à somatória dos produtos obtidos entre o peso da característica e o nível de influência dado a essa característica (fórmula 4).

$$Tfactor = \sum_{i=1}^{13} peso_i * \ (\textit{nível de influência}) \qquad (4)$$

O *TCF* é obtido pela fórmula (5):

$$TCF = 0{,}6 + (0{,}01 * Tfactor) \qquad (5)$$

O *EF* é calculado utilizando uma relação de recursos (Tabela 3.6), na qual é atribuído o nível de disponibilidade de cada recurso no projeto. A escala do nível de disponibilidade varia de zero a cinco, sendo que zero significa que o recurso não está presente no projeto e cinco significa que o recurso está totalmente presente no projeto. Dados os níveis de disponibilidade a cada recurso, o fator de ambiente (*Efactor*) é calculado. Esse fator corresponde a somatória dos produtos obtidos entre o peso do recurso e o nível de disponibilidade dado a essa recurso (fórmula 6):

$$Efactor = \sum_{i=1}^{8} peso_i * \ (\textit{nível de disponibilidade}) \qquad (6)$$

O *EF* é obtido pela fórmula (7):

$$EF = 1{,}4 + (-0{,}03 * Efactor) \qquad (7)$$

Tabela 3. 5 - Características técnicas (KARNER, 1993).

Característica	Peso	Nível de influência	Resultado
Sistema distribuído	2	X_1	Y_1
Tempo de resposta	2	X_2	Y_2
Eficiência	1	X_3	Y_3
Processamento complexo	1	X_4	Y_4
Código reusável	1	X_5	Y_5
Facilidade de instalação	0,5	X_6	Y_6
Facilidade de uso	0,5	X_7	Y_7
Portabilidade	2	X_8	Y_8
Facilidade de mudança	1	X_9	Y_9
Concorrência	1	X_{10}	Y_{10}
Recursos de segurança	1	X_{11}	Y_{11}
Acessível por terceiros	1	X_{12}	Y_{12}
Requer treinamento especial	1	X_{13}	Y_{13}
Fator técnico (Tfactor)			$\sum_{i=1}^{13} Yi$

Tabela 3. 6 - Recursos disponíveis (KARNER, 1993).

Recurso	Peso	Nível de disponibilidade	Resultado
Familiaridade com algum processo formal	1,5	X_1	Y_1
Experiência com a aplicação em desenvolvimento	0,5	X_2	Y_2
Experiência com Orientação a Objetos	1	X_3	Y_3
Presença de analista experiente	0,5	X_4	Y_4
Motiviação	1	X_5	Y_5
Requisitos estáveis	2	X_6	Y_6
Desenvolvedores em meio expediente	-1	X_7	Y_7
Linguagem de programação difícil	2	X_8	Y_8
Fator ambiental (Efactor)			$\sum_{i=1}^{8} Yi$

Nas Tabelas 3.5 e 3.6, X_i corresponde ao nível de influência e ao nível de disponibilidade respectivamente, e Y_i corresponde ao produto entre o peso o nível aplicado.

Para se determinar a estimativa de tempo, inicialmente calcula-se o valor de Pontos por Caso de Uso Ajustado (*AUCP*). Esse valor é obtido através do *UUCP*, do *TCF* e do *EF*. O produto entre esses três parâmetros determina o valor de pontos por

funcionalidade ajustado. Dessa forma, a valor de pontos por funcionalidade ajustado é o valor resultante da seguinte fórmula (8):

$$AUCP = UUCP * TCF * EF \qquad (8)$$

Encontrado o $AUCP$, aplica-se a ele a quantidade de horas-homem que a equipe de desenvolvimento consome para implementar um ponto por caso de uso. Karner (1993) propôs um fator de 20 homens-hora por cada ponto de casos de uso. Como exemplo, se o $AUCP$ corresponde a 10, a estimativa de tempo seria de 200 horas.

3.6. Considerações Finais

Neste capítulo foram abordadas atividades de garantia da QS que são contempladas na abordagem proposta neste trabalho. Foram apresentados os principais conceitos sobre as atividades de teste e inspeção e, referente ao planejamento, foi apresentada a técnica de pontos por caso de uso, utilizada para estimar o esforço necessário em projetos de software. Como citado neste capítulo, essas atividades são fundamentais para o sucesso de um software. A ocorrência de não-conformidades e erros durante o desenvolvimento pode implicar em vários problemas, tais como erro no custo previsto para o desenvolvimento, atraso no cronograma e credibilidade.

Como este trabalho está concentrado no paradigma de desenvolvimento ágil e como as atividades de teste e inspeção são tratadas na abordagem aqui proposta, no próximo capítulo comenta-se como as atividades de teste e inspeção têm sido abordadas no contexto ágil.

CAPÍTULO 4

TESTE E INSPEÇÃO NO CONTEXTO ÁGIL

4.1. Considerações Iniciais

Como apresentado no Capítulo 2, não são detalhadas explicitamente diretrizes de para utilização dos MA. Esse fato envolve também as atividades de teste e inspeção e, portanto, questões referentes ao “Como fazer?”, “Quando fazer?” e “Com o quê fazer?” ainda não estão claras, embora as atividades de teste, principalmente, sejam consideradas um dos alicerces desse paradigma de desenvolvimento.

Para contextualizar as atividades de teste e inspeção nesse paradigma, este capítulo apresenta os principais conceitos encontrados na literatura acerca dessas atividades.

Este capítulo está organizado da seguinte forma: na Seção 4.2 são abordados os princípios da atividade de teste no contexto ágil; Na Seção 4.3, são mostrados exemplos de automação de testes de unidade e de aceitação; Na Seção 4.4, a utilização do testador no contexto ágil é discutida; Na Seção 4.5, a inspeção no contexto ágil é abordada e na Seção 4.6, se encontram as considerações finais deste capítulo.

4.2. A Atividade de Teste no contexto Ágil.

Como já foi mencionado no Capítulo 2, no contexto ágil enfatiza-se o contato direto e constante da equipe de desenvolvimento com o cliente, agregando maior valor ao projeto do que à elaboração formal de planos em geral, inclusive planos de teste e relatórios de falhas (MARICK, 2001; PETTICHORD, 2002). Segundo Marick (2001), nos MA a atividade de teste possui quatro características: i) estar voltada às regras de negócio, para focar os interesses do cliente; ii) contemplar a tecnologia utilizada; iii) contemplar a criticidade do produto; e iv) dar apoio à programação, uma vez que as atividades de teste passam a fazer parte do processo de desenvolvimento, permitindo que os desenvolvedores as utilizem para orientar a programação.

Ambler (2004) também menciona a importância das atividades de teste ressaltando duas práticas: i) considere a testabilidade, dizendo que antes e durante o desenvolvimento é necessário prever como o software será testado e que a execução freqüente das atividades de teste garante que o software está sendo produzido corretamente; e ii) comprove o código, sugerindo que se use o código para validar as tarefas que estão e serão realizadas, no sentido de que, ao se desenvolver uma rotina, deve-se apresentá-la ao cliente com o intuito de ter o seu parecer.

Através do estudo realizado, conclui-se que as atividades de teste dentro do contexto ágil estão relacionadas aos testes de unidade, através da estratégia do Desenvolvimento Orientado por Testes (DOT), e aos testes de aceitação, devido a participação do cliente durante o projeto. Verificou-se também que não se encontrou nada definido explicitamente sobre as atividades de teste no contexto dos MA embora o teste seja um dos grandes alicerces desse movimento. A única exceção foi relacionada ao método XP, que enfatiza essas atividades, fornecendo um pouco mais de detalhes para as atividades de teste de unidade e de aceitação, sendo o teste de unidade realizado através do desenvolvimento orientado por testes e o teste de aceitação tendo seus casos de teste escritos diretamente pelo cliente. Na próxima Seção esses dois tipos de teste são abordados.

4.2.1. Teste de Unidade

O teste de unidade no contexto ágil adota como prática o Desenvolvimento Orientado por Testes (DOT). Essa estratégia é uma das práticas adotadas pelo XP (PANCUR et. al, 2003). O DOT considera que os testes de unidade devem ser automatizados, e que a criação dos casos de teste deve acontecer antes da implementação da funcionalidade, para que eles guiem a implementação, de forma que, ao ser submetida ao caso de teste, a implementação passe por ele produzindo o resultado esperado. Esse princípio faz com que o desenvolvedor execute o caso de teste de uma funcionalidade mesmo sem sua implementação, provocando, dessa forma, um resultado não satisfatório, que reflete o que ele deve produzir para que o comportamento do que será implementado seja o comportamento esperado (BECK, 2004; TELES, 2004; CRISPIN, 2006; ERDOGMUS et. al., 2005; BOHNET, MESZAROS, 2005; JANERT, 2004; JANZEN, SAIEDIAN, 2006; JANZEN, SAIEDIAN, 2005; PANCUR et.al 2003; MARICK, 2003; STEINDL, 2004).

Pancur et al. (2003) denominam “ritmo” do DOT a execução das seguintes etapas:

- Escrever rapidamente o caso de teste para a funcionalidade;
- Executar o caso de teste e verificar que a funcionalidade testada não foi bem sucedida (ela ainda não foi implementada);
- Implementar a funcionalidade;
- Executar o caso de teste até que a funcionalidade seja validada.
- Eliminar código redundante.
- Iniciar um novo ciclo (caso de teste).

Make (2001) faz uma analogia entre essas etapas e um semáforo. Segundo ele, a funcionalidade validada pelo caso de teste corresponde à luz verde, indicando que o caso de teste não encontrou falhas na unidade testada. A luz amarela indica que o caso de teste não pode ser executado, e a luz vermelha indica que o caso de teste foi executado porém falhas foram encontradas na unidade testada.

Steindl (2004), Pettichord (2002) e Beck (2001) definem as seguintes motivações e contribuições para a adoção do DOT:

- Se planejar a realização das atividades de teste ao fim da implementação elas não serão feitas.
- A utilização do DOT faz com que não seja possível executar um código sem que tenha passado pelos testes.
- Encoraja e ajuda a refletir sobre o escopo da implementação.
- Ajuda o entendimento da especificação do sistema (análise e projeto).
- Promove maior segurança de que o funcionamento do sistema está de acordo com a especificação. Como os testes são automatizados, eles podem ser executados sempre que há alteração no código do projeto. O código é escrito tendo como objetivo principal atender o caso de teste, para que ele seja bem sucedido.

- Contribui para a simplicidade do projeto, pois o código é escrito somente para que atenda o caso de teste.

A eficiência do DOT é relatada por Janzen e Saiedian (2006), com base em pesquisas realizadas, tanto no meio acadêmico como na indústria. Essa estratégia estabelece uma relação positiva com a produtividade do programador, tendo maior aceitação após a sua primeira aplicação, (JANZEN, SAIEDIAN, 2006). Crispin (2006) comenta que à medida que se adquire experiência na utilização dessa estratégia, maior é a eficácia obtida. Além disso, ela auxilia na compreensão do domínio, que é um componente chave no processo de desenvolvimento do software.

Percebeu-se durante o estudo que trabalhos estão sendo elaborados para que o DOT possa ser aplicado em variados contextos. Como exemplo desses trabalhos, têm-se o *Presenter First* (ALLES et al., 2006), uma técnica para organização de código para que o DOT possa ser utilizado em aplicações com interfaces complexas. Outro trabalho propõe o *framework* para testes, voltado para processos nos quais as regras de negócio são complexas (SAURER et. al, 2006).

O DOT é executado pelo próprio programador através da criação dos testes de unidade para cada classe a ser desenvolvida, mais particularmente para os métodos das classes. Esses testes não são executados no fim, mas durante toda a fase de desenvolvimento e têm como objetivo verificar se os métodos de cada classe retornam o valor esperado (YNCHAUSTI, 2001). Na Seção 4.3.1 é apresentado um exemplo que ilustra a aplicação do DOT utilizando-se a ferramenta JUnit.

Os testes de unidade devem ser o mais simples possível, de forma a retornar um único valor indicando se o teste falhou (o comportamento do método não foi o esperado) ou passou (o comportamento do método foi o esperado). Eles devem ser mantidos e atualizados juntamente com o sistema, podendo ser utilizados toda vez que o sistema for modificado (YNCHAUSTI, 2001).

4.2.2. Teste de Aceitação

Como citado anteriormente, dos MA estudados, somente o método XP descreve com alguns detalhes os testes de aceitação. Os testes de aceitação são realizados com base nos requisitos funcionais do sistema, ou seja, as regras de negócio. Os testes de

aceitação procuram verificar se o comportamento do software está correto, conforme os requisitos definidos nas histórias de usuário. Essa verificação é feita a cada incremento do ciclo de vida.

Os testes de aceitação são escritos pelo cliente, que também é encarregado de fornecer os dados de teste, podendo ser auxiliado por integrantes da equipe. O XP enfatiza que pessoas responsáveis pelas regras de negócio tomam decisões sobre as regras de negócio e pessoas responsáveis pela parte técnica tomam decisões técnicas (MARTIN et. al, 2004).

Os testes de aceitação devem ser escritos antes da implementação da funcionalidade (TELES, 2004) e sua execução deve ser realizada no decorrer de cada iteração auxilia no entendimento do processo. Quando executados durante as iterações, os testes auxiliam a entender a funcionalidade e, quando executados ao fim, verificam se a funcionalidade foi bem construída (STEINDL, 2004; PETTICHORD, 2002).

Os testes de aceitação representam os interesses do cliente, dando a ele confiança na aplicação e mostrando que ela contém as funcionalidades especificadas e que elas são executadas corretamente. Esses testes são responsáveis por capturarem as exigências do usuário e verificarem se a aplicação atende a essas exigências, expondo problemas que os testes de unidade não identificam, tais como usabilidade, praticidade e validações de interface. Esses testes fornecem uma visão de como é o sistema através da execução e validação das funcionalidades (MILLER, COLLINS, 2001).

4.3. Automação dos Testes de Unidade e de Aceitação

Segundo Myers (2004), por menor que seja o sistema a ser desenvolvido, é grande a quantidade de testes de unidade a serem elaborados e realizados, tornando indispensável o uso de ferramentas para a automação destes testes. Steindl (2004) cita que não é possível utilizar uma funcionalidade caso ela não tenha sido submetida a testes.

A automação dos casos de testes permite verificar, a qualquer momento, o comportamento atual do sistema (BECK, 2004). Dessa forma, a automação dos testes permite verificar se cada alteração ou incremento de novas funcionalidades não estão apresentando problemas ou ocasionando problemas no sistema como um todo.

Dos MA estudados, o método XP é o que mais enfatiza a automação dos testes. Para a automação de testes de unidade, o *JUnit* foi a ferramenta mais evidenciada nos estudos realizados e citada como referência para a automação dos casos de teste, porém é aplicada somente em projetos em que a linguagem de implementação adotada seja *Java*. Para a automação de testes de aceitação, uma ferramenta verificada, menos evidenciada que o *JUnit* e voltada somente para aplicações Web, é o *Selenium*. A seguir, são exemplificados a automação de casos de teste de unidade e de aceitação, utilizando-se as ferramentas *JUnit* e *Selenium* respectivamente.

4.3.1. Automação de Testes de Unidade com o JUnit

A ferramenta *JUnit* foi desenvolvida por Erich Gamma e Kent Beck, e implementa os testes de unidade em *Java*, estruturando-os e executando-os automaticamente (JUNIT).

No *JUnit*, cria-se uma classe de teste para a classe que será implementada. A classe de teste criada estende a classe *TestCase* do *JUnit*, sendo que cada teste a ser feito é um método desta classe de teste. A classe *TestCase* provê uma série de métodos que permitem e auxiliam confrontar o resultado obtido com o resultado esperado. Esses métodos permitem identificar se o teste falhou ou não, ou houve uma exceção em sua execução.

O exemplo a seguir foi adaptado do exemplo contido em Myers (2004). Para este exemplo foram utilizados:

- IDE de código aberto Eclipse 3.2
- Framework JUnit 3.8.1
- JRE System Library [jre1.5.0_03]
- Compilador Java 5.0

Especificação: Desenvolver uma aplicação em *Java*, tal que fornecido um número inteiro positivo n, com $1 \leq n \leq 100$, determine se n é primo ou não.

A prática do DOT determina que devemos escrever os casos de teste antes de codificar a funcionalidade (MYERS, 2004).

Tendo como base a especificação dada e utilizando o teste caixa-preta com o critério análise do valor limite, especificam-se os seguintes casos de teste mostrados na Tabela 4.1.

Tabela 4. 1 - Casos de teste conforme a especificação (adaptado de MYERS, 2004).

Caso de Teste	Entrada	Resultado esperado	Comentário
1	3	Informar que n é primo	Teste tendo como entrada um número primo e dentro dos limites especificados
2	100	Informar que n não é primo	Teste tendo como entrada um número não primo e igual ao limite superior especificado
3	1	Informar que n não é primo	Teste tendo como entrada um número não primo e igual ao limite inferior especificado
4	101	Informar que a entrada n não é válida	Teste tendo como entrada um número acima do limite superior especificado
5	-1	Informar que a entrada n não é válida	Teste tendo como entrada um número negativo e abaixo do limite inferior especificado
6	“a”	Informar que a entrada não é valida	Teste tendo como entrada um dado não numérico
7	Vazio (“ “)	Informar que a entrada não é válida	Teste para entrada nula.
8	3.1	Informar que a entrada não é válida	Tese tendo como entrada um número não inteiro

Com a especificação dos casos de teste, primeiramente criam-se as classes de teste. Para criar a aplicação é necessário criar uma classe denominada *Primo*, que seja responsável por determinar se a entrada informada é um número primo ou não.

O DOT estabelece que, antes de criar a classe *Primo*, é necessário escrever os testes para essa classe. Assim a primeira classe de teste criada é denominada *TestePrimo*, que será responsável pelos testes da classe *Primo*. Cada método da classe *TestePrimo* será um teste a ser executado. Tendo como base o caso de teste número 1 da Tabela 4.1, temos a classe de teste:

```
public class TestePrimo extends TestCase {
      public void CasoTeste1(){
            Primo primo = new Primo(3);
            assertEquals(1,primo.Verifica());
      }
}
```

O método *assertEquals* é proveniente da classe *Assert* do *JUnit*. Esse método verifica a igualdade entre seus argumentos. A classe *Assert* possui um conjunto de métodos de afirmação, úteis para o desenvolvimento dos testes.

Ao executar o teste, através do *JUnit*, o mesmo falha pois indica que a classe *Primo* e o método *Verifica* não existem. O ato de executar o teste mesmo não tendo a funcionalidade implementada é denominado como "*expressar a intenção no código*" (TELES, 2004). Isso faz com que os programadores reflitam primeiramente no resultado a ser obtido pela funcionalidade e não em como implementar a funcionalidade.

Para cada teste criado, primeiramente é preciso mostrar que o mesmo apresenta falha, para que, implementando-se a funcionalidade e executando-se novamente o teste, seja possível verificar se o retorno é o resultado esperado. Cria-se então a classe *Primo* e a codificação do método *Verifica* para atender o teste executado.

Ao executar o teste referente ao caso de teste número 4 da Tabela 4.1, percebe-se que a informação de entrada não está sendo validada, pois a aplicação não poderia aceitar números que não estejam no intervalo definido na especificação, sendo necessário a criação de uma classe para a validação da informação de entrada.

Dessa forma, à medida que são criados os testes, as funcionalidades são implementadas.

Para a elaboração das classes de teste da classe de validação, denominada *Valida*, a mesma Tabela 4.1 pode ser aplicada. O código-fonte das classes de teste e das classes de negócio da aplicação está listado no Apêndice A.

O *JUnit* permite executar cada classe de teste individualmente ou executar todas as classes automaticamente. Para executar todas as classes automaticamente é necessário criar uma classe que defina o método *public static Test suíte()*, adicionando a ele todas as classes de teste do projeto.

Ao executar uma classe de teste, todos os seus métodos são executados e cada método representa um teste específico.

Os métodos *AssertEquals*, *AssertNotSame*, *AssertTrue* e *AssertFalse* são utilizados para confrontar o resultado esperado com o resultado obtido. Qualquer divergência apurada por esses métodos faz com que o teste em execução seja falho.

O *JUnit* representa graficamente o resultado dos testes. Quando todos os casos de teste são bem sucedidos, a barra de status, mostrada na Figura 4.1, fica na cor verde e todos os casos de teste recebem uma marcação que denotam essa situação. Quando um ou mais casos de teste é mal sucedido a barra de status fica na cor vermelha e o caso de teste falho recebe uma marcação indicando que foi falho. A Figura 4.1 mostra os resultados obtidos dos testes, dos quais um dos testes evidenciou falha na funcionalidade.

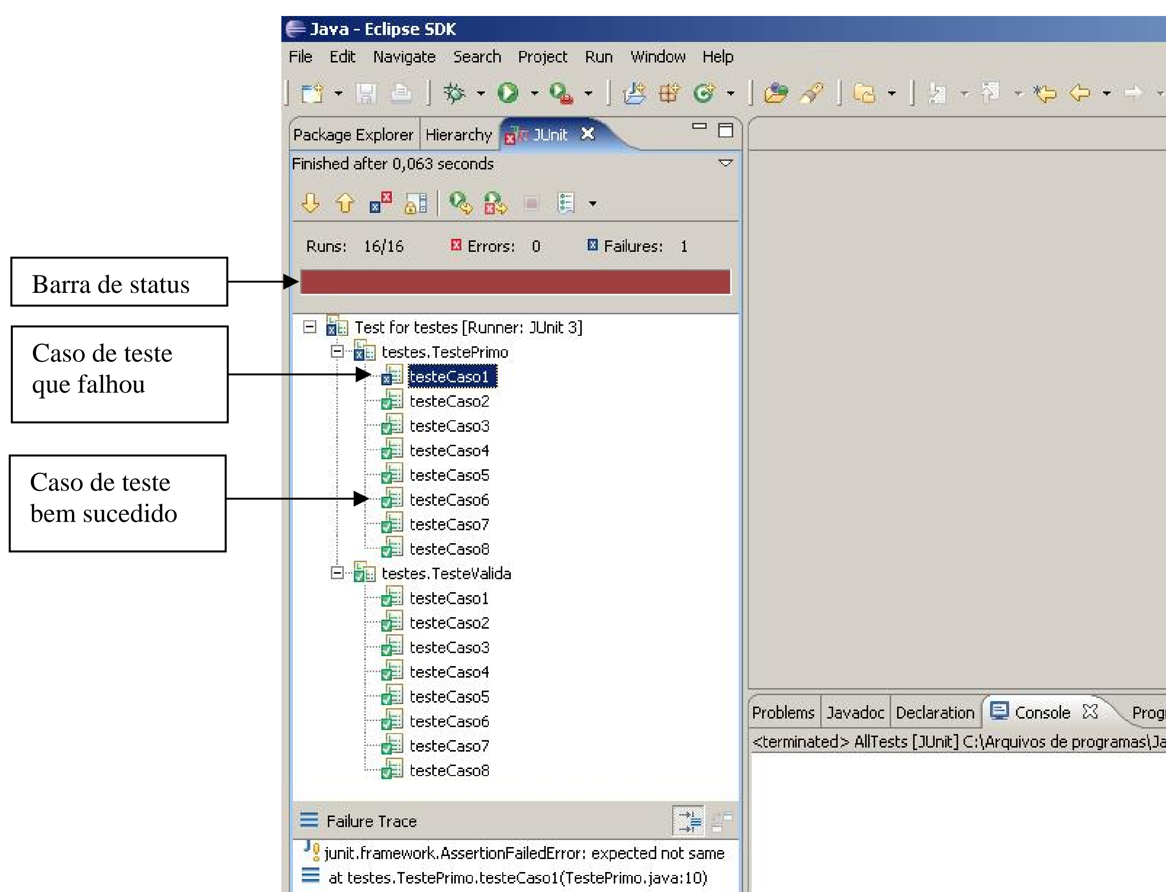

Figura 4. 1- Resultados de testes no JUnit.

