Definição e gerenciamento de métricas de teste no contexto de métodos ágeis

André Abe Vicente

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Definição e gerenciamento de métricas de teste no contexto de métodos ágeis

André Abe Vicente

Orientador: ***Prof. Dr. Márcio Eduardo Delamaro***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação — ICMC/USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional.

USP - São Carlos
Março/2010

Aos meus pais Dorival e Meire minha imensa gratidão por toda dedicação, apoio e valores de humildade, caráter e perseverança por vocês transmitidos.

Ao meu avô Manuel Vicente, avó Maria Calizotti, oditchan Tetsuzo Abe "in memorian" e obatchan Okamoto Tyoko pelo grande exemplo de vida.

Agradecimentos

Agradeço à Deus, por me iluminar nos momentos difíceis e por me proteger nas viagens até Cascavel-PR e São Paulo-SP.

À toda minha família, principalmente meus pais Dorival e Meire que sempre me incentivaram e apoiaram de uma forma única os meus estudos e conquistas que obtive até hoje em minha vida. Minhas irmãzinhas Carla (*Carlota*) e Mariana (*Mari*) que mesmo à distância souberam transmitir muito amor e carinho quando eu precisava. Me faltam palavras para descrever todo o meu amor por vocês! Também não posso deixar de agradecer aos meus avós Manoel, Maria, Tetsuzo (*in memorian*) e Tyoko, pessoas sábias, que assim como meus pais, sempre foram e serão fontes de inspiração em minha vida. À toda família Vicente e família Abe pelo constante incentivo e principalmente ao meu tio Rubens Abe por sempre me acolher em sua casa em minhas visitas à São Paulo durante o mestrado.

Ao Prof. Dr. Márcio Eduardo Delamaro, pela amizade, orientação, incentivo à pesquisa e principalmente pela confiança em mim depositada.

À outros professores do LabES que de diversas formas contribuíram com esse trabalho, com a minha formação acadêmica e também pessoal: Prof. Dr. José Carlos Maldonado pela orientação em grande parte deste trabalho, a Profa. Dra. Elisa Yumi Nakagawa e Prof. Dr. Adenilso da Silva Simão pelo importante apoio no início de minhas pesquisas.

À Profa. Dra. Ellen Francine Barbosa e Profa. Dra. Sandra Fabbri pelas valiosas sugestões dadas a este trabalho durante o exame de qualificação.

Ao André Endo e Marco Graciotto pela colaboração com a revisão do texto, sugestões e principalmente com a condução do estudo de caso. Sem dúvida a ajuda de vocês foi muito importante para os resultados desse trabalho! Agradeço também, os orientandos(a) do Prof. Fabio Kon (IME/USP): Claudia Melo, Hugo Corbucci e Paulo Meirelles que forneceram informações a respeito dos projetos que foram utilizados neste trabalho.

Aos meus amigos do LabES, pelos churrascos, conversas durante o café e "2º tempo" na cantina: André Domingues, Alessandra Chan, Bruno Cafeo, Carlos Pereira Jr., Diogo Nascimento, Draylson Souza (*Drauzio*), Edson de Oliveira Jr., Erika Höhn, Katia Felizardo, Kenji Toyohara, Lúcio Felipe de Melo, Marcella Letícia, Marcelo Eller, Maria Adelina, Mário Machado (*Casão*), Otávio Lemos, Paula Donegan, Paulo Nardi, Rafael Messias, Rodolfo Barbeiro, Rodrigo Gondim, Vânia Neves e Viviane Malheiros. Nesses agradecimentos também incluo amigos de mestrado do LCAD, LABIC, LaSDPC, Engenharia de Produção e outros amigos da lista icmc-pos-alunos.