Para esse exemplo foram criadas duas classes de negócio sendo que, para cada classe, foram criados oito métodos correspondentes a cada teste a ser executado.

4.3.2. Automação de Testes de Aceitação com Selenium

A automação dos testes de aceitação consiste em criar ferramentas que simulem as ações que o usuário possa exercer sobre a aplicação, executando eventos, atribuindo valores e outros (MARTIN, 2005).

Para a automação dos testes de aceitação, é necessário que haja, no mínimo, três separações na arquitetura do projeto, uma referente ao "*contorno*" da interface, uma que isola as rotinas de teste e uma referente ao acesso à base de dados (MARTIN, 2005). Essas três separações na arquitetura são descritas na Tabela 4.2 e visualizadas nas Figuras 4.2, 4.3 e 4.4, respectivamente.

Tabela 4. 2 - Automação dos testes de aceitação (MARTIN, 2005).

Separações	Motivo
Contorno da Interface	As rotinas de teste devem ser independentes da camada de interfaces do usuário. Elas devem ter acesso direto às entidades referentes às regras de negócio, substituindo as interfaces. Esse contexto é mostrado na Figura 4.2
Isolando os testes	Em sistemas com interfaces de usuário complexas, dentro da camada das interfaces, uma nova camada, referente aos testes da interface (apresentação), deve ser criada. Dessa forma, a arquitetura do projeto apresenta uma camada referente às regras de negócio, na qual atuam os testes que tratam essas regras e a camada de apresentação, na qual atuam as interfaces do projeto e os testes que tratam a camada de apresentação. Esse contexto é mostrado na Figura 4.3.
Separando a base de dados	O acesso à base de dados pode fazer com que a execução dos testes seja lenta. Para resolver esse problema pode-se quebrar a base de dados em pedaços que sejam relacionados às funcionalidades testadas, gerando pequenos bancos de dados. Outra possibilidade é trabalhar com os dados na memória e não nas unidades de armazenamento. Quando as rotinas de testes são iniciadas esses dados são gerados e, quando finalizadas, os dados são destruídos. A Figura 4.4 ilustra a separação da base de dados da lógica do domínio. A separação da lógica do domínio (regras de negócio) da lógica de acesso a dados é aconselhada.

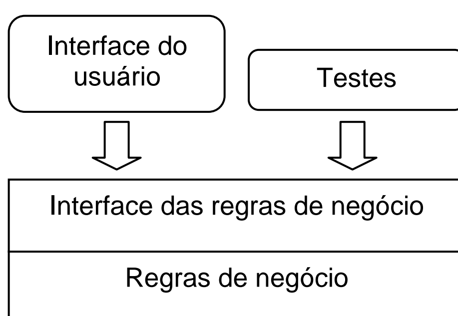

Figura 4. 2 – Teste independente da interface do usuário (adaptado de Martin, 2005).

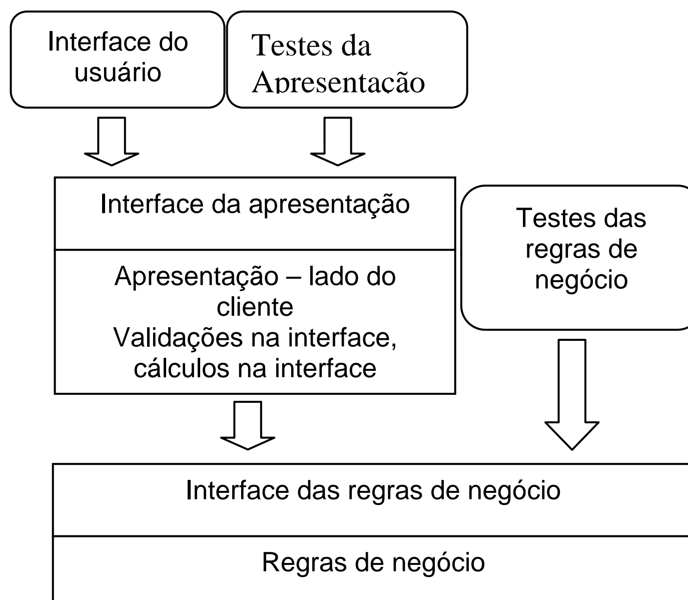

Figura 4. 3 - Isolando os testes de interface dos testes das regras de negócio (adaptado de Martin, 2005).

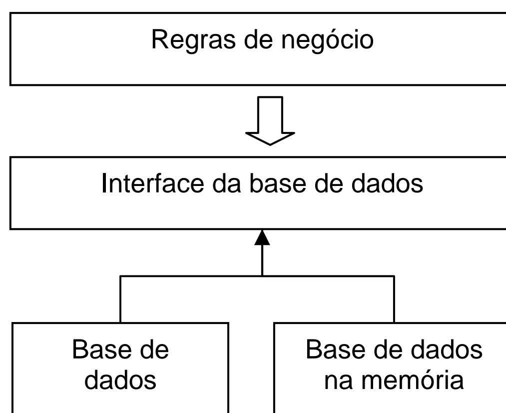

Figura 4. 4 - Trabalhando com a base de dados (adaptado de Martin, 2005).

A definição de quais testes de aceitação serão automatizados resultará da complexibilidade da funcionalidade testada (MYERS, 2004). Um exemplo dessa situação seria a validação, pelo cliente, das telas da aplicação, em que, nessa situação, não haveria a necessidade de automação do teste. As especificações que envolvem

funcionalidades críticas da aplicação e compreendem alta complexidade devem ser automatizadas, tendo como exemplo o cálculo de custo médio de uma empresa.

Holmes e Kellog (2006) citam que os testes de aceitação, normalmente, são feitos manualmente, ao contrário do teste de unidade.

FitNesse e *Selenium* são ferramentas utilizadas para a automação dos testes de aceitação.

O *FitNesse* permite que clientes, testadores e programadores, através da colaboração mútua, especifiquem as funcionalidades que o software deve executar. A ferramenta compara, automaticamente, as especificações com o comportamento apresentado pela aplicação, comparando as expectativas do cliente, dos testadores e dos programadores, aos resultados reais obtidos.

Semelhantemente ao *FitNesse*, o *Selenium* é uma ferramenta de testes para aplicações web. Os testes rodam diretamente no *browser*, simulando exatamente as operações que os usuários reais fazem na aplicação (HOLMES, KELLOG, 2006; DSDM).

A seguir é mostrado um exemplo de utilização da ferramenta *Selenium*. Para este exemplo foram utilizados:

- Ferramenta de teste para aplicações web Selenium, versão 0.8.7, instalada como *plugin* do Navegador Web.
- Navegador Web FireFox 2.0.0.7.

Especificação: Validar a operação de inclusão de um contato em uma aplicação *web* que armazena contatos.

O caso de teste de aceitação a ser automatizado nesse exemplo é mostrado na Tabela 4.3.

Tabela 4. 3 - Caso de teste de aceitação.

Caso de teste	Teste	Dados de entrada	Resultado Esperado
#1	Validar inclusão de um contato	Código, nome, telefone1, telefone2 e e-mail	“Cadastrado”

Para se automatizar o caso de teste de aceitação mostrado na Tabela 4.3, inicialmente executa-se a ferramenta *Selenium* IDE. Nesse exemplo, ela foi instalada como um *plugin* do Navegador web *FireFox*. A interface da ferramenta é mostrada na Figura 4.5.

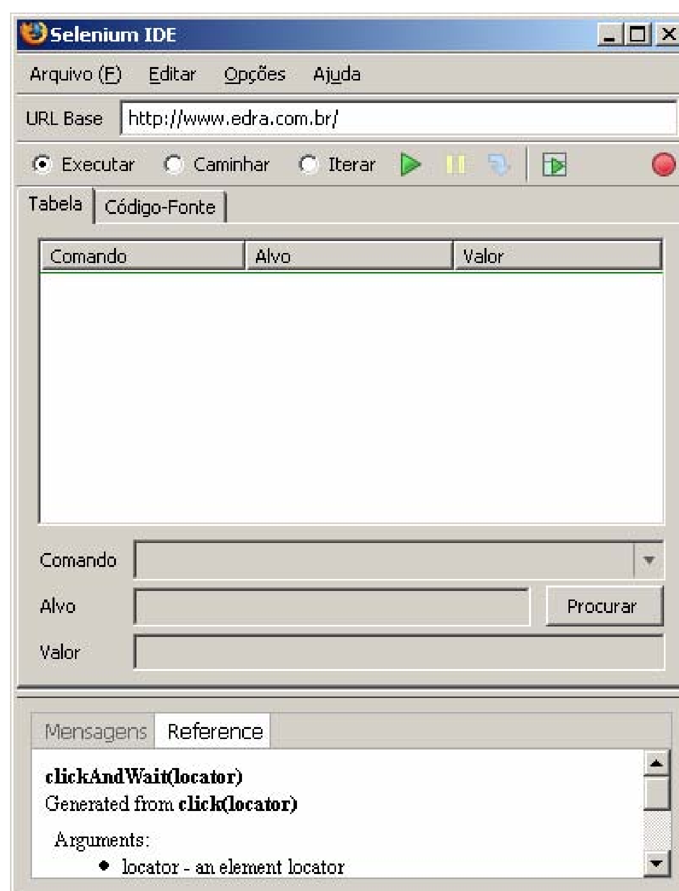

Figura 4. 5 - Interface da ferramenta Selenium IDE.

A automação dos casos de teste na ferramenta *Selenium* consiste em fornecer as ações a serem executadas, os dados de entrada e o resultado esperado para o teste. Essas informações são inseridas na ferramenta através de comandos que são registrados em uma tabela. Para cada comando são informados o seu alvo e valor. A existência desses argumentos, alvo e valor, depende do comando a ser utilizado, por exemplo, para o comando *open*, responsável por abrir uma URL, pode ser especificado somente seu alvo, enquanto para o comando *type*, utilizado para especificar um texto em um campo texto, são informados o alvo e o valor. O comando refere-se à ação que se deseja

executar, o argumento alvo refere-se à entidade que receberá o dado informado no argumento valor. Como exemplo, para se especificar um conteúdo em uma caixa de texto em um formulário web, seria utilizado o comando *type*, informando como alvo o identificador do componente (nome do componente) e no argumento valor, o conteúdo a ser inserido no componente. A Figura 4.6 mostra a interface do *Selenium* com dois comandos especificados, o *open* e o *type*.

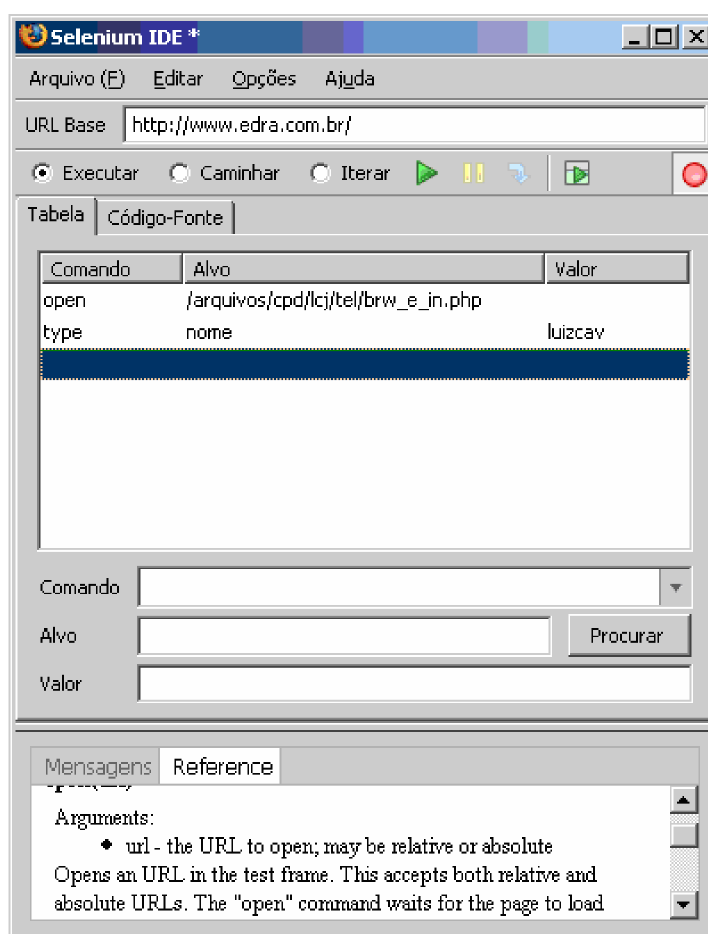

Figura 4. 6 - Comandos *open* e *type* e seus argumentos.

Para a automação do caso de teste contido na Tabela 4.3, inicialmente deve-se levantar as ações que seriam executadas pelo usuário, tais como acesso a uma URL, clique em botões e inserção de dados. O resultado esperado proveniente das ações do usuário também deve ser determinado. As ações e o resultado esperado para o caso de teste #1 são mostrados na Tabela 4.4.

Tabela 4. 4 - Operações a serem realizadas para o caso de teste #1 da Tabela 4.3.

#Id	Tipo	Descrição
1	Ação	Abrir a URL http:\\www.edra.com.br\arquivos\cpd\lcj\tel\bro_e_in.php
2	Ação	Informar o código para o novo contato
3	Ação	Informar o telefone 1 do contato
4	Ação	Informar o telefone 2 do contato
5	Ação	Informar o e-mail do contato
6	Ação	Confirmar a inclusão
7	Resultado	"cadastrado"

Definidas as operações para o caso de teste, essas devem ser passadas para a ferramenta *Selenium*. Essas operações podem ser inseridas manualmente ou, caso a funcionalidade já esteja implementada, através do recurso de gravação da ferramenta. Esse recurso consiste em criar um caso de teste registrando todas as ações executadas e dados fornecidos pelo usuário à medida que a aplicação é utilizada. O usuário também registra a saída obtida através das ações executadas e dos dados fornecidos. Esse caso de teste pode ser gravado e, consequentemente, executado a qualquer momento.

A Figura 4.7 mostra as operações da Tabela 4.4 no *Selenium*.

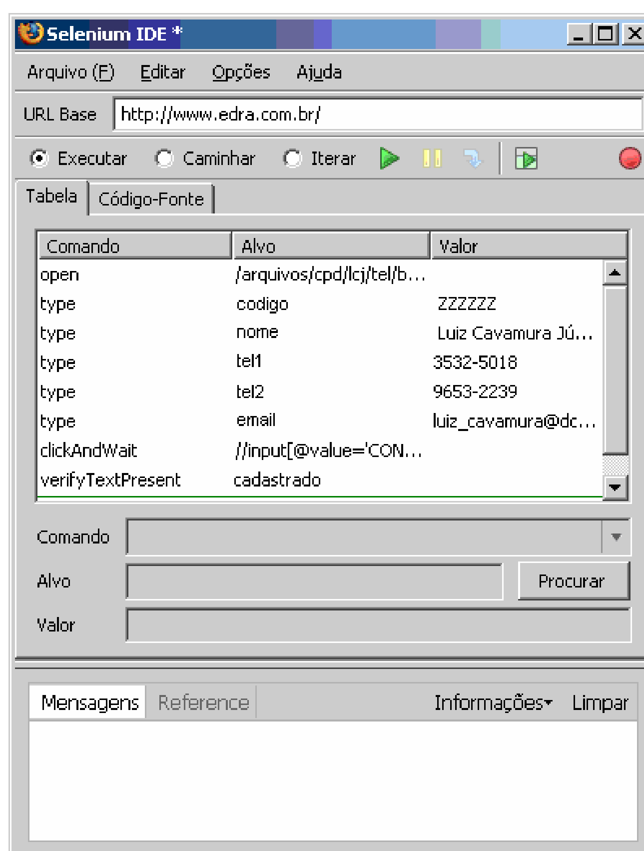

Figura 4. 7 - Operações registradas no Selenium.

Ao executar o caso de teste, esse é bem sucedido, ou seja, o *Selenium* conseguiu executar todas as operações do caso de teste sem apresentar erros, incluindo o contato na aplicação. Quando o teste é bem sucedido, na interface da ferramenta, a linha que contém o comando que verifica a igualdade entre o retorno obtido e o retorno esperado, nesse exemplo o comando *verifyTextPresent*, fica na cor verde. Além dessa indicação, na mesma interface, na guia "Mensagens", é mostrado que o comando *verifyTextPresent* não retornou o valor *false*, ou seja, o retorno obtido foi o retorno que era esperado, que nesse exemplo é a expressão *cadastrado*. A Figura 4.8 mostra o caso de teste bem sucedido.

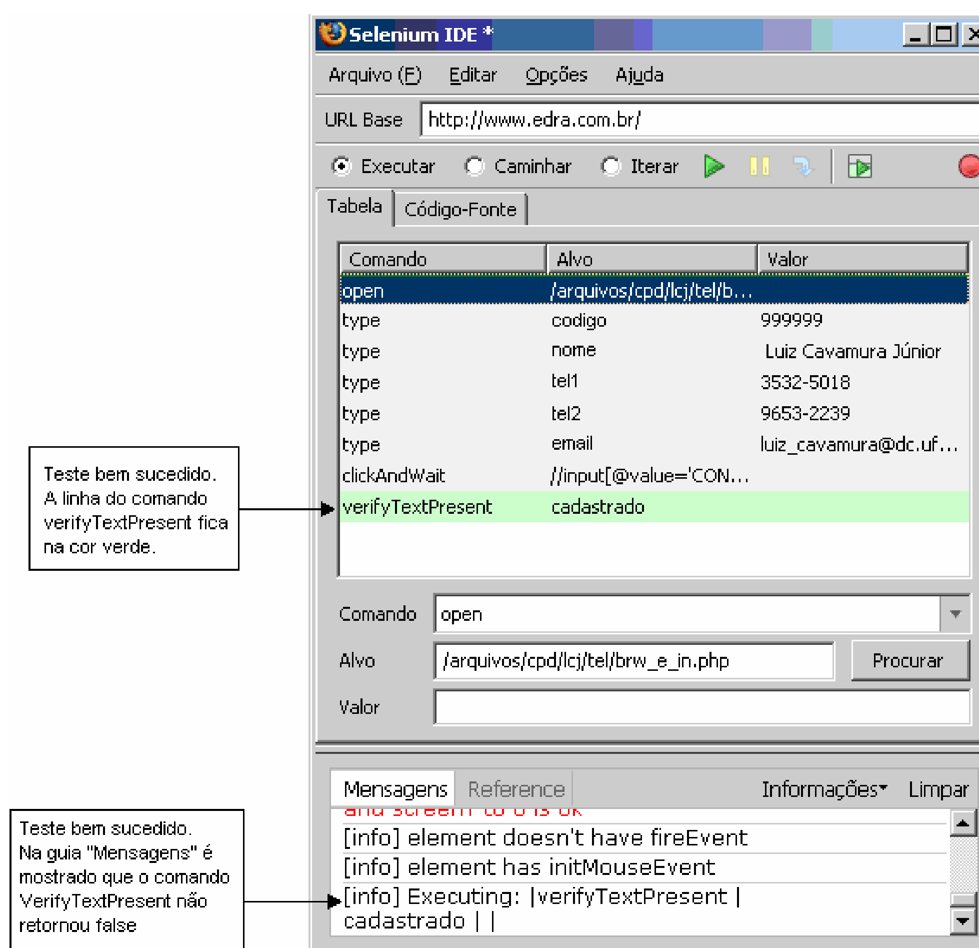

Figura 4. 8 - Caso de teste bem sucedido.

Ao executar o mesmo caso de teste novamente, esse agora não é bem sucedido. Isso se deu porque o caso de teste tentou inserir um contato já cadastrado na aplicação. Ao tentar inserir um contato existente, o resultado esperado não foi a mensagem "cadastrado", conforme especificado no caso de teste, ocasionando uma falha no comando que identifica o resultado esperado. Quando o teste é falho, na interface da ferramenta, a linha que contém o comando que verifica a igualdade entre o retorno obtido e o retorno esperado, nesse exemplo o comando *verifyTextPresent*, fica na cor vermelha. Além dessa indicação, na mesma interface, na guia "Mensagens", é mostrado que o comando *verifyTextPresent* retornou o valor *false*. A Figura 4.9 mostra que o caso de teste falhou.

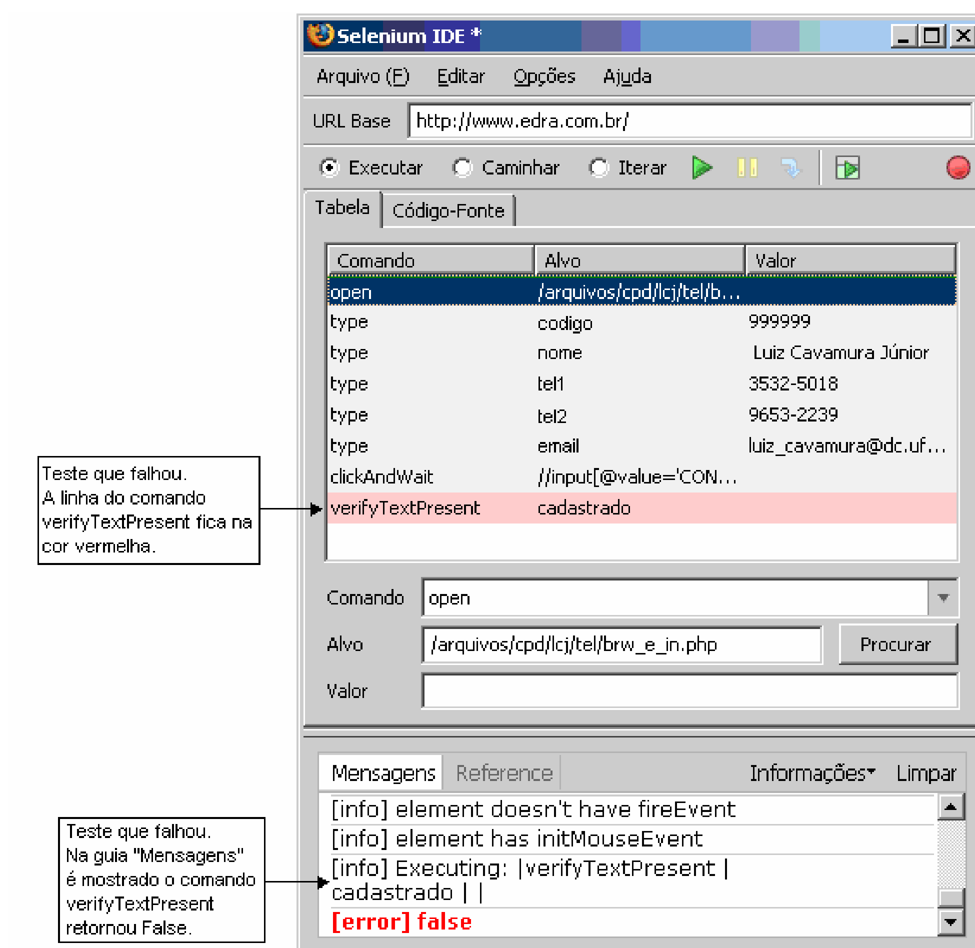

Figura 4. 9 - Caso de teste que falhou.

4.4. O Testador no Contexto Ágil

No contexto ágil, os testes de unidade são escritos pelos programadores antes da implementação e os testes de aceitação são, conforme os princípios do XP, escritos pelo cliente. Com esses fatos, qual seria o papel do testador dentro de uma equipe que utiliza os MA?

Em projetos ágeis, segundo Talby et al. (2006), cada integrante da equipe é responsável por testar os resultados de seu trabalho.

Nos testes de unidade, através do DOT, os programadores escrevem o código para que o caso de teste criado seja bem sucedido. Nos testes de aceitação, a utilização dos programadores para criar os casos de teste não é indicada. Eles não querem encontrar falhas no código que eles mesmos escreveram (CRISPIN, 2006).

Deixar a responsabilidade dos casos de teste de aceitação para o cliente também não é indicado. O cliente, ao escrever os casos de teste de aceitação, estará focado em verificar se a aplicação realiza a funcionalidade que ele deseja, ou seja, o curso normal da funcionalidade. O caso de teste sendo bem sucedido para uma determinada situação pode levar o cliente a generalizar o resultado para as outras situações, não se preocupando com outras entradas, condições ou restrições, não percebendo detalhes não relacionados às regras de negócio. Crispin (2006) relata que é difícil para o cliente visualizar aspectos não relacionados às regras de negócio, tais como inconsistências na aplicação.

Em um projeto ágil, o testador não exerce as mesmas responsabilidades que exerceria em metodologias tradicionais (RASMUSSON, 2003)

Dadas as considerações acima, o papel do testador é importante no contexto ágil, executando as seguintes tarefas (RASMUSSON, 2003; CRISPIN, 2006; TALBY et al., 2006):

- Auxiliar o cliente transmitir as funcionalidades a serem implementadas.
- Auxiliar a esclarecer dúvidas do projeto.

- Auxiliar os desenvolvedores e o cliente a escrever os testes de aceitação (ferramentas, *scripts* de automação, dados de teste).
- Focar na qualidade da aplicação.
- Coletar métricas referentes à produtividade da equipe.
- Testar a aplicação sob perspectiva da usabilidade e com foco nas expectativas do cliente.
- Auxiliar o planejamento das iterações.
- Atuar junto ao cliente e a equipe, contribuindo com o entendimento entre eles.
- Gerenciar as atividades de teste no geral.

O testador não agrega criando mais casos de teste, mas sim ajudando os desenvolvedores a criar os casos de teste (TALBY et al., 2006).

4.5. Inspeção no Contexto Ágil

Durante o estudo realizado, com exceção do PI-XP (Processo de Inspeção – XP) proposto por Toma (2004), não foram encontradas referências que abordassem as atividades de inspeção nos MA. O PI-XP é uma inspeção ágil, a qual foi desenvolvida sobre o método XP. A Inspeção Ágil define três momentos de validação e uma verificação dentro do ciclo de vida do XP, denominados, respectivamente, Validação I, Validação II, Validação III e Verificação. As validações são voltadas para os requisitos enquanto a verificação está focada no projeto. Essas atividades são executadas tão logo se tenha os artefatos por elas utilizados.

A Validação II, Validação III e a Verificação são executadas através das técnicas de Leitura Ágil LAg1, LAg2 e LAg3, respectivamente, as quais estão disponíveis no Apêndice B. A Validação I, por ser executada através de reuniões, não utiliza técnicas de leitura. As validações e a verificação do PI-XP são executadas pelos integrantes da

equipe de desenvolvimento, os quais exercem os papéis definidos no método XP, abordados no Capítulo 2. A seguir são descritas as validações e a verificação do PI-XP e também sua estratégia de aplicação no processo do método XP.

a) Validação I

Objetivo: Validar as histórias do usuário.

Participantes: Participam da Validação I o cliente sênior, o testador e os clientes representativos. É denominado cliente sênior o representante do cliente que estará junto à equipe de desenvolvimento, e cliente representativo, os usuários responsáveis pelas histórias do usuário.

Execução: A Validação I ocorre com a realização de reuniões entre o cliente sênior, o testador e o cliente representativo. Nessas reuniões, são realizados encontros individuais com cada cliente representativo, validando as histórias que são de sua responsabilidade.

Ao fim da Validação I, todos os clientes representativos devem ter validado as histórias de usuário. De acordo com a quantidade de alterações que se fizerem necessárias em uma determinada história, ela pode ser totalmente reescrita, e um novo encontro com o cliente representativo deve ser realizado.

Após a validação das histórias de usuário na Validação I, os desenvolvedores criam os cartões de tarefa a partir das histórias de usuário validadas. Os cartões de tarefa contêm tarefas a serem realizadas pelos programadores.

b) Validação II

Objetivo: Verificar se os conceitos especificados nas histórias do usuário foram bem representados nos cartões de tarefa.

Participantes: Participam da Validação II o testador, pois esteve envolvido na Validação I, o treinador, por possuir conhecimento técnico, porém não esteve presente na elaboração das histórias de usuário e dos cartões de tarefa, e os desenvolvedores, por serem responsáveis pelos cartões de tarefas.

Execução: Essa atividade é executada tão logo se tenham os cartões de tarefa. A Validação II, utiliza a técnica de leitura denominada LAg1 e confronta as histórias do

usuário com os cartões de tarefa. Essa validação tem como saída um relatório de Discrepâncias que serve para registrar as possíveis divergências entre a história de usuário e o cartão de tarefa. Na existência de divergências, o cartão de tarefa deve ser retrabalhado; caso contrário, os cartões de tarefa estão consistentes em relação às histórias do usuário. Enquanto existirem registros no relatório de Discrepâncias, gerado na Validação II, a mesma deve ser repetida.

c) Validação III

Objetivo: Inspecionar os casos de teste de aceitação gerados e verificar se eles retratam as funcionalidades, condições e restrições especificadas nas histórias do usuário.

Participantes: Participam da Validação III o testador, pois este participou da Validação I e auxiliou na elaboração dos casos de teste de aceitação, o treinador, porque não esteve envolvido na elaboração dos artefatos apresentados e, tendo conhecimento técnico, pode realizar a validação sob outra perspectiva, e o cliente, pois escreveu e validou as histórias de usuário e elaborou os casos de teste de aceitação.

Execução: Esta validação é executada tão logo se tenham os casos de teste de aceitação, utilizando-se a técnica de leitura LAg2. Caso algum caso de teste de aceitação não esteja compatível com as respectivas histórias do usuário, ele deve ser retrabalhado. O mesmo relatório de Discrepâncias utilizado na Validação II é gerado caso ocorra discrepâncias. Enquanto existirem registros no relatório de Discrepâncias, a Validação III deve ser repetida.

Terminada a Validação III, as tarefas podem ser implementadas.

d) Verificação

Objetivo: Verificar se os cartões de tarefa, os cartões CRC e os casos de teste de unidade elaborados estão consistentes.

Participantes: A Verificação é executada pelos pares de programadores. Cada programador realiza a verificação através da técnica de leitura LAg3. O testador pode participar dessa atividade pois pode auxiliar os pares de programadores caso necessário.

Execução: A Verificação é executada durante as etapas de implementação do método XP, confrontando os cartões de tarefa, os cartões CRC e os casos de teste de unidade elaborados utilizando a técnica de leitura LAg3.

Na existência de discrepâncias durante a verificação, o relatório de Discrepâncias é gerado. A verificação deve ser refeita até que todas as discrepâncias sejam corrigidas.

Quando os dois programadores terminarem a verificação, os relatórios gerados por cada um são trocados entre eles. Dessa forma, cada programador pode analisar as anotações registradas pelo outro.

e) Estratégia de aplicação

A seqüência das atividades do XP relacionadas aos requisitos do sistema, assim como as validações I, II e III, são mostradas na Figura 4.10.

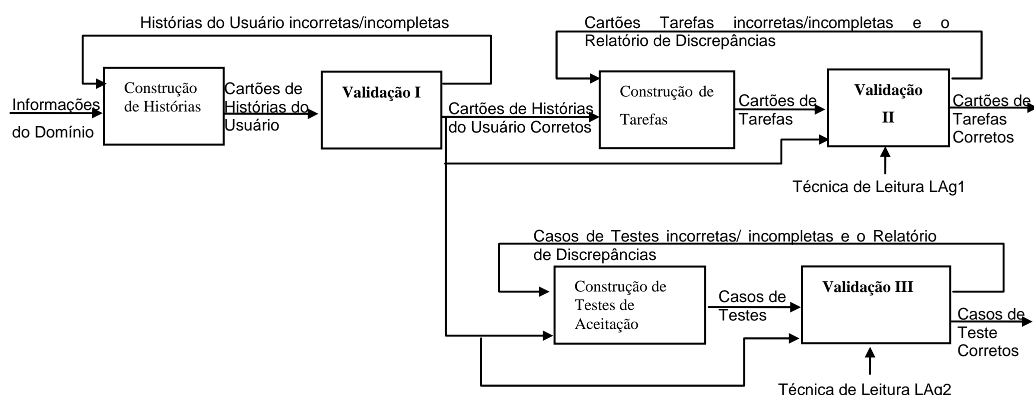

Figura 4. 10 - PI-XP baseado nos requisitos – abstração do processo (TOMA, 2004).

As atividades Validação II e Validação III utilizam as técnicas de leitura LAg1 e LAg2, respectivamente. Essas técnicas também são mostradas na Figura 4.10.

A Verificação do PI-XP está focada no projeto, sendo realizada nas iterações do processo. O momento em que as iterações são executadas no processo do método XP foi abordado no Capítulo 2. Na Figura 4.11 é mostrado como é feito o desenvolvimento em uma iteração do XP e o momento em que a Verificação é executada.

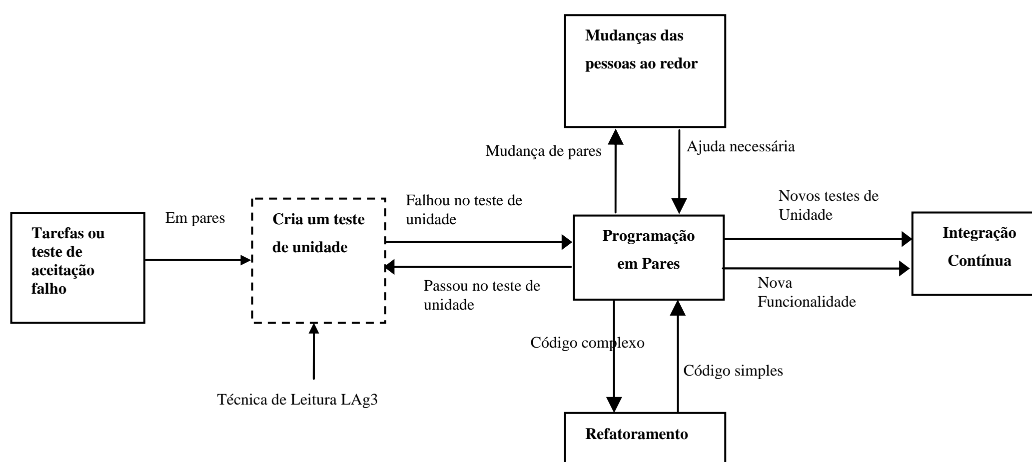

Figura 4. 11 - Detalhe da atividade de desenvolvimento do XP (TOMA, 2004).

A Verificação do processo PI-XP é executada assim que os casos de teste de unidade são criados. Ela utiliza a Técnica de Leitura Lag3, exatamente no momento em que essa atividade é realizada.

Através da estratégia de aplicação do PI-XP, mostrado nas Figuras 4.10 e 4.11, é possível visualizar todo o fluxo de informações utilizado para as atividades de validação e verificação descritas no início desta Seção.

Dada a relevância do PI-XP para este trabalho, um estudo de caso referente ao PI-XP foi realizado para avaliar suas atividades voltadas ao levantamento de requisitos, avaliando também as técnicas de leitura utilizadas nessas atividades (CAVAMURA et al.,2007). Os resultados obtidos nesse estudo de caso evidenciaram as contribuições do PI-XP, focado nos requisitos, ao processo de desenvolvimento.

4.6. Considerações Finais

Neste Capítulo foram apresentadas as atividades de Teste e Inspeção no contexto ágil. O trabalho aqui proposto procurou identificar os artefatos, características, papéis e ferramentas que envolvem essas atividades. Embora a filosofia dessas atividades esteja bem concreta, diretrizes para sua implementação, assim como para os MA, ainda não são bem definidas. Esse fato motivou a proposta que será apresentada no Capítulo 5, procurando estabelecer diretrizes que possibilitem identificar a origem para a criação e utilização dos casos de teste no contexto ágil, não ficando somente nos princípios dessa estratégia.

CAPÍTULO 5

AQUA-Atividades de Qualidade no Contexto Ágil

5.1 Considerações Iniciais

Através do estudo realizado, percebeu-se, nos MA, a falta de diretrizes detalhadas para a realização de todas as suas atividades. Percebem-se também que as publicações voltadas à implantação e utilização de tais métodos, as descrevem em um nível alto de abstração, prendendo-se aos princípios e conceitos do método, omitindo seus detalhes, os quais seriam fundamentais para sua utilização real. Abrahamsson et. al. (2003) comentam que os princípios de tais métodos são bastante enfatizados na literatura, mas evidências baseadas em experiência real são limitadas.

A falta de diretrizes detalhadas para a implantação dos MA é, portanto, observada inclusive na falta de definição das atividades de teste e de planejamento, as quais são citadas como fundamentais para que os processos dos MA funcionem e sejam bem sucedidos. Mesmo o XP tendo como uma de suas práticas a testabilidade, e adotando a geração e execução dos casos de teste antes da implementação do código, não se encontram, na literatura, diretrizes que determinem como é feita essa geração dos casos de teste. Crispin menciona que o DOT não é propriamente uma técnica de teste (CRISPIN, 2006), mas sim uma sistemática de desenvolvimento.

Assim, dado esse contexto, apresenta-se neste capítulo, a abordagem AQUA-Atividades de Qualidade no Contexto Ágil, para melhoria de qualidade no processo dos MA, com foco nas atividades de inspeção, teste e planejamento. Essa abordagem propõe uma sistemática um pouco mais detalhada do que a usual praticada nos MA, no que diz respeito às atividades iniciais do processo, relacionadas com o levantamento dos requisitos e com o planejamento das iterações. Considera-se, nessa

abordagem, que mesmo que um dos princípios dos MA seja “mais código e menos documentação”, um mínimo de documentação é necessária para o sucesso na utilização desses métodos. As atividades propostas na abordagem procuram não causar impactos nos princípios da Metodologia Ágil, tendo como característica, a possibilidade de ser instanciada no processo de desenvolvimento ágil, independentemente do método em questão.

Para apresentar essa abordagem, este capítulo foi organizado da seguinte forma: a Seção 5.2 apresenta uma caracterização dos MA, obtida através do estudo realizado; na Seção 5.3 é apresentada a estratégia proposta e na Seção 5.4 estão as considerações finais.

5.2 Caracterização dos Métodos Ágeis Estudados

Como se pretende que a abordagem aqui apresentada seja genérica, no sentido de poder ser utilizada junto com o processo de qualquer MA, realizou-se um estudo dos seis MA mais evidenciados na literatura, com o objetivo de identificar os artefatos por eles usados e as diferenças e semelhanças entre suas fases. O resultado desse estudo é apresentado nas seções subseqüentes.

5.2.1. Artefatos e Processo

Para que a estratégia proposta tenha como característica a generalidade, de maneira que possa ser aplicada a qualquer método ágil, foram identificados os artefatos gerados nos métodos estudados, os quais são apresentados na Tabela 5.1. Ao tentar identificar esses artefatos pôde-se extrair um processo genérico de desenvolvimento realizado pelos MA, o qual está apresentado na Figura 5.1.

Na Tabela 5.1, os artefatos destacados em negrito são aqueles que devem ser utilizados na aplicação da estratégia proposta, dependendo do método que está sendo utilizado. Além disso, a Tabela 5.1 apresenta atividades ou entidades citadas no processo de cada método, que podem gerar artefatos que não estão declarados explicitamente. Por exemplo, no XP são muito citados os “testes de aceitação” e “testes de unidade”, os quais devem ficar registrados de alguma forma em um artefato, seja ele

manual ou eletrônico.

Tabela 5. 1 – Artefatos usados nos métodos ágeis estudados.

Método	Artefatos identificados no processo dos métodos	Atividades citadas no processo que podem levar a um artefato	Fonte
XP	**-Cartão de história** -Cartão de Tarefas -Cartão CRC	Testes de aceitação Testes de unidade	(BECK, 2004)
Scrum	**-Lista de Trabalho do produto (*Product Backlog)*** -Lista de Trabalho da iteração (*Sprint Backlog*)	(O Scrum não define técnicas e/ou práticas de projeto. A equipe define o que será utilizado)	(SCHWABER, 2004) (ABRAHAMSSON et al., 2002)
FDD	**-Lista de Funcionalidades** -Diagrama de Casos de Uso -Diagrama de Classes -Diagrama de Seqüência -Casos de teste		(HIGHSMITH, 2002) (ABRAHAMSSON et al., 2002) (CHAU et al., 2003)
DSDM	**-Lista de requisitos priorizada** -Lista de requisitos não funcionais -Relatório de risco -Relatório de Viabilidade -Esboço do planejamento geral -Cronograma de tarefas -Plano das iterações -Plano de implementação -Plano de desenvolvimento -Documentação e registro de testes -Documentos de revisão dos modelos gerados	Definição e descrição da área de negócio	(DSDM) (HIGHSMITH, 2002) (ABRAHAMSSON et al., 2002)
Família Crystal (Clear / Orange)	**-Documento de Requisitos** -Diagrama de Casos de Uso -Modelo de Objetos -Cronograma -Plano de liberação e seqüência de entrega -Documentação do projeto de interface -Manual do usuário -Código-fonte -Plano de migração -Documento de especificações internas -Relatório de Status	Aplicação de Testes	(COCKBURN, 2006)
ASD	**-Especificação do Produto (Casos de Uso, Cartões de história do XP, outros semelhantes)** -Plano do projeto -Plano de Ciclo -Lista de componentes de cada ciclo -Cronograma -Lista de riscos -Lista de tarefas -Plano de teste -Casos de teste	<não identificado>	(HIGHSMITH, 2002) (ABRAHAMSSON et al., 2002)

Embora os artefatos da Tabela 5.1 tenham sido identificados na literatura, ressalta-se que não foram encontradas diretrizes que determinam explicitamente o formato desses artefatos e como são gerados, bem como exemplos de utilização. O método ASD, por exemplo, cita a utilização de um plano de teste, de casos de teste, de um plano de ciclo, porém não é definido como são gerados e qual o formato desses artefatos.

Como foi visto no Capítulo 2, os MA realizam a implementação do sistema por partes, as quais são abordadas em cada iteração. Uma iteração pode, então, implementar uma ou mais funcionalidades, sendo que o termo "*funcionalidade*" corresponde a um requisito ou um conjunto de requisitos, que compõe uma funcionalidade a ser oferecida pelo sistema e que está descrita no artefato pertinente a cada método ágil, cujo propósito seja registrar os requisitos definidos pelos clientes. O que será implementado em cada iteração é registrado no artefato pertinente de cada método. Por exemplo, no XP, isso fica caracterizado nas histórias do usuário; no Scrum, através da lista de trabalho do produto e da lista de trabalho da iteração (*Product Backlog*, *Sprint BackLog*); e no DSDM através da lista de requisitos. Uma funcionalidade pode estar contida em único artefato ou em um conjunto de artefatos, assim como um único artefato pode conter várias funcionalidades. Como exemplo, utilizando-se um documento de requisitos, uma funcionalidade pode corresponder a um requisito ou ser composta por um conjunto de requisitos; utilizando-se as histórias de usuário do XP, a funcionalidade pode estar contida em uma história do usuário ou em um conjunto de histórias, assim como uma história pode conter mais que uma funcionalidade.

Com relação ao processo dos métodos estudados, verificou-se que, para todos eles, as funcionalidades são identificadas e registradas, independentemente do artefato utilizado, no início do processo de desenvolvimento. Antes do início das iterações, como uma forma de planejamento, as funcionalidades são selecionadas, definindo-se o que será implementado em cada iteração. O sistema é gerado a partir dessas iterações, permitindo que ocorram entregas freqüentes de software executável (TELES, 2004; BECK, 2004; MANIFESTO, 2001). Em geral, no início do processo de desenvolvimento, a partir das funcionalidades registradas nos artefatos pertinentes a cada método, são trabalhadas, dentre outras, questões referentes a estimativa, prioridade e esforço.

Com base nos estudos realizados sobre esses MA, conclui-se que o processo de desenvolvimento ágil é composto pelas etapas de levantamento das funcionalidades, priorização das funcionalidades e planejamento das iterações, e execução das iterações com entregas freqüentes. Esse processo é representado pela Figura 5.1.

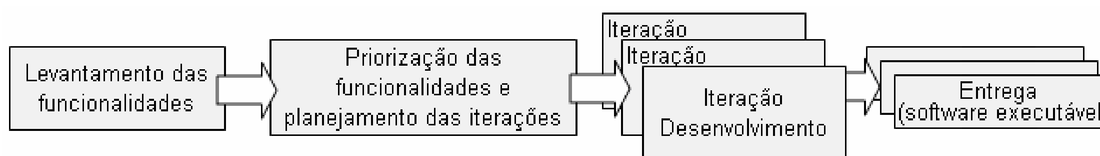

Figura 5. 1 - Processo genérico dos métodos ágeis.

5.2.2. Métodos Ágeis e o Paradigma Orientado a Objetos

Durante o estudo sobre os MA verificou-se também que o desenvolvimento está baseado no paradigma Orientado a Objetos (OO). Os conceitos desse paradigma estão presentes na maioria dos processos que apóiam o desenvolvimento ágil (TOMA, 2004). Além disso, Bellur e Vallieswaran (2006) mencionam que a tendência atual do desenvolvimento OO é a iteratividade, justamente pelo fato desse paradigma estar sendo adotado no contexto ágil. O trabalho de Olague et al. (2007) também faz referência à utilização do paradigma OO no contexto ágil uma vez que faz uma validação empírica de métricas que auxiliam a prever falhas no desenvolvimento de classes OO quando são utilizados MA. Outra referência à utilização do paradigma OO no contexto ágil é a utilização da UML, como pode ser visto na Tabela 5.1 pelos artefatos lá relacionados.

Embora os princípios da metodologia ágil apontem para um processo no qual a relação cliente – desenvolvedor é prioridade e possui maior valor que documentação extensa, a metodologia ágil também ressalta que a atenção à excelência técnica e um bom projeto promovem a agilidade (MANIFESTO, 2001).

Os princípios da orientação a objetos, tais como herança, agregação, composição, e outros, se tornam difíceis de serem identificados quando não são adotadas ferramentas de análise e projeto. Assim, considera-se que a utilização da UML

no contexto ágil, quando adotada e vista como um recurso de auxílio à análise e projeto, e não como extensa documentação, pode propiciar essa excelência técnica.

O XP e Scrum, por exemplo, não fazem referência à UML. No entanto, embora o XP não adote nenhum diagrama da UML, Andrea (2003) cita que o conceito de história do usuário pode estar embutido no caso de uso e, através do estudo realizado, notou-se também uma semelhança entre o cartão CRC e uma classe contida no diagrama de classes. No caso do método Scrum, como ele não especifica práticas de ES, deixando a critério da equipe de desenvolvimento as práticas a serem adotadas, nada impede, por exemplo, que se utilizem os diagramas da UML.

Com base nos estudos realizados sobre a metodologia ágil e os MA, este trabalho propõe uma abordagem para proporcionar maior qualidade ao processo ágil, com base em atividades de inspeção, teste e planejamento do tempo de duração das iterações, que possa ser aplicada, genericamente, a qualquer método ágil. Essa abordagem é denominada AQUA – Atividades de Qualidade no contexto Ágil e envolve atividades de Inspeção, Teste e Planejamento, como é detalhado na próxima seção.

5.3. AQUA – Atividades de Qualidade no contexto Ágil

AQUA é uma abordagem composta por quatro atividades: IFC (Inspeção das funcionalidades junto ao Cliente), IFA (Inspeção das funcionalidades nos Artefatos), ECT (Elaboração dos Cartões de Teste) e ETI (Estimativa de Tempo das Iterações), executadas durante o processo de desenvolvimento ágil, no âmbito do levantamento e definição das funcionalidades do sistema.

O levantamento de requisitos é comum a qualquer projeto, independentemente do paradigma de desenvolvimento adotado (TOMA, 2004). Particularmente, no caso do desenvolvimento ágil, a qualidade está fortemente relacionada com o levantamento e estruturação dos requisitos (ARAUJO, RIBEIRO, 2005).

As atividades da abordagem AQUA utilizam as funcionalidades do sistema e foram propostas para apoiar a condução dos processos ágeis de duas formas: evitar problemas referentes à elicitação de requisitos, uma vez que este é um dos principais problemas apontados nos MA (NAWROCKI et al., 2002), e apoiar as atividades do

desenvolvimento do projeto, abordando os casos de teste e a estimativa de tempo das iterações. A Figura 5.2 apresenta o momento em que essas atividades se aplicam no processo genérico dos MA.

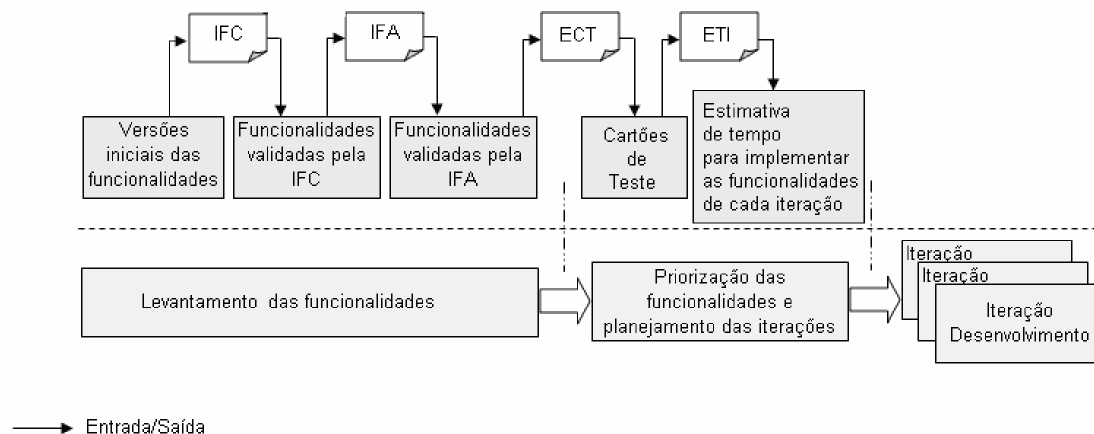

Figura 5. 2 – Atividades da abordagem AQUA

Na Figura 5.2, nota-se que há uma relação de dependência entre as atividades IFC, IFA, ECT e ETI para que elas alcancem seus objetivos. Ressalta-se que essas atividades utilizam os artefatos que contêm a descrição das funcionalidades que o sistema deve atender, de acordo com o método ágil que está sendo utilizado, ou seja, tais artefatos são aqueles mostrados em destaque na Tabela 5.1. Se o método utilizado for o XP, têm-se as histórias do usuário; se for o FDD, o artefato deve ser a lista de funcionalidades e assim por diante. Como as atividades da AQUA utilizam os artefatos em que as funcionalidades estão descritas, a especificação dos casos de uso, embora não contidos na Tabela 5.1, quando mais detalhada que tais artefatos (Tabela 5.1), essa deve substituí-los.

Ressalta-se que o método Scrum não define técnicas e práticas de projeto, ficando a critério da equipe de desenvolvimento o que será utilizado. Assim, de acordo com as técnicas e práticas adotadas pela equipe, outros artefatos, como por exemplo, o diagrama de casos de uso, podem ser utilizados como entrada para as atividades da abordagem AQUA, desde que contenham a descrição das atividades a serem implementadas.

Das atividades da abordagem AQUA, a atividade IFC corresponde à Validação I do PI-XP (TOMA, 2004), diferenciando-se no que diz respeito aos participantes que, neste trabalho, é proposta a participação de dois membros da equipe de

desenvolvimento, enquanto que no PI-XP somente um membro da equipe de desenvolvimento participa da reunião. A atividade IFA está focada no relacionamento entre as funcionalidades, com o objetivo de identificar falhas e inconsistências nas funcionalidades, de acordo como elas foram retratadas nos artefatos utilizados. A atividade ECT, também focada nas funcionalidades, propõe a criação de cartões de teste para aprimorar o processo do método utilizado. Por fim, a atividade ETI propõe estimar o tempo de implementação das funcionalidades de cada iteração, para verificar se estão coerentes com o tempo de duração de cada iteração proposto por cada método ágil, o que contribui também para a melhoria do processo.

Para registrar as ocorrências e discrepâncias encontradas nas atividades IFC, IFA e ECT, recomenda-se, respectivamente, o uso dos formulários de relato de ocorrências – IFC, o formulário de relato de discrepâncias – IFA e o formulário de relato de ocorrências – ECT, os quais são baseados no relatório de discrepâncias de Toma (2004). Esses formulários são apresentados no Apêndice C.

Na próxima seção é apresentada a estratégia de aplicação da abordagem aqui proposta. Um roteiro relaciona os passos a serem seguidos para a aplicação da abordagem AQUA.

5.3.1. A Estratégia de Aplicação das atividades da AQUA

Na Figura 5.3 apresenta-se a estratégia de aplicação da abordagem AQUA, sob o ponto de vista do processo genérico dos MA, apresentado na Figura 5.1

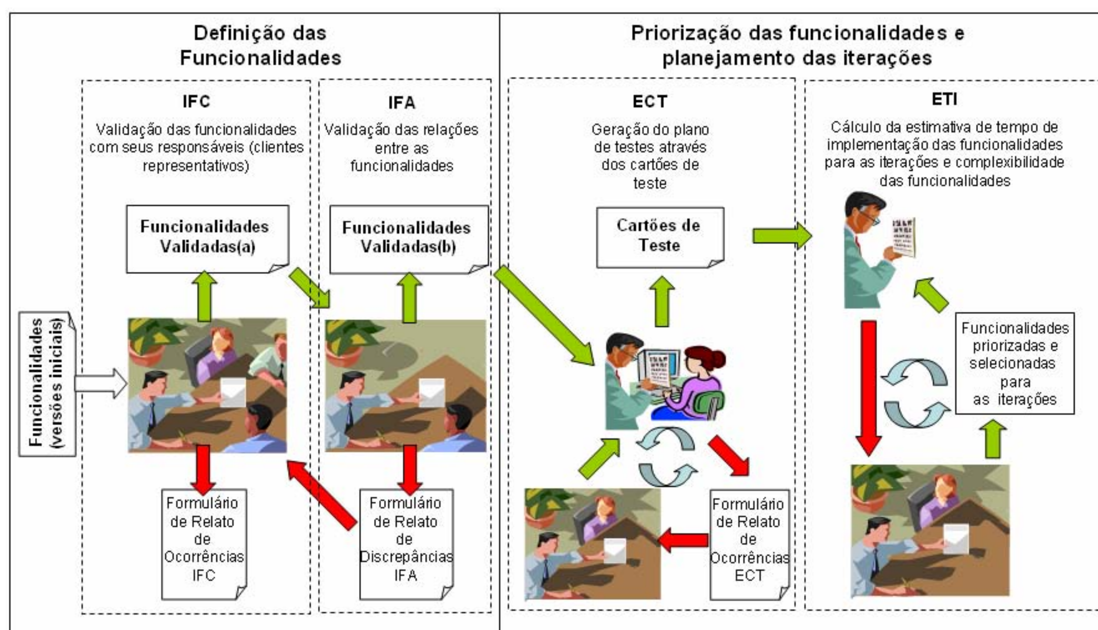

Figura 5. 3 - Estratégia de aplicação da abordagem AQUA.

O roteiro a seguir descreve a estratégia da AQUA:

1) Na fase de levantamento das funcionalidades, faça:

a) Com as versões iniciais das funcionalidades, realize a atividade IFC até que todas as funcionalidades tenham sido inspecionadas.

i) Na ocorrência de problemas:

(1) Registre o problema no formulário de relato de ocorrências – IFC.

(2) Se o problema é proveniente de funcionalidades pertencentes ao cliente representativo presente na reunião, solucione o problema nessa reunião e prossiga com a reunião.

(3) Se o problema na funcionalidade inspecionada provier de funcionalidades de outro cliente representativo, eles devem ser solucionados na reunião com esse outro cliente representativo. Caso já tenha ocorrido essa reunião, uma nova deve ser agendada. Prossiga com a inspeção atual.

b) Com as funcionalidades validadas pela atividade IFC, realize a atividade IFA até

que todas as funcionalidades tenham sido inspecionadas.

i) Na ocorrência de problemas:

(1) Preencha o formulário de relato de discrepâncias – IFA e prossiga com a atividade.

(2) Ao fim da atividade, para cada problema registrado, identifique o cliente representativo das funcionalidades envolvidas no problema.

(3) Agende uma reunião com o cliente sênior para solucionar o problema.

(4) Se o problema não for solucionado com o cliente sênior, execute novamente a atividade IFC para as funcionalidades envolvidas no problema. Estando solucionados os problemas, continue a aplicação das atividades da abordagem AQUA.

2) Na fase de priorização das funcionalidades e planejamento das iterações faça:

a) Com as funcionalidades validadas pela atividade IFA, realize a atividade ECT até que todas as funcionalidades tenham o seu cartão de teste.

i) Na ocorrência de problemas na atividade faça:

(1) Registre o problema no formulário de relato de ocorrências – ECT e prossiga com a atividade.

(2) Ao fim da atividade, todas as ocorrências registradas devem ser analisadas. Para cada ocorrência, verifique sua relevância e, caso necessário, o responsável pela causa da ocorrência deve ser procurado, podendo esse ser o membro da equipe ou da organização cliente. Estando solucionados os problemas e tendo os cartões de teste, continue a aplicação das atividades da AQUA.

b) Com os cartões de teste gerados pela atividade ECT e as funcionalidades separadas por iteração (definidas pela equipe de desenvolvimento e a organização cliente; constatada nos métodos estudados), realize a atividade ETI para cada iteração prevista.

i) Verifique se o tempo estimado obtido pela atividade está coerente com o tempo de duração da iteração.

ii) Se o tempo estimado estiver coerente com a duração da iteração, encaminhe os cartões de teste para os programadores para que os auxiliem no desenvolvimento das iterações do projeto.

iii) Caso o tempo estimado ultrapasse o tempo de duração da iteração definido pelo método faça:

(1) Reporte o problema ao líder da equipe responsável pelo projeto.

(2) Realize uma reunião com o cliente sênior e o responsável pela equipe de desenvolvimento para definir quais ações serão tomadas: se o planejamento das iterações será refeito ou se ele será mantido mesmo que ultrapasse o tempo de duração da iteração proposto pelo método utilizado. Caso seja necessário realize a atividade ETI novamente.

Nas seções subseqüentes, as atividades da abordagem AQUA são apresentadas. As atividades IFC, IFA e ECT são descritas de acordo com os seguintes parâmetros: *Objetivo*, que estabelece o que a atividade se propõe a fazer; *Embasamento*, que apresenta a motivação para definição da atividade; *Evidência do Embasamento*, na qual são identificadas situações que mostram que o embasamento faz sentido; *Artefatos de entrada para a atividade*, que relacionam quais artefatos devem ser utilizados como entrada; *Participantes,* que relaciona quem deve participar da atividade; *Como*, que descreve a maneira que a atividade é executada; *Quando*, que diz em que momento a atividade é executada; *Saídas*, que relaciona quais os documentos gerados pela atividade; *Observações*, que serve para registrar qualquer detalhe importante sobre a atividade. A descrição da atividade ETI também é feita de acordo com esses parâmetros, não constando apenas as evidências do embasamento por não se tratar de uma atividade que visa melhorar o planejamento do processo, diferentemente das demais atividades que estão voltadas para a inspeção e teste.

5.3.2. Atividade IFC – Inspeção das Funcionalidades junto ao Cliente.

Objetivo: Validar as versões iniciais das funcionalidades. Essa atividade procura identificar possíveis erros e obter maior entendimento das funcionalidades. Ela foi gerada com base na Validação I do PI-XP.

Embasamento: Embora os MA enfatizem a necessidade da presença constante do cliente junto à equipe de desenvolvimento (MANIFESTO, 2001), esse princípio nem sempre é aplicado dado o custo gerado por manter o funcionário da organização cliente fora de suas atividades produtivas. Isso justifica a proposta dos papéis do cliente sênior e do cliente representativo na organização cliente, ambos utilizados na Validação I do PI-XP.

A existência dos papéis do cliente sênior e o cliente representativo não são definidos nos MA estudados, porém, a existência desses papéis na organização cliente é verificada na maioria das empresas através dos cargos de gerência e subordinados.

O cliente sênior é o elo entre a equipe de desenvolvimento e a organização cliente, responsável pelo sistema em geral, enquanto os clientes representativos são as pessoas que terão maior interação com as funcionalidades específicas do sistema.

Durante o levantamento das funcionalidades em suas versões iniciais, ou seja, no levantamento de requisitos, o cliente sênior, embora tenha o domínio das regras de negócio, ele não conseguirá fornecer todos os detalhes para a implementação das funcionalidades em um único momento. Isso pode ocasionar dúvidas à equipe de desenvolvimento e inconsistências entre essas funcionalidades, aumentando a possibilidade de inconsistências e falhas. Assim, no levantamento das versões iniciais das funcionalidades, podem ser omitidos detalhes fundamentais necessários para o desenvolvimento das funcionalidades.

Mesmo adotando o papel do cliente sênior, em geral, este também não tem disponibilidade total para atender as dúvidas da equipe de desenvolvimento a qualquer momento.

Evidência do embasamento: Os exemplos que são mostrados foram obtidos em duas empresas, aqui denominadas “A” e “B”, cujo segmento de mercado não é voltado à Tecnologia da Informação. Esses exemplos mostram falhas que podem ocorrer na especificação das funcionalidades, principalmente quando elas são escritas somente por membros da organização cliente.

A empresa “A” atua no segmento de transformação de fibra-de-vidro. Atualmente conta com, aproximadamente, quatrocentos funcionários e um faturamento mensal de seis milhões de reais. Essa empresa utiliza um sistema de gestão que abrange todos os seus departamentos e áreas produtivas. A empresa “B” atua no segmento de distribuição de materiais de construção. É considerada uma empresa de pequeno porte, tendo hoje, aproximadamente, 10 funcionários e não possuindo um sistema de gerenciamento integrado. Embora as empresas não utilizem MA, os exemplos citados mostram situações que aconteceram na prática e que podem ocorrer, independentemente do método utilizado.

Na empresa “A”, o gerente de vendas solicitou ao Departamento de Informática que disponibilizasse no sistema de gestão uma funcionalidade que fornecesse um acompanhamento de cada pedido de venda emitido pela empresa, de acordo com suas necessidades. O departamento de informática solicitou ao gerente de vendas que redigisse essas necessidades para que pudessem ser implementadas. Nesse contexto, o gerente de vendas da indústria assumiu o papel de cliente sênior, enquanto o Departamento de Informática assumiu o papel da equipe de desenvolvimento.

A Figura 5.4 mostra o documento redigido pelo cliente sênior, entregue à equipe de desenvolvimento, representando a funcionalidade por ele desejada.

CONTROLE DO CAMINHO DO PEDIDO DE VENDA

N. DO PEDIDO | | | | | OBSERVAÇÃO
EMPRESA
PRAZO ENTREGA

COMERCIAL	CHEG.	HORA	SAIDA	HORA
CUSTOS	CHEG.	HORA	SAIDA	HORA
DIRETORIA	CHEG.	HORA	SAIDA	HORA

Figura 5. 4 - Especificação da customização a ser feita no sistema de gestão empresarial.

Nesse documento, o cliente sênior indicou quais as informações que deseja obter para cada pedido de venda emitido, tais como "N. DO PEDIDO", "EMPRESA", "PRAZO ENTREGA", "OBSERVAÇÃO" e as datas e horários de chegada e saída do pedido em cada departamento, como por exemplo:

"COMERCIAL CHEG. HORA SAÍDA HORA"

Observando a Figura 5.4 é possível perceber também que o cliente sênior focou apenas no resultado que ele espera obter do sistema. Em sua especificação não são citadas situações como liberações, faturamentos e expedições parciais, as quais podem ocorrer sem critério de seqüência, situações essas que são de total conhecimento dele e são executadas pelos funcionários de cada departamento, que seriam, no caso, clientes representativos. Apenas foram especificadas informações que são de seu interesse próprio, tais como dados de entrada e saída do pedido nos departamentos comercial, custos e outros, não se atentando, além das situações citadas acima, de como é o processo de cada um desses departamentos, por exemplo, reprovações e revisões do pedido no departamento de custo. Outro fato não atentado pelo cliente sênior é que, embora mencionado o termo "N. DO PEDIDO" na especificação da funcionalidade (Figura 5.4), esse ainda não existe nas primeiras etapas da especificação, pois essas etapas tratam justamente a aprovação da venda para a geração do Pedido. O cliente está tão focado no que espera obter, que detalhes das funcionalidades acabam não sendo relatados.

A Figura 5.5 é outro exemplo que mostra funcionalidades descritas pelo cliente sênior da empresa "B". O proprietário da empresa, querendo melhorar as atividades de sua empresa, solicitou a uma empresa de desenvolvimento de software um sistema que gerencie, de forma simplificada, as tarefas diárias da empresa. Foi solicitado ao proprietário da empresa "B" que descrevesse suas necessidades, relacionadas às atividades da empresa, e suas expectativas com relação ao sistema. O proprietário forneceu um documento, do qual dois trechos são mostrados na Figura 5.5.

Programa p/ Empresa. Cadastro Fornecedores Clientes Transportadora Funcionario Vendedor Despesas	Movimentos controle conta corrente " contas receber " contas apagar Emissão - Duplicatas/Boletos " Notas Fiscais " Orçamentos " Pedidos
A	**B**

Figura 5. 5 - Funcionalidades descritas pelo proprietário da empresa "B".

Assim como no exemplo anterior (Figura 5.4), verifica-se a escassez de informações para implementação do sistema. Embora os sistemas de gestão empresarial tenham funcionalidades comuns à maioria das empresas, alguns detalhamentos são importantes. Na Figura 5.5 item (B), por exemplo, não foram fornecidos detalhes de como são feitos os movimentos nela mostrados.

Nesses dois exemplos, mesmo não utilizando os artefatos da Tabela 5.1 e não utilizando MA, foram mostradas situações em que a atividade IFC atua de forma positiva para evidenciar tais problemas, que podem causar impacto negativo no desenvolvimento do produto.

Artefatos de entrada para a atividade: Artefatos destacados em negrito na Tabela 5.1, de acordo com o método utilizado.

Participantes: Cliente sênior, cliente representativo, testador e um outro membro da equipe de desenvolvimento.

Como mostrado no Capítulo 4, mesmo com o DOT, o testador exerce atividades importantes no projeto. Ele possui maior facilidade em contextualizar e identificar possíveis causas de problemas. Essa atividade é realizada em dupla, tendo os mesmos princípios da programação em pares do XP. A presença de dois membros da equipe evita que as informações e sua compreensão fiquem amarradas somente a uma pessoa.

Quando somente uma pessoa detém informações importantes, mesmo que essas estejam registradas, a equipe torna-se dependente dessa pessoa. Essa situação deve ser evitada. Outro incentivo à presença de dois membros da equipe nas reuniões é o fato que haverá duas pessoas analisando simultaneamente a funcionalidade, existindo uma constante

inspeção sobre os pontos discutidos.

Como: A atividade IFC consiste na realização de reuniões junto a cada cliente representativo da organização responsável pelas funcionalidades do projeto. Essas reuniões têm como objetivo validar as funcionalidades correspondentes ao cliente representativo, sendo que o mesmo pode corrigir, acrescentar ou remover funcionalidades. Nessas reuniões deve-se compreender e obter todas as informações necessárias para implementação das funcionalidades, procurando esclarecer questões do tipo "*o que será feito?*" , "*por quê?*", "*como será feito?*" e "*com o que será feito?*". Essas informações são relevantes para a atividade ECT que é a terceira atividade do processo.

Inicialmente, as funcionalidades devem ser identificadas. Conforme mencionado na Seção 5.2, uma funcionalidade pode se referir a um requisito ou a um conjunto de requisitos, podendo um artefato conter uma ou mais funcionalidades, como uma funcionalidade pode estar descrita em um conjunto de artefatos. Identificadas as funcionalidades, são identificados os responsáveis por cada funcionalidade para que as reuniões possam ser agendadas e executadas.

Ao fim dessas reuniões, o cliente representativo deve estar de acordo com as funcionalidades correspondentes a ele. Na ocorrência de problemas com as respectivas funcionalidades, os participantes da reunião devem solucioná-los. Essas ocorrências devem ser registradas no Formulário de Relato de Ocorrências – IFC, mostrado no Apêndice C. Esse registro, além de documentar o ocorrido, servirá como dado histórico para futuros projetos.

Novas funcionalidades podem ser identificadas durante as reuniões. Essas novas funcionalidades devem ser registradas no artefato pertinente. Caso o cliente representativo com o qual esteja sendo feita a reunião no momento seja o responsável por essas novas funcionalidades, elas devem ser validadas nesse momento. Caso contrário, deve-se realizar uma reunião da atividade IFC com os clientes representativos apropriados, mesmo que uma reunião com eles já tenha ocorrido.

Aconselha-se agendar a maior quantidade de reuniões possíveis em uma mesma data, procurando agrupar as reuniões que possuem clientes representativos envolvidos em uma mesma área. Essas ações proporcionam: diminuição do custo de deslocamento da

equipe ou do cliente; diminuição do custo, relacionado ao cliente, referente à quantidade de saídas do funcionário de suas funções diárias; o entendimento do projeto como um todo é facilitado, pois todas as idéias estão recentes.

Quando: O início da realização das reuniões se dá tão logo se tenha o registro das versões iniciais das funcionalidades do sistema.

Saídas: Funcionalidades validadas quanto às expectativas do cliente, as quais são tratadas diretamente com cada responsável.

Observações: Em pequenas organizações, nas quais o escopo do projeto é pequeno, podem-se encontrar situações em que o cliente sênior é o próprio cliente representativo. Mesmo que o levantamento das versões iniciais das funcionalidades tenha sido obtido através do mesmo cliente com o qual será feita a validação, a execução dessa atividade se mantém, pois, em um segundo momento, os membros responsáveis da equipe de desenvolvimento e esse cliente estarão mais amadurecidos em relação a tais funcionalidades.

5.3.3. Atividade IFA – Inspeção das Funcionalidades nos Artefatos

Objetivo: Validar a consistência entre as funcionalidades, estabelecendo uma compreensão do sistema como um todo.

Embasamento: A validação dos relacionamentos existentes entre as funcionalidades auxilia a atividade de seleção e priorização das funcionalidades que serão escolhidas para as iterações uma vez que, quando uma funcionalidade é dependente de outra, ela não poderá ser implementada antes da funcionalidade da qual depende. Embora a atividade IFC tenha validado cada funcionalidade com seu cliente representativo, é necessário validar as funcionalidades de forma global, avaliando as relações que possam existir entre funcionalidades diferentes, verificando se elas estão consistentes. Na atividade IFC, o cliente representativo está focado nas funcionalidades de sua responsabilidade, não visualizando o sistema como um todo.

Evidência do Embasamento: Tomando como exemplo a Figura 5.5, item (B), é solicitado o controle de orçamentos e também o controle de pedidos. Porém, na mesma Figura 5.5, item (A), no qual se encontram os cadastros solicitados, não é mencionado o cadastro de produtos. Supondo que as funcionalidades do item (A) e do item (B) pertençam a clientes representativos distintos, possivelmente detalhes relacionados ao cadastro de produto com as rotinas de controle de orçamento e pedidos não seriam abordados. Esse problema seria visualizado na atividade IFA quando as funcionalidades relacionadas são validadas.

Artefatos de entrada para a atividade: Artefatos destacados em negrito na Tabela 5.1 de acordo com o método utilizado, validados pela atividade IFC.

Participantes: Testador e mais um membro da equipe de desenvolvimento, diferente daquele que participou da atividade IFC. Essa atividade é realizada em dupla, tendo os mesmos princípios da programação em pares do XP. O testador é um dos participantes porque participou da atividade anterior, IFC. O outro membro não deve ser o mesmo para propiciar outra visão das funcionalidades, sem conhecimento dos detalhes discutidos na atividade anterior, possibilitando uma revisão constante.

Como: A atividade é apoiada pelo roteiro apresentado na Tabela 5.2, que contém as diretrizes a serem aplicadas em todas as funcionalidades.

Ao término da validação de todas as funcionalidades, havendo registros no formulário de relato de discrepâncias – IFA verifique quem são os clientes representativos responsáveis pelas funcionalidades com problema e agende uma reunião com eles para que os problemas registrados sejam esclarecidos, ou seja, refaça com eles a atividade IFC.

Tabela 5. 2 - Roteiro da atividade IFA

Etapa	Descrição
Para cada funcionalidade não marcada como revisada faça:	
1	Leia a funcionalidade inspecionada para poder entendê-la.
2	Procure e separe outras funcionalidades que possam estar relacionadas com a funcionalidade inspecionada. As funcionalidades relacionadas podem ser identificadas por terem em sua descrição substantivos candidatos à classe (objetos de análise) em comum com a funcionalidade inspecionada.
2.1	Caso não encontre funcionalidades relacionadas, indique esse fato na funcionalidade inspecionada e recomece esse roteiro a partir do item 1, com outra funcionalidade a ser inspecionada.
2.2	Faça uma referência cruzada entre a funcionalidade inspecionada e as funcionalidades relacionadas, colocando o identificador da inspecionada nas relacionadas e vice-versa. Essas referências evitam que o mesmo relacionamento seja avaliado mais que uma vez.
3	Verifique se a funcionalidade inspecionada e as funcionalidades relacionadas estão consistentes entre si, executando os itens seguintes.
3.1	Verifique se há definições contraditórias entre os conceitos (substantivos candidatos a classes e ações) descritos na funcionalidade inspecionada e na funcionalidade relacionada.
3.2	Se constatar inconsistência entre a funcionalidade inspecionada e a funcionalidade relacionada, registre o problema encontrado no formulário de relato de discrepâncias – IFA.
3.3	Marque a funcionalidade como revisada.
4	Após a validação da funcionalidade inspecionada com as funcionalidades relacionadas, recomece esse roteiro, a partir do item 1, com outra funcionalidade ainda não inspecionada. Todas as funcionalidades devem assumir o papel da funcionalidade inspecionada.

Quando: Essa atividade é executada tão logo se tenham as funcionalidades validadas pela atividade IFC.

Saídas: Funcionalidades validadas e/ou o formulário de relato de discrepâncias - IFA.

Observações: A execução da atividade IFA não necessita de pessoas experientes em atividades de inspeção. Essa atividade pode ser executada por qualquer membro da equipe de desenvolvimento que tenha conhecimento dos objetivos do sistema e experiência em atividades da análise.

5.3.4. Atividade ECT – Elaboração dos Cartões de Teste

Objetivo: Identificar casos de teste com base na descrição das funcionalidades do sistema e registrá-los em cartões de teste como o mostrado na Figura 5.6.

Embasamento: Como visto no Capítulo 4, o testador, no contexto ágil, deve auxiliar os programadores e a organização cliente a identificar os casos de teste do sistema. No entanto, não se identifica, na literatura, como essa ajuda é realizada e nem mesmo como e se os resultados disso são armazenados para consulta posterior. O uso do cartão de teste formaliza e registra essa atividade, auxiliando principalmente os programadores que, a partir desse cartão, terão uma orientação no desenvolvimento do sistema. Esse cartão fornecerá os casos de teste que podem ser utilizados pelos programadores no DOT e também auxiliará a identificar as classes de negócio do sistema.

Outro ponto analisado é que, embora o DOT tenha uma declaração clara de seus objetivos e tenha sido identificado no contexto dos MA, principalmente no XP que adota essa prática, não foram identificadas estratégias para a definição de casos de teste.

A utilização do cartão de teste também pode auxiliar o gerenciamento dos casos de teste, pois, havendo alteração em uma funcionalidade, o cartão de teste é uma referência aos casos de teste gerados para essa funcionalidade, facilitando possíveis adaptações nos mesmos.

Evidência do Embasamento: No exemplo de Teles (2004), citado no Capítulo 4, o autor exemplifica a prática do DOT através da implementação de uma classe que converte um número hexadecimal em um número decimal, mas não cita os casos de teste que tratam outras entradas, diferentes da entrada que gera a saída esperada para essa funcionalidade. No exemplo de Myers (2004), também citado no Capítulo 4, o autor exemplifica a prática do DOT através da implementação de uma classe que verifica se uma dada entrada é ou não um número primo, mas embora tenha gerado um plano de teste a ser seguido, não foi citada uma estratégia para se obter tal plano a partir das funcionalidades do sistema.

A prática do DOT é voltada à criação dos testes de unidade. Quanto aos testes de aceitação, dentre os métodos estudados, o XP é o único que cita e explora casos de teste de aceitação em seu processo. No entanto, no XP os casos de teste de aceitação são

criados e executados pelos membros da organização cliente o que, segundo Talby et al. (2006), pode ocasionar problemas quanto à qualidade desses casos de teste. O membro da organização cliente não consegue expressar todas as validações necessárias a serem aplicadas no sistema para validá-lo totalmente. Os testes de aceitação representam o interesse do cliente e procuram fornecer confiança na aplicação, mostrando que esta contém as funcionalidades especificadas e que elas realizam o que foi solicitado. Uma vez que os testes de aceitação capturam as exigências do usuário, eles servem para verificar se a aplicação atende essas exigências (MYERS, 2004).

O membro da organização cliente, ao escrever os casos de teste de aceitação, está focado nos seus interesses e, geralmente, por não ser um testador, planeja os casos de teste apenas para o fluxo normal da funcionalidade que, se bem sucedidos, induzirão ao mesmo, erroneamente, a pensar que a funcionalidade está trabalhando corretamente. A execução bem sucedida do fluxo normal não indica que a funcionalidade está trabalhando corretamente porque os fluxos alternativos dessa funcionalidade, ou seja, todas as possibilidades que podem ocorrer para sua execução, tais como suas condições e restrições, podem não ter sido verificadas. Um exemplo comum seria que, dificilmente, o cliente testaria a integridade do sistema quando executadas rotinas de exclusão de cadastros.

Artefatos de entrada para a atividade: Artefatos da Tabela 5.1, de acordo com o método utilizado, validados pela atividade IFA.

Participantes: Testador, pois ele participou das atividades anteriores e é o responsável pelas atividades de teste.

Como: Os casos de teste são identificados através da descrição da funcionalidade do sistema. Esses casos de teste são registrados no cartão de teste proposto na Figura 5.6. Cada funcionalidade gera um cartão de teste.

Myers (2004) recomenda utilizar uma combinação de critérios de teste, pois um critério pode encontrar erros que outro não encontrará. O procedimento recomendado é desenvolver casos de teste utilizando critérios da técnica funcional e complementar com critérios da técnica estrutural.

Técnicas estruturais de teste não são consideradas porque, no contexto do teste ágil, o DOT gera os casos de teste para orientar a implementação. Como o DOT gera os casos de teste antes da implementação, técnicas estruturais não são aplicáveis nesse contexto. Técnicas funcionais devem ser adotadas para identificar os casos de teste do processo, obtendo-se casos de testa a partir das funcionalidades do sistema.

A definição dos casos de teste que compõe um cartão de teste deve ser realizada aplicando-se os critérios de teste da técnica funcional, tais como o *particionamento em classes de equivalência*, *análise do valor limite*, e "*error guessing*". Ressalta-se que o critério da análise causa-efeito, comparado com os outros critérios mencionados, demanda maior tempo de aplicação. Por esse motivo, esta atividade sendo aplicada no contexto ágil, esse critério não foi considerado ao destacar as diretrizes no cartão de teste. A aplicação desses critérios pode ser auxiliada verificando-se algumas diretrizes que permitem identificar suas características, como por exemplo:

No critério ***particionamento em classes de equivalência***, as classes de equivalência, consequentemente os casos de teste, podem ser identificadas através de restrições ou intervalos especificados na descrição da funcionalidade.

O critério da ***análise do valor limite*** é utilizado com base nas classes de equivalência, criadas pelo critério de *particionamento em classes de equivalência*, geradas a partir de intervalos identificados nas funcionalidades.

No critério "***error guessing***"(suposição de erros), os casos de teste podem ser identificados através da suposição de possíveis erros que possam ocorrer e de situações propícias a erro, que possam ser deduzidas da funcionalidade, como por exemplo, validações de pertinência, informações obrigatórias e verificação de consistência.

No critério ***análise causa-efeito***, os efeitos e, consequentemente, suas causas, podem ser identificados verificando-se as possíveis saídas para a funcionalidade, ou seja, possíveis tratamentos que geram outras saídas diferentes da saída esperada.

Para apoiar o testador na geração dos casos de teste, essas diretrizes, com exceção do critério *análise de causa-efeito*, devem estar registradas no cartão de teste, como mostra a Figura 5.6. Os casos de teste definidos e registrados no cartão servirão, basicamente, como testes de unidade, já que são baseados na funcionalidade e procuram explorar as

diretrizes dos critérios, como mencionado anteriormente. Muitos desses casos de teste podem também ser considerados como testes de aceitação, no caso de explorarem a funcionalidade de forma mais ampla e caracterizarem um cenário de uso do sistema.

Cartão de Teste

Funcionalidade:______________________________________

Candidatos à classe envolvidos:

Verificar:

- P.E.
  »Restrições
  »lista ou conjunto de opções
- A.V.L.
  »Intervalos
- E.G.
  »Possíveis erros ou situações que tendem a erros (informações obrigatórias, consistências, outros)

Elaborado por:____________________ Data: ___/___/___

CT	Descrição	Entrada(s)	Saída

- P.E. : Particionamento em classes de equivalência
- A.V.L. : Análise do Valor Limite
- E.G. : *Error Guessing*

Figura 5. 6 – Cartão de Teste proposto.

Como pode ser observado na Figura 5.6, sugere-se que o cartão de teste apresente as diretrizes de cada critério de teste mencionado, a saber:

P.E. refere-se ao critério de particionamento em classes de equivalência; A.V.L. refere-se ao critério da análise do valor limite e E.G. refere-se ao critério "*error guessing*".

Além disso, para preparar as informações para a próxima atividade, o cartão de teste deve também armazenar os substantivos candidatos à classe, identificados e utilizados na atividade IFA. Essas informações serão utilizadas pelo programador e também serão utilizadas para calcular o tempo estimado de duração das iterações. O programador, de

posse dos cartões de teste, implementa os casos de teste de unidade automatizados e identifica possíveis classes e relacionamentos envolvidos na implementação.

Tendo como base os estudos realizados, ressalta-se que atividades CRUD (*Create*, *Read*, *Update*, *Delete*), no contexto ágil, não são explicitamente registradas e detalhadas, estando elas implícitas como funcionalidades a serem implementadas. Como exemplo, essas atividades quando referenciadas nas funcionalidades do sistema, são descritas com o termo "cadastro" e não detalhadamente com os termos "deve permitir incluir...", "dever permitir alterar...", e outros. As ações de CRUD devem ser consideradas ao criar o cartão de teste para a funcionalidade, estando elas explícitas ou não na descrição da funcionalidade.

Na ocorrência de algum problema no decorrer da atividade, esse deve ser registrado no formulário de relato de ocorrências – ECT. Sendo gerado esse formulário, as ocorrências nele descritas devem ser analisadas. Para cada ocorrência verifica-se sua relevância e, caso necessário, o responsável pela causa da ocorrência deve ser procurado, podendo ele ser qualquer membro da equipe ou da organização cliente.

Quando: A geração dos cartões de teste deve ser realizada durante o planejamento das iterações, tão logo se tenham as funcionalidades validadas pela atividade IFA.

Saídas: Cartões de teste do sistema e/ou formulário de relato de ocorrências – ETC.

5.3.5. Atividade ETI – Estimativa de Tempo para as Iterações

Objetivo: Auxiliar o planejamento do tempo de duração de cada iteração, possibilitando:

- Determinar o esforço necessário para implementação das funcionalidades de cada iteração.
- Verificar se as funcionalidades selecionadas para a iteração estão coerentes com o tempo de duração da iteração, de acordo com o método utilizado.

Embasamento: A seleção das funcionalidades que serão implementadas em cada iteração é estabelecida com base na prioridade atribuída a elas, que é definida pela equipe de desenvolvimento e a organização cliente, respeitando-se as necessidades do cliente.

A seleção das funcionalidades, seja ela feita aleatoriamente ou simplesmente com base na seleção estabelecida pelo cliente, sem utilização de métricas de esforço e complexidade, pode violar o planejamento do projeto, ocasionando atrasos no cronograma e, consequentemente, atraso nas entregas. As funcionalidades selecionadas para cada iteração não devem distorcer o tempo de duração da iteração proposto por cada método.

Artefatos de entrada para a atividade: Cartões de Teste gerados na atividade anterior e lista das funcionalidades priorizadas, por iteração. Essa lista foi identificada no processo dos MA estudados no Capítulo 2.

Participantes: o Testador, pois, como citado no Capítulo 4, no contexto ágil, ele não se limita somente a exercer as atividades de teste, mas auxilia na fase de projeto, exercendo outras atividades. Nessa atividade, ele verifica se as funcionalidades selecionadas para cada iteração estão coerentes com prazo de duração da iteração, de acordo com o método utilizado.

Como: A técnica utilizada para estimar o tempo de duração de uma iteração é a técnica de pontos por caso de uso, pois se pode usar, como alternativa ao número de transações, a quantidade de objetos de análise (classes) contidos na especificação dos casos de uso (KARNER, 1993), para estimar o esforço necessário. Os objetos de análise foram identificados nas funcionalidades durante a execução da atividade IFA e foram registrados nos respectivos cartões de teste gerados na atividade ECT. Dessa forma, cada cartão de teste é correspondente a um caso de uso.

Um ponto que se altera na estratégia de aplicação da técnica de pontos por caso de uso é a contagem dos atores. Quando a técnica é aplicada para estimar o esforço da equipe para um sistema, a contagem dos atores é feita uma vez. Na atividade ETI, estima-se o tempo por iteração. Dessa forma, é necessário que a identificação e contagem dos atores também seja feita apenas uma vez para todo o sistema, porém, para estimar o esforço de

cada iteração, o peso dos atores, ao ser adotado para o cálculo de cada iteração, deve ser dividido pelo número de iterações previstas.

Para se estimar o tempo de implementação das funcionalidades da iteração em horas-homem utiliza-se a quantidade de horas-homem utilizadas por ponto de caso de uso. Embora Karner (1993) proponha que essa quantidade seja de 20 horas-homem por ponto de caso de uso, essa quantidade varia conforme a equipe de desenvolvimento. Cabe à equipe determinar qual é seu valor correspondente. Uma forma de determinar a quantidade de horas-homem por ponto de funcionalidade ou caso de uso, seria aplicar esta atividade em projetos já concluídos, encontrando-se o valor de pontos por funcionalidade ajustado para esses projetos. Tendo-se o tempo total gasto no projeto e o valor de pontos por funcionalidade ajustado, determinamos o valor de hora-homem por ponto de funcionalidade fazendo o processo inverso ao da técnica de pontos por caso de uso, ou seja, dividimos o tempo total gasto no projeto pelo valor de pontos ajustados obtido. Dessa forma, obtemos a quantidade de hora-homem por ponto de funcionalidade correspondente à equipe de desenvolvimento. Quanto maior a base empírica, melhor ajustada ficará a quantidade de hora-homem por ponto de funcionalidade.

Obtendo-se o tempo estimado para a implementação das funcionalidades da iteração, verifica-se se essas funcionalidades estão coerentes com o tempo da iteração proposto pelo método ágil de desenvolvimento utilizado e a quantidade de membros da equipe de desenvolvimento.

Caso o tempo estimado obtido ultrapasse o tempo de duração da iteração proposto pelo método, uma reunião, envolvendo o responsável pela equipe de desenvolvimento e o cliente sênior, deve ser realizada. Nessa reunião deve-se definir se o planejamento dessa iteração, e consequentemente de outras iterações, serão mantidos ou serão refeitos para que se enquadre no tempo de duração da iteração. Caso o planejamento seja refeito, esta atividade deve ser refeita.

Quando: Esta atividade deve ser realizada durante o planejamento das iterações, tão logo se tenha os cartões de teste das funcionalidades.

Saídas: Estimativa de tempo para implementação das funcionalidades selecionadas para a iteração.

5.4. Considerações Finais

Neste Capítulo foi apresentada a abordagem AQUA que tem por objetivo melhorar a qualidade do processo dos MA. Ela foi definida tendo como base um processo genérico, o qual foi identificado depois de se estudar os MA XP, Scrum, DSDM, FDD, ASD, Crystal Clear e Crystal Orange. Nesse estudo foram também identificados alguns artefatos utilizados por esses métodos, com os quais a abordagem é aplicada. Essa abordagem é composta de quatro atividades: IFC – Inspeção das Funcionalidades junto ao Cliente, IFA – Inspeção das Funcionalidades nos Artefatos, ECT – Elaboração dos Cartões de Teste e ETI – Estimativa de Tempo para as Iterações, que se concentram na fase de definição das funcionalidades do sistema.

A melhoria do processo se dá pelas atividades de inspeção IFC e IFA, que têm por objetivo validar as versões iniciais dos requisitos e adquirir um melhor entendimento deles; pela atividade de teste ECT, que tem por objetivo formalizar o papel do testador no contexto ágil e auxiliar na identificação dos casos de teste a serem utilizados e implementados; e pela atividade de planejamento ETI, que tem por objetivo estimar o tempo de duração das iterações, auxiliando no planejamento do processo.

Como foi identificado no estudo realizado acerca dos MA selecionados, as atividades de teste e de planejamento são essenciais para o sucesso de suas aplicações e, além disso, é certo que algumas informações sobre essas atividades devem ficar registradas de alguma forma para que possam ser usadas ao longo de desenvolvimento. No entanto, não fica claro na literatura como isso é feito, o que dificulta sobremaneira a adoção desse paradigma de desenvolvimento. Assim, antes de tudo, a abordagem aqui proposta procura dar algumas diretrizes de como fazer e como armazenar informações que são relevantes para a adoção de atividades de garantia de QS no contexto ágil de desenvolvimento.

O próximo capítulo mostra um exemplo de uso da abordagem AQUA com o intuito de verificar sua viabilidade de aplicação e sua possível contribuição para a melhoria da qualidade do processo.

CAPÍTULO 6

EXEMPLO DE APLICAÇÃO

6.1. Considerações iniciais

No Capítulo 5 foi apresentada a abordagem AQUA que dá apoio às atividades de garantia de qualidade no contexto dos MA, no que diz respeito às atividades de inspeção, teste e planejamento, tendo como base as funcionalidades do sistema. Ressalta-se que, como verificado através do estudo realizado, os MA não descrevem diretrizes detalhadas das atividades executadas e dos artefatos utilizados, porém, algumas informações, geradas e utilizadas durante o processo são de fundamental importância que fiquem armazenadas de alguma forma para que dêem suporte ao processo de desenvolvimento.

Para ilustrar o uso dessa abordagem, neste capítulo é apresentado um exemplo de aplicação das atividades que a compõem, com o intuito de verificar a viabilidade de aplicação das mesmas. A atividade IFC, por ser uma atividade de inspeção realizada através de reuniões realizadas com os clientes, não será contemplada neste exemplo, pois ele não trata de uma situação real. Esse exemplo utiliza um sistema desenvolvido durante a realização de uma disciplina de pós-graduação na Universidade Federal de São Carlos. A disciplina tinha como objetivo analisar a utilização do método Scrum juntamente com padrões de projeto. Por esse motivo o método utilizado foi o Scrum e o sistema desenvolvido foi um Sistema para Clínica Veterinária (SCV), cujas principais características eram o gerenciamento de serviços e a venda de produtos oferecidos pela clínica.

Este capítulo está organizado da seguinte forma: na Seção 6.2 é apresentada uma breve descrição do sistema exemplo (SCV); na Seção 6.3 é exemplificada a atividade IFA, na Seção 6.4 é exemplificada a atividade ECT; na Seção 6.5 é exemplificada a atividade ETI; e na Seção 6.6 são apresentadas as considerações finais deste capítulo.

6.2. Descrição do Sistema para Clínica Veterinária (SCV)

Como o método ágil utilizado foi o Scrum e ele não determina os tipos de artefatos, voltados às práticas de ES, a serem elaborados, decidiu-se usar o diagrama de casos de uso para representar os requisitos do sistema. Esse diagrama é mostrado na Figura 6.1. Nesse exemplo, os casos de uso correspondem às funcionalidade do sistema utilizadas na abordagem AQUA, sendo cada caso de uso uma funcionalidade.

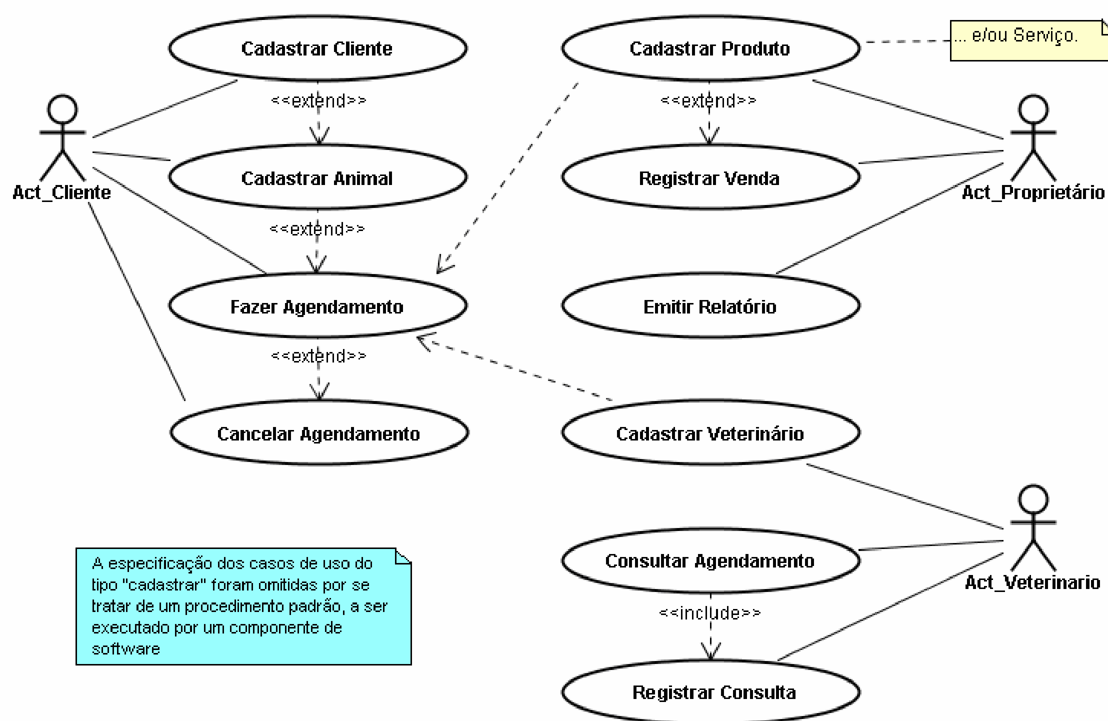

Figura 6. 1 - Diagrama de casos de uso do sistema para Clínica Veterinária (SCV).

Nas Tabelas 6.1, 6.2 e 6.3 são mostradas, como exemplo, respectivamente, a especificação dos casos de uso *Fazer Agendamento*, *Consultar Agendamento* e *Cancelar Agendamento*. Essas especificações são utilizadas para exemplificar a abordagem AQUA.

Tabela 6. 1 - Especificação do caso de uso "Fazer agendamento".

Número	01
Caso de Uso	Fazer Agendamento
Escopo	Sistema Clínica Veterinária
Descrição	Caso de uso referente ao agendamento de consultas com os veterinários cadastrados, na data e hora informadas pelo cliente. O cliente fornece suas informações e as informações de seu animal.
Ator:	Act_Cliente
Nível	Objetivo do usuário
Cenários	
Caso normal 1 - O Cliente informa que deseja fazer um agendamento 2 - Há produtos/serviços cadastrados 3 - Há veterinários cadastrados 4 - O Cliente informa os produtos/serviços desejados 5 - Os produtos/serviços existem 6 - O Cliente informa o dia desejado 7 - Há horários disponíveis para esse dia 8 - O Cliente escolhe um horário disponível 9 - O Cliente informa seu nome 10 - O Cliente existe 11 - O Cliente informa o nome de seu Animal 12 - O Animal existe 13 - O Cliente confirma os dados do agendamento 14 - Cadastra o agendamento 15 - Encerra Caso alternativo 2 2.1 - Não há produtos/serviços cadastrados 2.2 - Encerra Caso alternativo 3 3.1 - Não há veterinários cadastrados 3.2 - Encerra Caso alternativo 5 5.1 - Os produtos/serviços desejados não existem 5.2 - Encerra Caso alternativo 7 7.1 - Não há horários disponíveis para esse dia 7.2 - O Cliente escolhe outro dia 7.3 - Retorna ao passo 7 do fluxo normal Caso alternativo 7.2 7.2.1 - O Cliente não deseja escolher outro dia 7.2.2 - Encerra Caso alternativo 10 10.1 - O Cliente não existe 10.2 - Chama o caso de uso "Cadastrar Cliente" 10.3 - Retorna ao passo 11 do fluxo normal Caso alternativo 12 12.1 - O Animal não existe 12.2 - Chama o caso de uso "Cadastrar Animal" 12.3 - Retorna ao passo 13 do fluxo normal	

Tabela 6. 2 - Especificação do caso de uso "Cancelar agendamento".

Número	02
Caso de Uso	Cancelar Agendamento
Escopo	Sistema Clínica Veterinária
Descrição	Caso de uso referente ao cancelamento de agendamentos de consulta ainda não executadas.
Pré-Condição	A consulta não ter sido realizada
Ator:	Act_Cliente
Nível	Objetivo do usuário
Cenários	
Caso normal 1 - O Cliente informa que deseja cancelar um agendamento 2 - O Cliente informa os dados do agendamento 3 - O Agendamento é localizado 4 - O Cliente confirma os dados e o cancelamento 5 - O Agendamento é cancelado 4 - Encerra Caso alternativo 3 3.1 - O Agendamento não é localizado 3.2 - Encerra	

Tabela 6. 3 - Especificação do caso de uso "Consultar agendamento".

Número	03
Caso de Uso	Consultar Agendamento
Escopo	Sistema Clínica Veterinária
Descrição	Caso de uso referente a consulta de registros de agendamentos ainda não executados.
Pré-Condição	Haver veterinários cadastrados
Ator:	Act_Veterinário
Nível	Objetivo do usuário
Cenários:	
Caso normal 1 - O Veterinário inicia o sistema 2 - O sistema seleciona o dia corrente 2 - Mostra os agendamentos de "todos" os veterinários 3 - Encerra Caso alternativo 1 1.1 - O Veterinário seleciona opção do menu 1.2 - O Veterinário escolhe o dia desejado 1.3 - O Veterinário seleciona 1 ou + veterinários 1.4 - Mostra os agendamentos dos veterinários selecionados 1.5 - Encerra	

Nas seções seguintes apresenta-se a aplicação da estratégia de uso da abordagem AQUA, de acordo com as atividades que a compõem.

6.3. Aplicação da atividade IFA

Seguindo o roteiro proposto pela atividade, de posse da especificação dos casos de uso, ou seja, das funcionalidades, é feita a inspeção entre eles.

Iniciando-se a inspeção com a funcionalidade *Fazer Agendamento* (funcionalidade inspecionada), o roteiro é iniciado. A seguir, são descritas as ações executadas para cada etapa do roteiro da atividade IFA, apresentado no Capítulo 5.

Etapa 1

A funcionalidade "Fazer Agendamento" é lida para ser entendida. Nesse exemplo, a especificação do caso de uso "Fazer Agendamento" é lida.

Etapa 2

Dentre todas as funcionalidades, são selecionadas as funcionalidades que possam estar relacionadas à funcionalidade inspecionada. Essa seleção é feita com base nos substantivos candidatos a classe contidos na funcionalidade inspecionada, sendo as funcionalidades relacionadas às que também contém, em sua descrição, um ou mais desses substantivos candidatos a classe.

Na funcionalidade inspecionada, ou seja, na especificação do caso de uso "Fazer Agendamento", são identificados os candidatos a classe: *cliente*, *agendamento*, *consulta*, *produtos/serviços*, *veterinário* e *animal*. Através desses candidatos a classe, as funcionalidades relacionadas são selecionadas. As funcionalidades relacionadas identificadas, por terem algum desses candidatos a classe em sua especificação, são: *Cadastrar Cliente*, *Cadastrar Animal*, *Cadastrar Veterinário*, *Cadastrar Produto*, *Cancelar Agendamento*, *Consultar Agendamento* e *Registrar Consulta*.

Etapa 2.1

Não se aplica, pois foram encontradas funcionalidades relacionadas.

Etapa 2.2

Tendo as funcionalidades relacionadas, as marcações de referência, solicitada no roteiro, são feitas. Dessa forma, por exemplo, na funcionalidade *Fazer Agendamento* são feitas as marcações que referenciam as funcionalidades relacionadas. Essas marcações poderiam ser feitas da seguinte forma: "*Cad. Animal*", "*Cad. Veterinário*", e assim por diante. Ressalta-se que nas funcionalidades relacionadas a essa marcação, referente à funcionalidade inspecionada, também deve ser feita, por exemplo, na funcionalidade *Cadastrar Animal* a marcação referenciando a funcionalidade inspecionada poderia ser feita da seguinte forma: "*Fazer Agendamento*".

Etapa 3 e 3.1

Estando marcadas as funcionalidades envolvidas, é verificado se a funcionalidade inspecionada está consistente com cada uma das funcionalidades relacionadas, verificando-se, por exemplo, se há declarações contraditórias entre conceitos. Após validar a funcionalidade inspecionada com as funcionalidades relacionadas, também é verificada e analisada a relevância de candidatos à classe que foram identificados na funcionalidade inspecionada e não foram referenciados em outras funcionalidades, mesmo com a realização da atividade IFC, problemas dessa natureza ainda podem ocorrer, conforme explicado no Capítulo 5.

Etapa 3.2

Nesse exemplo não seriam encontradas inconsistências. Um possível problema que seria levantado e registrado no Formulário de relato de discrepâncias – IFA seria o fato de que o candidato a classe "Serviço", relevante ao domínio, não foi identificado em nenhuma outra funcionalidade.

Etapa 4

A funcionalidade inspecionada "Fazer Agendamento" recebe um marcação que a identifique como uma funcionalidade já inspecionada.

Etapa 5

O roteiro é executado novamente adotando-se outra funcionalidade que ainda não tenha sido inspecionada, como por exemplo, a funcionalidade "Registrar Consulta".

6.4. Aplicação da atividade ECT

Nesse exemplo, as funcionalidades utilizadas foram os três casos de uso cujas especificações foram mostradas nas Tabelas 6.1, 6.2 e 6.3.

A atividade ECT gera os cartões de teste do sistema, tendo como base a descrição das funcionalidades. Para cada funcionalidade foi obtido seu cartão de teste correspondente. Os casos de teste, registrados no cartão de teste, foram extraídos através das diretrizes constantes no cartão de teste, as quais orientam o testador.

Geração do cartão de teste da funcionalidade *Fazer Agendamento*

Adotando-se os princípios do critério de particionamento por classes de equivalência, a Tabela 6.4 mostra as classes de equivalência derivadas da funcionalidade *Fazer Agendamento*. O critério da análise do valor limite não foi utilizado, pois não foram identificados intervalos na descrição da funcionalidade. Os casos de teste que seriam gerados através do critério "*error guessing*", que tratariam situações do tipo "dados inválidos" e "dados não informados", por exemplo, foram identificados pelo critério do particionamento por classes de equivalência.

Tabela 6. 4 - Classes de equivalência da funcionalidade "Fazer Agendamento".

Id	Classes de equivalência	Id	Classe válida	Id	Classe inválida
1	Agendamento deve ter cliente	CV1	Cliente cadastrado	CI1	Cliente não cadastrado
2	Agendamento deve ter animal	CV2	Animal cadastrado	CI2	Animal não cadastrado
3	Agendamento deve ter produto/serviço	CV3	Produto/serviço cadastrado	CI3	Produto/serviço não cadastrado
4	Agendamento deve ter veterinário	CV4	Veterinário cadastrado	CI4	Veterinário não cadastrado
5	Agendamento deve ter data	CV5	Data válida	CI5	Data inválida
6	Agendamento deve ter horário disponível	CV6	Horário disponível	CI6	Horário inválido
7	Agendamento deve acessar o cadastro do cliente caso o cliente não seja cadastrado	CV7	Cliente não cadastrado	CI7	Cliente cadastrado
8	Agendamento deve acessar o cadastro de Animal caso o animal não seja cadastrado	CV8	Animal não cadastrado	CI8	Animal cadastrado

A Tabela 6.4 mostra as classes de equivalência identificadas na funcionalidade, com as respectivas classes válidas e inválidas. As classes válidas e inválidas receberam um identificador, CV e CI respectivamente, para indicar, nesse exemplo, a origem dos casos de teste contidos no cartão de teste.

Com as classes de equivalência, mostradas na Tabela 6.4, os casos de teste foram gerados de forma que satisfaçam as classes de equivalência válidas e inválidas da funcionalidade *Fazer Agendamento*, sendo que, para classes inválidas, foi gerado um caso de teste para cada classe. O cartão de teste dessa funcionalidade é mostrado através da Tabela 6.5.

Tabela 6. 5 – Cartão de Teste da funcionalidade "Fazer Agendamento".

Cartão de Teste

Funcionalidade: Fazer agendamento

Candidatos à classe envolvidos:
Agendamento, produto, serviço, cliente, animal, consulta, Veterinário

Verificar:

● P.E.
»Restrições
»Lista ou conjunto de opções

● A.V.L.
»Intervalos

● E.G.
»Possíveis erros ou situações tendam a erros (informações obrigatórias, consistências, outros)

Elaborado por: Luiz Cavamura Júnior em 19/nov/2007

CT	Requisitos de Teste	Entrada(s)	Saída
#1	Fazer agendamento Origem: (CV1, CV2, CV3, CV4, CV5, CV6, CI7, CI8)	produtos/serviço data horário cliente animal veterinário	"Consulta agendada"
#2	Fazer agendamento produto/serviço inválido Origem: (CI3)	data horário cliente animal veterinário	"Produto/Serviço não informado"
#3	Fazer agendamento veterinário inválido Origem: (CI4)	produtos/serviço data horário cliente animal	"Não há veterinário cadastrado"
#4	Fazer agendamento sem horário disponível ou inválido Origem: (CI6)	produtos/serviço data horário indisponível cliente animal veterinário	"Horário indisponível"
#5	Fazer agendamento informando data inválida Origem: (CI5)	produtos/serviço data inválida horário cliente animal veterinário	"Data informada inválida"
#6	Fazer agendamento com cliente inválido Origem: (CI1)	produtos/serviço data horário animal cliente inválido veterinário	"Cliente não informado ou inválido"
#7	Fazer agendamento sem cliente cadastrado Origem: (CV7)	produtos/serviço data horário veterinário animal	Possibilitar o cadastro do cliente nesse momento

#8	Fazer agendamento com animal inválido Origem: (CI2)	produtos/serviço data horário cliente animal inválido	"Animal não informado ou inválido"
#9	Fazer agendamento sem animal cadastrado Origem: (CV8)	produtos/serviço data horário cliente	Possibilitar o cadastro do animal nesse momento
• P.E. : Particionamento em classes de equivalência • A.V.L. : Análise do Valor Limite • E.G. : *Error Guessing*			

Geração do cartão de teste da funcionalidade *Cancelar Agendamento*

Adotando-se os princípios do critério de particionamento por classes de equivalência, a Tabela 6.6 mostra as classes de equivalência identificadas na funcionalidade *Cancelar Agendamento*. Assim como ocorrido na geração do cartão de teste da funcionalidade anterior, o critério da análise do valor limite não foi utilizado, pois não foram identificados intervalos na descrição da funcionalidade.

Tabela 6. 6 - Classes de equivalência da funcionalidade "Cancelar Agendamento".

Id	Classes de equivalência	Id	Classe válida	Id	Classe inválida
1	deve ser informado a data do agendamento	CV1	Data válida	CI1	Data inválida
2	Deve ser informado o animal	CV2	Animal cadastrado	CI2	Animal inválido
3	Agendamento deve ser localizado	CV3	Agendamento cadastrado	CI3	Agendamento não cadastrado
4	Deve ter confirmação do cliente	CV4	Confirmação do cliente	CI4	Não confirmação do cliente

Adotando-se os princípios do critério "*error guessing*", a Tabela 6.7 mostra situações que podem levar a erros não identificadas na Tabela 6.6. Essas situações não são citadas na descrição da funcionalidade. Os casos de teste gerados a partir dessas situações complementam os testes a serem executadas para a funcionalidade.

Tabela 6. 7 - Situações da Funcionalidade "Cancelar Agendamento" que tendem a erros que não foram identificados nos critérios anteriores.

Id.	Possíveis situações de erro
EG1	Agendamento já realizado
EG2	Agendamento já cancelado

Através do critério "*error guessing*", dos casos de teste identificados, dois não haviam sido identificados pelos outros critérios utilizados. O cartão de teste da funcionalidade *Cancelar Agendamento*, contendo todos os casos de teste, é mostrado através da Tabela 6.8.

Tabela 6. 8 – Cartão de Teste da funcionalidade "Cancelar Agendamento".

Cartão de Teste

Funcionalidade: Cancelar agendamento

Candidatos à classe envolvidos:
Agendamento, cliente, animal

Verificar:

- P.E.
  - »Restrições
  - »Lista ou conjunto de opções
- A.V.L.
  - »Intervalos
- E.G.
  - »Possíveis erros ou situações tendam a erros (informações obrigatórias, consistências, outros)

Elaborado por: Luiz Cavamura Júnior em 19/nov/2007

CT	Requisitos de Teste	Entrada(s)	Saída
#1	Cancelar agendamento informando animal , data do agendamento e confirmação do cliente Origem: (CV1, CV2, CV3, CV4)	Animal Data do agendamento Confirmação do cliente	"Agendamento cancelado"
#2	Cancelar agendamento informando data inválida Origem: (CI1)	Animal Data do agendamento inválida Confirmação do cliente	"Data inválida"
#3	Cancelar agendamento informando animal inválido Origem: (CI2)	Data do agendamento Confirmação do cliente	"Animal não informado ou inválido"
#4	Cancelar agendamento inexistente Origem: (CI3)	Animal Data do agendamento	"Agendamento não localizado"
#5	Cancelar agendamento sem confirmação do cliente Origem: (CI4)	Animal Data do agendamento Cliente não confirma cancelamento	"Agendamento não cancelado"
#4	Cancelar agendamento já executado Origem: (EG1)	Animal Data do agendamento	"Agendamento já foi realizado"
#5	Cancelar agendamento já cancelado Origem: (EG2)	Animal Data do agendamento	"Agendamento já cancelado"

Geração do cartão de teste da funcionalidade Consultar Agendamento

Adotando-se os princípios do critério de particionamento por classes de equivalência, a Tabela 6.9 mostra as classes de equivalência identificadas na funcionalidade *Consultar Agendamentos*.

Tabela 6. 9 - Classes de equivalência da funcionalidade "Consultar Agendamento".

Id	Classes de equivalência	Id	Classe válida	Id	Classe inválida
1	Deve ser utilizada a data atual do sistema ou uma data especificada	CV1	Data válida	CI1	Data inválida
2	Deve ser selecionado um ou mais veterinários	CV2 CV3	Selecionar um veterinário Selecionar três veterinários	CI2	Não selecionar veterinários

Para gerar as classes válidas e inválidas da classe de equivalência número 2, mesmo sendo especificado somente o limite inferior, foram considerados os princípios do critério da análise do valor limite. Dessa forma, com relação ao limite inferior, foram consideradas a classe válida mínima (selecionar um veterinário) e a classe inválida após essa extremidade (não selecionar veterinário). Como o limite superior não é especificado, considerou-se uma classe válida (selecionar três veterinários).

Adotando-se os princípios do critério "*error guessing*", a Tabela 6.10 mostra uma situação que pode levar a um erro. Essa situação não é citada na descrição da funcionalidade.

Tabela 6. 10 – Situações da "Consultar Agendamentos" que tendem a erros.

Id.	Possíveis situações de erro
EG1	Não existir agendamentos cadastrados para os parâmetros informados (data e veterinários)

O cartão de teste da funcionalidade *Consultar Agendamento*,contendo todos os casos de teste, é mostrado através da Tabela 6.11.

Tabela 6. 11 – Cartão de Teste da funcionalidade “Consultar Agendamento”.

Cartão de Teste

Funcionalidade: Consultar agendamento

Candidatos à classe envolvidos:
Agendamento, veterinário

Verificar:

● P.E.	● A.V.L.	● E.G.
»Restrições »Lista ou conjunto de opções	»Intervalos	»Possíveis erros ou situações tendam a erros (informações obrigatórias, consistências, outros)

Elaborado por: Luiz Cavamura Júnior em 19/nov/2007

CT	Requisitos de Teste	Entrada(s)	Saída
#1	Consultar agendamentos Origem: (CV1, CV2)	Data Selecionar um veterinário	Exibir agendamentos
#2	Consultar agendamentos Origem: (CV1, CV3)	Data válida Selecionar três veterinários	Exibir agendamentos
#3	Consultar agendamentos Informando data inválida Origem: (CI1)	Data inválida Selecionar um veterinário	“Data inválida”
#4	Consultar agendamento não selecionando veterinários Origem: (CI2)	Data válida Não selecionar veterinários	“Veterinário inválido”
#5	Consultar agendamento sem agendamentos cadastrados Origem: (EG1)	Data (sem agendamentos) Selecionar um veterinário	“Não há agendamentos”

Os testes aplicados ao sistema, na ocasião da disciplina, foram obtidos através do padrão “*Application Design is Bounded By Test Design*”, o qual elabora o plano de teste baseado nos casos de uso do sistema. Essa prática obteve doze casos de teste para os três casos de uso considerados nesse exemplo. A atividade ECT, para as mesmas funcionalidades (mesmos casos de uso), formalizou dezenove casos de teste, os quais cobriram os doze casos obtidos pelo padrão e agregou mais sete casos de teste, promovendo um plano de teste mais robusto.

6.5. Aplicação da atividade ETI

Tendo-se os cartões de teste para cada funcionalidade do sistema, e a relação de funcionalidades a serem implementadas em cada iteração, o testador estima o tempo a ser empregado em cada iteração. Através do tempo estimado, é possível verificar se as funcionalidades selecionadas para a iteração estão coerentes com o tempo de duração da iteração, o qual é determinado de acordo com o método utilizado.

Para o projeto do sistema SCV foram definidas duas iterações (*sprints*). Na primeira iteração foram considerados os cadastros das entidades e a regra de negócio *Fazer Agendamento*. Na segunda iteração foram consideradas as regras de negócio *Venda de Produtos*, *Registro de Atendimento* e os *Relatórios*.

Neste exemplo, a atividade ETI foi aplicada nas funcionalidades selecionadas para a primeira iteração, procurando verificar se essas funcionalidades estão coerentes com tempo de duração da iteração, que, no scrum, é de aproximadamente trinta dias.

As funcionalidades envolvidas na primeira iteração são: *Cadastrar Cliente*, *Cadastrar Animal*, *Cadastrar Veterinário*, *Cadastrar Produto* e *Fazer Agendamento*. Tendo-se os cartões de teste, gerados pela atividade ECT, correspondentes a essas funcionalidades, a atividade ETI pode ser iniciada. As seguintes etapas são executadas:

a) Obter o peso total dos atores para o sistema

b) Obter o peso total das funcionalidades da iteração

c) Obter o peso total não ajustado da iteração

d) Obter os fatores de ajuste do sistema

e) Obter o valor de pontos por funcionalidade ajustado

f) Estimar o tempo para implementação da iteração

g) Confrontar o resultado obtido com o tempo de duração da iteração proposto pelo método ágil adotado.

a) Obter o peso total dos atores

A Tabela 6.12 mostra os atores identificados para o sistema com o respectivo tipo e peso. Uma vez classificado cada ator, o peso total dos atores é obtido através da soma do peso total de cada tipo de ator.

Tabela 6. 12 - Atores do sistema.

Ator identificado nas funcionalidades	Tipo	Peso
Cliente	Complexo	3
Veterinário	Complexo	3
Proprietário	Complexo	3
Peso total dos atores		**9**

A técnica de pontos por caso de uso faz a contagem e classificação dos atores apenas uma vez. Como a atividade ETI estima o tempo de cada iteração, o peso total dos atores deve ser dividido pela quantidade de iterações previstas. Para o projeto do sistema foram previstas duas iterações. Dessa forma o peso total dos atores a ser considerado neste exemplo será 4,5.

b) Obter o peso total das funcionalidades da iteração

A Tabela 6.13 mostra a classificação das funcionalidades selecionadas para a primeira iteração. Uma vez classificada cada funcionalidade, o peso total das funcionalidades é obtido através do peso total de cada tipo de funcionalidade.

Tabela 6. 13 - Funcionalidades da iteração.

Funcionalidade	Substantivos candidatos à classe envolvidos	Tipo	Peso
Fazer agendamento	Agendamento Consulta Produto Serviço Cliente Animal Veterinário	Médio	10
Cadastrar Cliente	Cliente	Simples	5
Cadastrar Animal	Animal Cliente Tipo	Simples	5
Cadastrar Produto	Produto Serviço	Simples	5
Cadastrar Veterinário	Veterinário	Simples	5
Peso total das funcionalidades			**30**

c) Obter o valor de pontos por funcionalidade não ajustado

Com o peso total dos atores envolvidos nas funcionalidades e o peso total das funcionalidades, calculam-se o valor de pontos por funcionalidade não ajustado. Esse valor é dado pela soma dos pesos dos atores e das funcionalidades. A Tabela 6.14 mostra esses dados.

Tabela 6. 14 - Pontos não ajustados da iteração.

Peso dos atores	Peso das funcionalidades	Pontos não ajustados da iteração
4,5	30	**34,5**

d) Obter os fatores de ajuste do sistema

A partir da relação de características técnicas, é atribuído o nível de influência de cada característica no projeto. Esse fator é obtido para todo o sistema e não por iteração.

A Tabela 6.15 mostra a relação de características técnicas com os respectivos pesos e níveis de influência para o SCV. O SCV é um sistema bastante simples que foi

desenvolvido para avaliar o uso de padrões organizacionais. Características, tais como concorrência, portabilidade, tempo de resposta e outros, relacionadas na Tabela 6.15, não foram consideradas para o SCV. Das características relacionadas na Tabela 6.15, foram consideradas somente código reusável, facilidade de instalação e facilidade de uso.

Tabela 6. 15 - Características técnicas.

Característica	Peso	Nível de influência	Resultado
Sistema distribuído	2	0	0
Tempo de resposta	2	0	0
Eficiência	1	0	0
Processamento complexo	1	0	0
Código reusável	1	2	2
Facilidade de instalação	0,5	2	1
Facilidade de uso	0,5	3	1,5
Portabilidade	2	0	0
Facilidade de mudança	1	0	0
Concorrência	1	0	0
Recursos de segurança	1	0	0
Acessível por terceiros	1	0	0
Requer treinamento especial	1	0	0
Fator técnico (Tfactor)			**4,5**

O fator de complexibilidade técnica (*TCF*) resultante foi 0,645. Esse valor, conforme mostrado no Capítulo 3, foi calculado utilizando-se o Fator Técnico (*Tfactor*) obtido e fórmula (5): $TCF = 0,6+(0,01 * Tfactor)$.

Calculado o fator de complexibilidade técnica, o fator de complexibilidade do ambiente é calculado utilizando a relação de recursos.

A Tabela 6.16 mostra a relação de recursos, os respectivos pesos e níveis de disponibilidade para o SCV. Esse fator também é obtido uma única vez, não sendo calculado por iteração. A equipe que desenvolveu o SCV era composta por três integrantes. Um dos integrantes assumiu o papel do mestre Scrum, executando atividades de líder. Os outros dois integrantes formavam a equipe Scrum, ou seja, a equipe de desenvolvimento. Todos os integrantes da equipe tinham vivência em ES, sendo que os dois integrantes que formavam a equipe Scrum, além de estudantes de pós-graduação, atuavam como desenvolvedores em empresas privadas. Sendo assim, a maioria dos recursos relacionados na Tabela 6.16 foram considerados no SCV.

Tabela 6. 16 - Recursos do projeto.

Recurso	Peso	Nível de disponibilidade	Resultado
Familiaridade com algum processo formal	1,5	5	7,5
Experiência com a aplicação em desenvolvimento	0,5	0	0
Experiência com Orientação a Objetos	1	5	5
Presença de analista experiente	0,5	5	2,5
Motivação	1	5	5
Requisitos estáveis	2	2	4
Desenvolvedores em meio expediente	-1	2	-2
Linguagem de programação difícil	2	0	0
Fator ambiental (Efactor)			**22**

O Fator de Ambiente (*EF*) resultante foi 0,74. Esse valor, conforme mostrado no Capítulo 3, foi calculado utilizando-se o Fator Ambiental (*Efactor*) obtido e a fórmula (7): $EF = 1,4 + (-0,03 * Efactor)$.

e) Obter o valor de pontos por funcionalidade ajustado

O valor de pontos por funcionalidade ajustado é o produto obtido entre o valor de pontos não ajustado, o fator de complexibilidade técnica e o fator de complexibilidade do ambiente. O valor obtido para a primeira iteração, utilizada como exemplo, foi 16,46.

g) Confrontando os resultados

Para se estimar o tempo de implementação das funcionalidades da iteração aplica-se ao valor de pontos por funcionalidade ajustado a quantidade de horas-homem utilizadas por ponto.

Considerando-se o valor de horas-homem proposto por Karner (1993), 20 horas por ponto de caso de uso, o tempo estimado para a primeira iteração seria de 329,34 horas.

A equipe de desenvolvimento do sistema Clínica Veterinária era composta por dois desenvolvedores, considerando-se que cada desenvolvedor trabalhe 220 horas por mês, o tempo estimado para iteração seria de 0,75 mês para desenvolver somente a primeira iteração. Na ocasião, o tempo total desprendido ao projeto (duas iterações) foi de 190 horas, o que corresponde a 0,43 meses. O mesmo sistema desenvolvido por uma outra equipe consumiu um tempo real total de 320 horas.

A variação ocorrida entre as horas que foram consumidas pelas equipes e a quantidade de horas estimada, foi gerada devido a quantidade de horas-homem por ponto de caso de uso proposta por Karner (1993).

Como citado no Capítulo 5, é necessário utilizar o valor de horas-homem especifico para cada equipe de desenvolvimento. Uma forma de determinar esse valor seria determinar o valor de pontos por caso de uso ajustado de projetos já realizados e dividir esse valor pela quantidade de horas-homem utilizada no projeto.

Com a estimativa de horas a serem consumidas na iteração e a quantidade de membros da equipe de desenvolvimento, é possível verificar se as funcionalidades selecionadas para a iteração estão coerentes com o tempo de duração da iteração proposto por cada método.

Considerando-se os seguintes parâmetros:

- Tempo estimado: 329,34;
- Quantidade de membros da equipe de desenvolvimento: 2;
- Quantidade de horas trabalhadas por dia (cada membro da equipe): 8;

Verifica-se que a iteração seria feita em aproximadamente 20 dias, o que, mesmo considerando dias não trabalhados (finais de semana), seria condizente com o tempo de duração proposto pelo Scrum, que é de aproximadamente 30 dias.

6.6. Considerações finais

Neste capítulo foi apresentado um exemplo de uso da abordagem AQUA, em que suas atividades, com exceção da atividade IFC, foram aplicadas nas funcionalidades de um sistema exemplo, de caráter acadêmico.

Observou-se que a aplicação das atividades IFA, ECT e ETI não demandou grande esforço ao ponto de impactar na agilidade do processo do método Scrum, que foi usado no exemplo. Quanto à atividade IFA, o fato de fazer inspeção nos artefatos que cadastram os requisitos mostrou que alguns erros são identificados, tais como inconsistências e ambigüidades, os quais passariam despercebidos caso a atividade não tivesse sido realizada. Quanto à atividade ECT, sua aplicação provocou a definição de mais casos de teste que o padrão utilizado quando o sistema foi implementado os quais foram extraídos das funcionalidades do sistema, com base nos critérios funcionais. Esses casos de teste foram formalizados e registrados nos cartões de teste, auxiliando o programador no desenvolvimento dos casos de teste automatizados. Quanto à atividade ETI sua aplicação permitiu definir de uma maneira mais sistemática, a estimativa de tempo da iteração.

Com isso, pôde-se verificar que, embora o exemplo utilizado tenha sido um sistema acadêmico, a aplicação dessas atividades é viável e há indícios das contribuições decorrentes de seu uso para a melhoria da qualidade do processo.

CAPÍTULO 7

CONCLUSÕES

Neste trabalho, foi apresentada a abordagem AQUA – Atividades de Qualidade no contexto Ágil. Essa abordagem foi definida para a melhoria da qualidade do processo dos métodos ágeis estudados, de forma que possa ser aplicada independentemente do método utilizado. Para tanto, inicialmente foi realizado um estudo dos métodos ágeis mais citados na literatura, verificando seus princípios, processos, as práticas adotadas e os papéis e responsabilidades desempenhados nesses métodos. Esse estudo possibilitou identificar fases genéricas nos processos desses métodos, permitindo que a definição da proposta pudesse ser aplicada a todos eles.

Como o objetivo era propor algumas atividades de garantia de qualidade no contexto ágil, dada a falta de diretrizes para aplicação desses métodos, foram estudados os conceitos de teste, inspeção e planejamento, já que estas são atividades de garantia de qualidade essenciais para o desenvolvimento de software, independentemente do paradigma adotado.

Também foi abordada a aplicação das atividades de teste e inspeção diretamente no contexto ágil. Dos métodos ágeis estudados, o XP é o que enfatiza e apresenta algumas diretrizes para as atividades de teste; porém, mesmo essas diretrizes não são detalhadas. Dois níveis de teste nesse contexto foram explorados, o teste de unidade e o teste de aceitação. No teste de unidade foi verificada a prática do desenvolvimento orientado por testes, a qual não é considerada uma técnica de teste, mas sim uma sistemática adotada durante o desenvolvimento, para contemplar as atividades de teste do XP (CRISPIN, 2006). No nível de teste de aceitação, verificou-se, no método XP, que o cliente é o responsável por escrever os casos de teste. Na literatura estudada, fica clara a necessidade do apoio de ferramentas automatizadas para a aplicação do teste, seja ele de unidade ou de aceitação. Assim, para ambos os tipos de teste, foi exemplificado, por meio de ferramentas, como esses testes podem ser automatizados. Com relação às atividades de inspeção, foi apresentado o PI-XP (TOMA, 2004), que é

um processo de inspeção composto por três atividades de validação e uma de verificação, apoiadas por técnicas de leitura, definidas especificamente para o processo do método XP.

Com relação às atividades de planejamento de software, foram estudados os principais conceitos e foi apresentada uma técnica que tem sido muito utilizada atualmente para fazer planejamento que é a técnica de Pontos por Casos de Uso. Com base nessa técnica é que foram definidas as diretrizes de planejamento desta proposta.

Assim, a partir da generalização do processo ágil, foi proposta a abordagem AQUA – Atividades de QUalidade no contexto Ágil, composta por quatro atividades que são executadas no âmbito das fases de identificação das funcionalidades do sistema, que é uma fase muito importante para a qualidade do desenvolvimento ágil (ARAUJO, RIBEIRO, 2005). Essas atividades procuram melhorar a qualidade do processo ágil, propondo algumas diretrizes de inspeção, auxílio à prática de testes e ao planejamento, através da estimativa de tempo das iterações.

A abordagem AQUA é composta pelas atividades: *IFC – Inspeção das Funcionalidades junto ao Cliente*, que valida as versões iniciais das funcionalidades com o cliente; *IFA – Inspeção das Funcionalidades nos Artefatos*, que verifica a consistência entre os artefatos que registram a descrição das funcionalidades; *ETC – Elaboração dos Cartões de Teste*, que auxilia as atividades de teste no contexto ágil e a *ETI – Estimativa de Tempo para as Iterações*, que auxilia no planejamento das iterações.

As duas primeiras atividades da abordagem AQUA, ou seja, as atividades *IFC* e *IFA*, atuam de forma a assegurar um melhor entendimento das funcionalidades do sistema e também que essas informações sejam transmitidas corretamente à equipe de desenvolvimento, de forma que as necessidades e expectativas do cliente sejam satisfeitas. A atividade *ECT* auxilia e formaliza o papel do testador no contexto ágil e a forma como os casos de teste utilizados para implementar o sistema devem ser extraídos e registrados. A atividade *ETI*, utiliza a técnica de pontos por casos de uso, possibilitando estimar o tempo de duração da iteração com base nas funcionalidades do sistema. Essa informação auxilia a equipe no planejamento do projeto e verifica se as

funcionalidades selecionadas para cada iteração estão coerentes com o tempo proposto pelo método utilizado.

A abordagem AQUA foi aplicada em um exemplo com o objetivo de ilustrar cada uma das atividades e os resultados fornecem indícios de que ela pode contribuir para melhorar a qualidade do processo. Essa atividade

Dada a relevância do PI-XP a este trabalho, um estudo de caso referente ao PI-XP foi realizado para avaliar suas atividades voltadas ao levantamento de requisitos, avaliando também as técnicas de leitura utilizadas nessas atividades (CAVAMURA et al.,2007). Os resultados obtidos nesse estudo de caso evidenciaram as contribuições do PI-XP, focado nos requisitos, ao processo de desenvolvimento.

7.1. Contribuições

Principais contribuições deste trabalho:

- Formaliza o papel do Testador na equipe de desenvolvimento quando adotados os métodos ágeis e a abordagem aqui proposta.

- Auxilia a atividade de teste, estimulando a utilização de critérios funcionais para a identificação de casos de teste no DOT, induzindo os programadores a desenvolverem testes mais completos e também melhorando a aplicação dos testes de aceitação, podendo esses serem identificados ao gerar os cartões de teste.

- Promove atividades de inspeção no contexto ágil.

- Possibilita a utilização da técnica de pontos por caso de uso para estimar o tempo de duração de cada iteração.

- Fornece formulários para registrar as não-conformidades encontradas durante a aplicação da abordagem AQUA.

7.2. Trabalhos Futuros

A seguir são relacionados trabalhos futuros que envolvem a abordagem AQUA:

- Aplicar o processo proposto em projetos reais que utilizam outros métodos ágeis que foram estudados neste trabalho.

- Avaliar a proposta por meio de estudos experimentais para poder caracterizar sua viabilidade de aplicação prática.

- Automatizar o processo de geração dos cartões de teste da atividade ECT e o cálculo da estimativa de tempo de duração de cada iteração, dando suporte e mais agilidade ao processo ágil.

- Aprimorar a abordagem AQUA, adicionando atividades de garantia de qualidade em outras fases do processo genérico dos métodos ágeis.

REFERÊNCIAS BIBLIOGRÁFICAS

(ABRAHAMSSON et al., 2002) Abrahamsson, P.; Salo, O.; Ronkainen, J.; Warsta, J. **Agile software development methods Review and analysis**. Finlândia: VIT Publications. 2002

(ABRAHAMSSON et al., 2003) Abrahamsson, P.; Warsta, J.; Siponen, M.T.; Ronkainen, J. **New directions on agile methods: a comparative analysis**. In: *Software Engineering*, *2003. Proceedings. 25th International Conference on*. Washington, USA: IEEE Computer Society, 2003. p. 244_254.

(ALLES et al., 2006) Alles, M.; Crosby, D.; Harleton, B.; Pattison, G.; Erickson, C.; Marsiglia, M.; Stienstra, C. **Presenter first: organizing complex gui applications for test-driven development**. In: *Proc. Agile Conference*, *2006*. Minneapolis, MI, USA: IEEE Computer Society, 2006. 10 p.

(AMBLER, 2004) Ambler, S. W. **Modelagem Ágil: Práticas Eficazes para a Programação Extrema e o Processo Unificado**. 1.ed. Porto Alegre: Bookman, 2004. 351 p.

(ANDREA, 2003) Andrea, J. **An agile request for proposal (RFP) process**. In: *Agile Development Conference*, *2003*. Salt Lake City, UT, USA: IEEE Computer Society, 2003. p. 152_161. Disponível em:<http://csdl.computer.org/comp/proceedings/adc/2003/2013/00/20130152abs.htm>.

(ARAUJO, RIBEIRO, 2005) Araujo J.; Ribeiro, J. **Towards an aspect-oriented agile requirements approach**. In: *Principles of Software Evolution 2005, Eighth International Workshop on*. Lisboa, Portugal: IEEE Computer Society, 2005. p. 140_143.

(BACK, 1995) Back, J. **Scrum Software Development Process**. 1995. Disponível em: <http://www.controlchaos.com/old-site/scrumwp.htm>. Acessado em 25 mar. 2006.

(BECK, 2001) Beck, K. **Aim, Fire**. IEEE Software, v. 18, n. 5, p. 87_89, 2001. Disponível em: <http://www.computer.org:80/software/so2001/s5087abs.htm>.

(BECK, 2004) Beck, K. **Programação extrema explicada: acolha as mudanças**. Porto Alegre: Bookman, 2004. 182 p.

(BELLUR, VALLIESWARAN, 2006) Bellur U.; Vallieswaran, V. **On OO design consistency in iterative development**. In: *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*. Las Vegas, NV, USA: IEEE Computer Society, 2006. p. 46_51.

(BOHNET, MESZAROS, 2005) Bohnet R.; Meszaros, G.. **Test-driven porting**. In: *AGILE, 2005*. Denver, CO, USA: IEEE Computer Society, 2005. p. 259_266. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/ADC.2005.46>.

(CANÓS et al., 2003) Canós, J.H.; Letelier, P.; Penadés, M.C. **Metodologias Ágiles em el Desarrollo de Software**. 2003. Disponível em: <http://www.willydev.net/descargas/prev/TodoAgil.Pdf>. Acessado em 08 jan. 2008.

(CAVAMURA et al., 2007) Cavamura, L. J.; Toma, K. M.; Fabbri, S. C. P. F. **Um Processo de Inspeção para o Levantamento de Requisitos no XP**. In: *CLEI 07 Conferência Latinoamericana de Informática, 2007*, San José. Proceedings XXXIII Conferência Latinoamericana de Informática, 2007.

(CHAU et al., 2003) Chau, T.; Maurer, F.; Melnik, G. **Knowledge sharing: Agile methods vs. tayloristic methods**. In: *WETICE, 2003*. Linz, Austria: IEEE Computer Society, 2003. p. 302_307. Disponível em: <http://csdl.computer.org/comp/proceedings/wetice/2003/1963/00/19630302abs.htm>.

(CLIFTON, DUNLAP, 2003) Clifton M; Dunlap, J. **What is DSDM ?** 2003. Disponível em: <http://www.codeproject.com/gen/design/desdm.asp>. Acessado em 02 abr. 2006.

(COCKBURN, 2006) Cockburn, A. **Agile Software Development: The Cooperative Game**. 2.ed. Estados Unidos: Addison Wesley Professional, 2006. 504 p.

(CRESPO et al., 2004) Crespo A. N.; Silva, O. J.; Borges, C. A.; Salviano, C. F.; Teive, M.; Junior, A.; Jino, M.. **Uma metodologia para teste de software no contexto da melhoria de processo**. In: *Simpósio Brasileiro de Qualidade de Software, 2004*. Brasília, DF, Brasil: SBC, 2004. 16 p.

(CRISPIN, 2006) Crispin, L. **Driving software quality: How test-driven development impacts software quality**. IEEE Software, v. 23, n. 6, p. 70_71, 2006. Disponível em: <http://csdl.computer.org/comp/mags/so/2003/03/s3021abs.htm>.

(DSDM) **Dynamic System Development Method** Web Site. Http://www.dsdm.org. Acessado em 25 mar. 2006.

(ERDOGMUS et al., 2005) Erdogmus, H; Morizio,M.; Torchiano, M. **On the efectiveness of the test-first approach to programming**. IEEETSE: IEEE Transactions on Software Engineering, v. 31, 2005.

(FABBRI, 2005) Fabbri, S. C. P. F. **Teste e Validação de Software**. São Carlos: Departamento de Computação-Universidade Federal de São Carlos, 2005.

(FALBO, 2005) Falbo, R. A. **Engenharia de Software: Notas de Aula**. Vitória: Universidade Federal do Espírito Santo, 2005. 145 p.

(FILHO, RIOS, 2003) Filho T. R. M., Rios, E. **Teste de Software**. 1.ed. Rio de Janeiro: Alta Books, 2003. 210 p.

(FOWLER, 2003) Fowler, M. **A Nova Metodologia,** 2003. Disponível em: <http://simplus.com.br/artigos/a-nova-metodologia/N307>. Acessado em 25 mar. 2006.

(HAILPERN, SANTHANAM, 2002) Hailpern B.; Santhanam, P. **Software debugging, testing, and verification**. IBM Systems Journal, v. 41, n. 1, p. 4_12, 2002. Disponível em: <http://researchweb.watson.ibm.com/journal/sj/411/hailpern.html>.

(HETZEL, 1987) Hetzel, W. **Guia Completo ao Teste de Software**. Rio de Janeiro: Campus, 1987. 206 p.

(HIGHSMITH, 2000) Highsmith, J. **Retiring lifecycle dinosaurs: Using adaptive software development to meet the challenges of a high-speed, high-change environment.** Software Testing & Quality Engineering (STQE), p. 22_28, July/August 2000.

(HIGHSMITH, 2002) Highsmith, J. **Agile Software Development Ecosystems**. 2.ed. Estados Unidos: Addison Wesley, 2002. 448 p.

(HOMES, KELLOG, 2006) Holmes A.; Kellogg, M. **Automating functional tests using Selenium**. In: *AGILE, 2006*. Minneapolis, MI, USA: IEEE Computer Society, 2006. p. 270_275. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/AGILE.2006.19>.

(IFTIKHAR, 2004) Iftikhar, B. **Validation, Verification and Debugging - The Process and Techniques**. 2004. Disponível em: <http://www.cs.utexas.edu/users/almstrum/cs370/iftikhar/ValidationVeri_cationandTesting.htm>

(JANERT, 2004) Janert, P. K. **Introducing test-driven software development**. IEEE Software, v. 21, n. 6, p. 100_101, 2004. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MS.2004.42>.

(JANZEN, SAIEDIAN, 2005) Janzen D.; Saiedian, H. **Test-driven development: Concepts, taxonomy, and future direction**. IEEE Computer, v. 38, n. 9, p. 43_50, 2005. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MC.2005.314>.

(JANZEN, SAIEDIAN, 2006) Janzen D.; Saiedian, H. **On the influence of test-driven development on software design**. In: *CSEE&T, 2006*. Lturtle Bay, HI, USA: IEEE Computer Society, 2006. p. 141_148. ISBN 0-7695-2557-1. Disponível em:<http://doi.ieeecomputersociety.org/10.1109/CSEET.2006.25>.

(JONSSON, 2002) Jonsson, P. **Agile development methods and impact analysis**. 2002. Disponível em: <http://www.ipd.bth.se/pjn/web/dl/pdapreportpjn:pdf>. Acessado em 10 abr. 2006.

(JUNIT) **JUnit Resources for a Test Driven Development** Web Site. <http://www.junit.org/>.

(KARNER, 1993) Karner, G. **Use case points - resource estimation for objectory projects objective systems**. Copyright by Rational Software, 1993. 9 p.

(LAITENBERGER, 2001) Laitenberger, O. **A Survey of Software Inspection Technologies**. jul. 09 2001. Disponível em: http://citeseer.ist.psu.edu/469864.html; ftp://cs.pitt.edu/chang/handbook/61b.pdf>.

(LARMAN, BASILI, 2003) Larman C.; Basili, V. R. **Iterative and incremental development: abrief history**. IEEE Computer, V. 36, n. 6, p. 47_56, jul. 2003.

(MAFRA, TRAVASSOS, 2005) Mafra, S. N., Travassos, G. **Técnicas de leitura de software: Uma revisão sistemática**. In: *Simpósio Brasileiro de Qualidade de Software, 2005*. Uberlândia, MG, Brasil: SBC, 2005.

(MAKE, 2001) Make, C. W. **The test-first stoplight.** 2001. Disponível em: <http://www.xp123.com/xplor/xp0101/index.shtml>. Acessado em 01 set. 2006.

(MALDONADO et al., 1998) Maldonado, J. C.; Vicenzi, A. M. R.; Barbosa, E. F. **Aspectos Teóricos e Empíricos de Teste de Cobertura de Software – Notas Didáticas**. São Carlos: Instituto de Ciências Matemáticas de São Carlos, USP, 1998.

(MANIFESTO, 2001) **Manifesto for Agile Software Development**. Web Site Http://agilemanifesto.org.

(MARICK, 2001) Marick, B. **Agile methods and agile testing**. 2001. Disponível em: <http://www.testing.com/agile/agile-testing-essay.html>. Acessado em 01 mai. 2006.

(MARICK, 2003) Marick, B. **Exploration through example**. 2003. Disponível em: <http://www.testing.com/cgi-bin/blog/2003/08/21agile-testing-project-1>. Acessado em 01 mai. 2006.

(MARTIN et al., 2003) Martin A.; Biddle, R.; Noble, J. **How do xp, scrum and asd build the right software?** In: *OOPSLA, 2003*. Anaheim, CA, USA: ACM, 2003. 1 p.

(MARTIN et al., 2004) Martin A.; Biddle, B.; Noble J. **The XP customer role in practice: Three studies**. In: *Agile Development Conference, 2004*. Salt Lake City, UT, USA: IEEE Computer Society, 2004. p. 42_54. Disponível em: <http://csdl.computer.org/comp/proceedings/adc/2004/2248/00/22480042abs.htm>.

(MARTIN, 2005) Martin, R. C. **The test bus imperative: Architectures that support automated acceptance testing**. IEEE Software, v. 22, n. 4, p. 65_67, 2005. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MS.2005.110>.

(MELNIK, MAURER, 2004) Melnik G.; Maurer, F. **Introducing agile methods: Three years of experience**. In: *EUROMICRO, 2004*. Renes, França: IEEE Computer Society, 2004. p. 334_341. ISBN 0-7695-2199-1. Disponível em: http://csdl.computer.org/comp/proceedings/euromicro/2004/2199/00/21990334abs.htm>.

(MILLER, COLLINS, 2001) Miller R. W.; Collins, C. T. **Acceptance Testing**. abr. 10 2001. Disponível em: http://citeseer.ist.psu.edu/527940.html; http://www.xpuniverse.com/2001/pdfs/Testing05.pdf>.

(MOLINARI, 2005) Molinari, L. **Teste de Software: Produzindo Sistemas Melhores e Mais Confiáveis**. 2.ed. São Paulo: Editora Érica Ltda., 2005. 228 p.

(MYERS, 2004) Myers, G. **The Art of Software Testing**. 2.ed. New York, NY: John Wiley & Sons, 2004. 234 p.

(NAWROCKI et al., 2002) Nawrocki, J.; Jasinski, M.; Walter, B.; Wojciechowski, A. **Extreme programming modified: embrace requirements engineering practices**. In: *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*. Essen, Alemanha: IEEE Computer Society, 2002. p. 303-310.

(OLAGUE et al., 2007) Olague, H. M.; Etzkorn, L. H.; Gholston, S.; Quattlebaum, S. **Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes**. IEEE Trans. Software Eng, v. 33, n. 6, p. 402_419, 2007. Disponível em: <http://dx.doi.org/10.1109/TSE.2007.1015>.

(PANCUR et al., 2003) Pancur, M.; Ciglaric, M.; Trampus, M.; Vidmar, T. **Towards empirical evaluation of test-driven development in a university environment**. In: *EUROCON 2003. Computer as a Tool. The IEEE Region 8*. Ljubljana, Slovenia: IEEE, 2003. v. 2, p. 83_86vol.2.

(PETTICHORD, 2002) Pettichord, B. **Agile Testing What is it? Can it work?.** 2002. Disponível em: <www.io.com/~wazmo/papers/agile_testing_20021015.pdf>. Acessado em 01 set. 2006.

(PRESSMAN, 2006) Pressman, R. S. **Engenharia de Software**. 6.ed. São Paulo: McGraw-Hill, 2006. 720 p.

(RASMUSSON, 2003) Rasmusson, J. **Introducing XP into greenfield projects: Lessons learned**. IEEE Software, v. 20, n. 3, p. 21_28, 2003. Disponível em: <http://csdl.computer.org/comp/mags/so/2003/03/s3021abs.htm>

(REICHLMAYR, 2003) Reichlmayr, T. **The agile approach in an undergraduate software engineering course project**. In: *Frontiers in Education, 2003. FIE 2003. 33rd Annual*. Boudlr, CO, USA: IEEE Computer Society, 2003. v. 3, p. S2C_13_18, vol.3.

(ROOIJEN, 2006) Rooijen, H. **Agile software development with scrum:Scrum faq by Ken Schwaber**. 2006. Disponível em: < http://www.scrum-master.com/resources/Conchango%20Scrum%20FAQ%20by%20Ken%20Schwaber.pdf >. Acessado em 01 set. 2006

(RTI, 2002) RTI. **The Economic Impacts of Inadequate Infrastructure for Software Testing.** Gaithersburg, MD, maio 2002. Disponível em: <http://www.nist.gov/director/prog-ofc/report02-3.pdf>.

(SAURER et al., 2006) Saurer G.; Schiefer, J.; Schatten, A. , S. A. **Testing complex business process solutions**. In: *Proc. First International Conference on Availability, Reliability and Security ARES, 2006*. Vienna, Austria: IEEE Computer Society, 2006. 8 p.

(SCHWABER, 2004) Schwaber, K. **Agile Software Development with Scrum.** 1.ed. **Cidade**: Microsoft Press, 2004. p.

(SCHWABER, BEEDLE, 2001) Schwaber K.; Beedle, M. **Agile Software Development with Scrum**. 1.ed.. Cidade: Alan R. Apt, 2001.

(SHULL, 1998) Shull, F. **Developing techniques for using software documents : a series of empirical studies**. Tese (Doutorado) - Dept. of Computer Science, Maryland University, EUA, 1998. Disponível em: <http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/Proposals/Forrest_Shull.pdf>. Acessado em 10 abr. 2006.

(SOARES, 2004) Soares, M. S. **Metodologias Ágeis extremme programming e scrum para o desenvolvimento de software**. 2004. Disponível em: <http://www.inf.ufrgs.br/zirbes/MaterialAula/Engenharia007/Medodologias.pdf>. Acessado em 10 abr. 2006.

(SOMMERVILLE, 2004) Sommerville, I. **Software Engineering**. 7.ed. Estados Unidos: Addison-Wesley, 2004. 784 p.

(STEINDL, 2004) Steindl, C. **Test-driven development**. 2004. Disponível em: <http://agilealliancebeta.org/system/article/_le/1423/_le.pdf>. Acessado em 01 set. 2006.

(TALBY et al., 2006) Talby, D.; Keren, A.; Hazzan, O.; Dubinsky, Y. **Agile Software Testing in a Large-Scale Project**. IEEE Software, v. 23, n. 4, p. 30_37, Jul/Ago 2006. Disponível em: <http://edu.technion.ac.il/Courses/cs_methods/eXtremeProgramming/XP_Papers/Testing_IEEE_Software.pdf>

(TELES, 2004) Teles, V. M. **Extreme Programming**. 1.ed.. São Paulo, SP: Novatec editora Ltda., 2004. 316 p.

(TOMA, 2004) Toma, K. M. **Atividades de Inspeção no Contexto de Métodos Ágeis**. Dissertação (Mestrado em Ciência da Computação)-Centro de Ciências Exatas e de Tecnologia, Universidade Federal de São Carlos, São Carlos, 2004.

(WILLOUGHBY, 2005) Willoughby, Y. M. **Future looks bright for agile coding**. Denver: Computerworld, set. 2005, 1 p.

(YNCHAUSTI, 2001) Ynchausti, R. A. **Integrating Unit Testing Into A Software Development Team's Process**. 2001. Disponível em: <http://citeseer.ist.psu.edu/538842.html;http://www.xp2001.org/conference/papers/Chapter19-Ynchausti.pdf>.

APÊNDICE A

Código-fonte das classes de teste e das classes de negócio da aplicação exemplo mostrada no Capítulo 4.

Classe de negócio *Primo*:

```
package classes;

public class Primo {
	private String entrada;
	public Primo(){}

	public Primo(String entrada) {
		super();
		this.entrada = entrada;
	}

	public int Verifica(){
		try{
			Valida valida = new Valida(entrada);
			if (valida.verifica()){
				int numero = Integer.parseInt(entrada);
				int i;
				int cont = 0;
				for (i=1;i<=numero;i++){
					if ((numero % i) == 0 ){
						cont++;
					}
				}
				if (cont == 2){
					return 1; //numero primo
				}else{
					return 0; //numero nao primo
				}
			}else{
				return 2; //entrada invalida
			}
		}catch(Exception e ){
			return 3; //erro
		}
	}
}
```

Classe de negócio *Valida*:

```
package classes;

public class Valida {
	private String dado;

	public Valida(String dado) {
		super();
		this.dado = dado;
	}

	public boolean verifica(){
		if ((!this.dado.matches("[0-9]*"))){
			return false;
		}else{
			int num = Integer.parseInt(dado);
			if ( (num >=1) && (num <= 100)){
				return true;
			}else{
				return false;
			}
		}
	}
}
```

Classe de Teste *AllTests*:

```
package testes;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

	public static Test suite() {
		TestSuite suite = new TestSuite("Test for testes");
		//$JUnit-BEGIN$
		suite.addTestSuite(TestePrimo.class);
		suite.addTestSuite(TesteValida.class);
		//$JUnit-END$
		return suite;
	}
}
```

Classe de Teste *TestePrimo*:

```
package testes;

import junit.framework.TestCase;
import classes.*;

public class TestePrimo extends TestCase {
      public void testeCaso1() throws Exception {
            String entrada = "3";
            Primo primo = new Primo(entrada);
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(0,primo.Verifica()); //numero nao primo
            assertNotSame(2,primo.Verifica()); //entrada invalida
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso2() throws Exception {
            String entrada = "100";
            Primo primo = new Primo(entrada);
            assertEquals(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(2,primo.Verifica()); //entrada invalida
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso3() throws Exception {
            String entrada = "1";
            Primo primo = new Primo(entrada);
            assertEquals(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(2,primo.Verifica()); //entrada invalida
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso4() throws Exception {
            String entrada = "101";
            Primo primo = new Primo(entrada);
            assertEquals(2,primo.Verifica()); //entrada invalida
            assertNotSame(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso5() throws Exception {
            String entrada = "-1";
            Primo primo = new Primo(entrada);
            assertEquals(2,primo.Verifica()); //entrada invalida
            assertNotSame(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso6() throws Exception {
            String entrada = "a";
            Primo primo = new Primo(entrada);
            assertEquals(2,primo.Verifica()); //entrada invalida
            assertNotSame(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso7() throws Exception {
            String entrada = " ";
            Primo primo = new Primo(entrada);
```

```
            assertEquals(2,primo.Verifica()); //entrada invalida
            assertNotSame(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }
      public void testeCaso8() throws Exception {
            String entrada = "3.1";
            Primo primo = new Primo(entrada);
            assertEquals(2,primo.Verifica()); //entrada invalida
            assertNotSame(0,primo.Verifica()); //numero nao primo
            assertNotSame(1,primo.Verifica()); //numero primo
            assertNotSame(3,primo.Verifica()); //ocorreu erro
      }

}
```

Classe de teste *TesteValida*:

```
package testes;

import classes.Valida;
import junit.framework.TestCase;

public class TesteValida extends TestCase {

      public void testeCaso1() throws Exception {
            String entrada = "3";
            Valida valida = new Valida(entrada);
            assertTrue(valida.verifica());
      }
      public void testeCaso2() throws Exception {
            String entrada = "100";
            Valida valida = new Valida(entrada);
            assertTrue(valida.verifica());
      }
      public void testeCaso3() throws Exception {
            String entrada = "1";
            Valida valida = new Valida(entrada);
            assertTrue(valida.verifica());
      }

      public void testeCaso4() throws Exception {
            String entrada = "101";
            Valida valida = new Valida(entrada);
            assertFalse(valida.verifica());
      }

      public void testeCaso5() throws Exception {
            String entrada = "-1";
            Valida valida = new Valida(entrada);
            assertFalse(valida.verifica());
      }

      public void testeCaso6() throws Exception {
            String entrada = "a";
            Valida valida = new Valida(entrada);
            assertFalse(valida.verifica());
      }
```

```
    public void testeCaso7() throws Exception {
        String entrada = " ";
        Valida valida = new Valida(entrada);
        assertFalse(valida.verifica());
    }

    public void testeCaso8() throws Exception {
        String entrada = "3.1";
        Valida valida = new Valida(entrada);
        assertFalse(valida.verifica());
    }

}
```

APÊNDICE B

Técnica de Leitura Ágil LAg1.

Técnica de Leitura LAg1 – Histórias do Usuário x Cartões de Tarefa (Validação II)

Objetivo: Verificar se os conceitos e funcionalidades que estão descritos nos Cartões de Estória do Usuário foram capturados apropriadamente pelos Cartões de Tarefa.

Entradas para o processo:

1. Um conjunto de Cartões de Estórias do Usuário.
2. Um conjunto de Cartões de Tarefa.

- Para cada Estória do Usuário aplique os passos de 1 a 5:

1	Leia a Estória para poder entendê-la.
2	Separe o(s) respectivo(s) Cartão(ões) de Tarefa correspondente(s) à Estória selecionada (veja pelo número da estória que consta no Cartão de Tarefa).
2.1	Se não encontrar nenhum Cartão de Tarefa, preencha o Relatório de Discrepâncias descrevendo o problema e recomece a Técnica com uma outra Estória.
3	Verifique se os substantivos importantes (candidatos a classes e atributos) da Estória estão registrados em algum Cartão de Tarefa.
3.1	Se existir algum substantivo que não é está registrado no(s) Cartão(ões) de Tarefa, verifique se esse conceito é relevante no contexto da Estória do Usuário. Se necessário, preencha o Relatório de Discrepâncias descrevendo o problema.
4	Verifique se as funcionalidades que compõem uma Estória, as quais estariam representadas por verbos ou descrições de ações, estão registrados em algum Cartão de Tarefa.
4.1	Se existir algum verbo ou descrição de ações que não está registrado no(s) Cartão(ões) de Tarefa, verifique se essa funcionalidade é relevante no contexto da Estória do Usuário. Se necessário, preencha o Relatório de Discrepâncias descrevendo o problema.
5	Verifique se as restrições ou condições da Estória estão registradas em algum Cartão de Tarefa;
5.1	Se existir alguma restrição ou condição da Estória que não está registrada no(s) Cartão(ões) de Tarefa, verifique se esse conceito é relevante no contexto da Estória do Usuário. Se necessário, preencha o Relatório de Discrepâncias descrevendo o problema.

- Se no final sobrar algum Cartão de Tarefa sem a sua Estória correspondente, preencha o Relatório de Discrepâncias.

Técnica de Leitura Ágil LAg2.

Técnica de Leitura LAg2 – Histórias do Usuário x Casos de Teste de Aceitação (Validação II)

Objetivo: Verificar se as restrições e condições que estão descritas no Casos de Teste de Aceitação foram elaboradas apropriadamente de acordo com os Cartões de Estória do Usuário.

Entradas para o processo:

1. Um conjunto de Cartões de Estórias do Usuário.
2. Um conjunto de Casos de Teste de Aceitação.

- Para cada Estória do Usuário aplique os passos de 1 a 5:

1	Leia a Estória para poder entendê-la.
2	Identifique o(s) respectivo(s) Caso(s) de Teste de Aceitação correspondente(s) à Estória selecionada e marque-os com o número da Estória.
2.1	Se não encontrar nenhum Caso de Teste de Aceitação, preencha o Relatório de Discrepâncias descrevendo o problema e recomece a Técnica com uma outra Estória.
3	Verifique se todas as funcionalidades relacionadas ao contexto da Estória estão exploradas em algum Caso de Teste.
3.1	Se existir alguma funcionalidade que não é explorada em nenhum Caso de Teste de Aceitação, verifique se essa funcionalidade é realmente relevante no contexto da Estória do Usuário. Se necessário, preencha o Relatório de Discrepâncias apontando a(s) funcionalidade(s) para a qual não foram criados Casos de Teste.
4	Verifique se todas as restrições relacionadas ao contexto da Estória estão exploradas em algum Caso de Teste.
4.1	Se existir alguma restrição que não é explorada em nenhum Caso de Teste de Aceitação, verifique se essa restrição é realmente relevante no contexto da Estória do Usuário. Se necessário, preencha o Relatório de Discrepâncias apontando a restrição para a qual não foram criados Casos de Teste.
5	Verifique se as condições relacionadas ao contexto da Estória estão exploradas em algum Caso de Teste.
5.1	Se existir alguma condição da Estória que não é representada em nenhum Caso de Teste de Aceitação, verifique se essa condição é realmente relevante no contexto da Estória do Usuário. Se necessário, preencha o Relatório de Discrepâncias apontando a condição para a qual não foram criados Casos de Teste.

- Se no final sobrar algum Caso de Teste de Aceitação sem a sua Estória correspondente, preencha o Relatório de Discrepâncias.

Técnica de Leitura Ágil LAg3.

Técnica de Leitura LAg3 – Cartões de Tarefa X Cartões CRC X Casos de Teste de Unidade (Verificação)

Objetivo: Verificar se os conceitos e funcionalidades que estão descritos nos Cartões de Tarefa foram capturados apropriadamente pelos Cartões CRC e pelos Casos de Teste de Unidade.

Entradas para o processo:

1. Um conjunto de Cartões de Tarefa.
2. Um conjunto de Cartões CRC.
3. Um conjunto de Casos de Teste de Unidade.

- Marque cada um dos Cartões CRC utilizando uma numeração seqüencial.
  - Para cada Cartão CRC numerado, aplique os passos de 1 a 6:

1	Leia o nome da classe, suas responsabilidades e seus colaboradores apontados no cartão CRC.
2	Identifique o(s) Cartão(ões) de Tarefa que faz referencias à classe descrita no topo do Cartão CRC marcando-o(s) com número do Cartão CRC em questão.
2.1	Se não encontrar nenhum Cartão de Tarefa, preencha o Relatório de Discrepâncias descrevendo o problema e recomece a Técnica com uma outro Cartão CRC.
3	Verifique se cada colaborador posssui o seu Cartão CRC correspondente, marcando na frente de seu nome, o número do Cartão CRC correspondente.
3.1	Se existir algum colaborador sem nenhuma numeração, preencha o Relatório de Discrepâncias descrevendo o problema.
4	Verifique se as responsabilidades e seus respectivos colaboradores apontados no Cartão CRC estão descritos em algum Cartão de Tarefa identificado no passo 2..
4.1	Se nem todas as responsabilidades e os colaboradores tiverem sido identificados em algum Cartão de Tarefa, preencha o Relatório de Discrepâncias descrevendo o problema e recomece a Técnica com um outro Cartão CRC.
5	Identifique o(s) respectivo(s) Caso(s) de Teste de Unidade correspondente(s) à classe lida no passo 1, marcando-o(s) com o número do Cartão CRC em questão.
5.1	Se não conseguir identificar nenhum Caso de Teste de Unidade para o Cartão CRC, preencha o Relatório de Discrepâncias e recomece a Técnica com um outro Cartão CRC.
6	Verifique se todas as responsabilidades da classe estão sendo exploradas em algum Caso de Teste de Unidade identificado no passo 5. para cada responsabilidade encontrada, marque-a com um "X".
6.1	Se existir uma responsabilidade que não esteja marcada com "X", preencha o Relatório de Discrepâncias descrevendo o problema.

- Se no final sobrar algum Cartão de Tarefa sem nenhuma numeração associada a algum Cartão CRC, preencha o Relatório de Discrepâncias.
- Se ao final sobrar algum Caso de Teste de Unidade sem nenhuma numeração associada ou algum Cartão CRC, preencha o Relatório de Discrepâncias.

APÊNDICE C

Formulário de Relato de Ocorrências

IFC - Inspeção das Funcionalidades junto ao Cliente

PROJETO	
CLIENTE	

Tipos de ocorrência:	
ME	Melhoria (novas funcionalidades)
NC	Não-Conformidade (Inconsistências)
IO	Informação Omitida (Não informado nas versões iniciais da func.)
NS	Não Solucionado (Depende de outro cliente representativo)
OU	Outro

Prioridade
A=Alta
M=Média
B=Baixa

Cliente sênior:	
Cliente respresentativo:	
Testador:	
Analista:	

Data:	
Início:	
Término:	

#Id	Tipo Ocorrência	Prioridade	Id. func.	descriçao	Disposição
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Formulário de Relato de Discrepâncias

IFA - Inspeção das Funcionalidades nos Artefatos

PROJETO	
CLIENTE	

Tipos de Discrepância	
I1	Inconsistência (conflitos de termo/desc.)
A	Ambiguidade (mútiplas definições)
I2	Incoerência (fato não verdadeiro de acordo com outra func. já inspecionada)
O	Outro

Prioridade
A=Alta
M=Média
B=Baixa

Testador:	
Analista:	

Data:	
Início:	
Término:	

#Id	Tipo Discrepância	Prioridade	Id. func.	descriçao (indicar as relações)	Disposição
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Formulário de Relato de Ocorrências

ECT - Elaboração dos Cartões de Teste

PROJETO	
CLIENTE	

Observações Gerais:

Prioridade
A=Alta
M=Média
B=Baixa

Responsável	

Data:	
Início:	
Término:	

#Id	Prioridade	Id. func.	descriçao do problema	Disposição
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				