Aos companheiros(as) de balada, TUSCAs, futebol de quarta-feira no Trem Bom, ou qualquer outra espécie de festa, churrasco ou confraternização, obrigado por fazerem com que a minha permanência em São Carlos se tornasse mais alegre e prazerosa:

Adriano Bezerra (*DJ Alemão*), Alexandre Huff (*Rumos*), Anderson Menezes, André Endo (*Endo*), Aretha Alencar, Bruno Nogueira, Camila Leal (*Canjiquinha*), Cássio Prazeres, David Neto (*Peixe*), Danilo Coimbra (*Danilão*), Jorge Cuttigi (*Piu*) Fabiano Ferrari (*Fabis*), Flávio Dusse (*Sorin*), Lucas Bueno (*Cabeça*), Marco Aurélio Graciotto (*Marcão*), Marllos Prado (*Tiozão*), Maycon Leone, Merley da Silva Conrado (*Miel*), Pablo Jaskowiak, Paulo Henrique Ribeiro (*Nerso*), Paulo Gabriel Queiroz (*Gambi / Ceará / Abadá*), Rafael Oliveira (*Frotinha*), Renato Resina (*Resinex*), Rodrigo Fraxino (*Ronaldo*), Sandro Bianchini (*KLB*), Tácito Tiburtino, Vinicius Durelli (*Fofo / Lenon*) e Vanessa Borges (*Magrela*).

Não poderia deixar de mandar um grande salve à pensão da Beth: Elizabeth Alexandre (*Beth*), Alexandre Martins (*Xandão*), Claudão, Getúlio Capellari (*Vargas*), Murilo Arruda, Reinaldo (*Nadinho*), Renan Pastore (*Calheiros*), além de diversos amigos(as) da Beth que passaram por essa famosa pensão. Juntamente com alguns amigos de São Carlos, vocês se tornaram a minha segunda família, agradeço a todos pelo apoio e principalmente pela companhia agradável durante esses anos.

À todos meus amigos(as) de Cascavel-PR, que me fazem sentir muitas saudades dessa cidade maravilhosa! Aproveito esse momento para estender este agradecimento ao meu grupo de amigos chamados de H.G.K e seus familiares. Me sinto honrado e orgulhoso de ter amigos que me proporcionam grandes momentos de felicidade, discussão e incentivo a novas conquistas. Mesmo distantes, felizmente foi possível manter um FORTE vínculo de amizade. Também agradeço a algumas amigas que podem ser consideradas muito especiais em minha vida, são elas: Karen Stadler (*Kahkahzinha*), Paula Fernandes (*Paulinha*), Kyhara Pessoa e Thalita Marques (*Thai*).

Aos professores (Prof. Dr. Victor Santander, Prof. Msc. Ivonei Freitas e Prof. Msc. Carlos J. M. Olguín) e amigos de graduação da UNIOESTE que se fizeram presentes mesmo durante o mestrado, que contribuíram com a minha formação e sempre me incentivaram a continuidade da minha vida acadêmica.

À todos funcionários do ICMC que colaboraram de alguma forma com este trabalho. Estendo este agradecimento especialmente as secretarias da pós-graduação que sempre me atenderam de uma forma bastante prestativa e também a equipe de guardas do CISC que com toda a simpatia me saudavam na entrada e saída do LabES, fazendo com que diariamente o meu trabalho começasse e também terminasse de uma forma mais agradável.

E finalmente, agradeço ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), pelo apoio financeiro.

Resumo

Métodos ágeis são técnicas adequadas para o desenvolvimento de software sujeito a mudanças constantes. Essas mudanças não devem afetar o cronograma, orçamento do projeto e devem assegurar o atendimento às necessidades do cliente. Diversos valores, princípios e boas práticas de desenvolvimento e de condução de projeto são aplicados em projetos ágeis com esse objetivo. Algumas dessas práticas são relacionadas a atividade de teste de software. Este trabalho teve como objetivo caracterizar a atividade de teste de software aplicada dentro de métodos de desenvolvimento ágil, buscando eliminar aspectos de teste não produtivos, identificando boas práticas e, principalmente, criando formas de acompanhar e melhorar continuamente a condução da atividade de teste. A partir da caracterização da atividade foi proposta a adoção de um conjunto de métricas para facilitar o seu acompanhamento e melhoria constante da mesma. Algumas dessas métricas de acompanhamento de testes foram implementadas na ferramenta *Agile Testing Metrics Management* (ATMM). O objetivo principal da ferramenta é gerenciar as iterações de desenvolvimento do projeto ágil e, também, exibir a evolução das métricas relacionadas ao código que está sendo testado e aos casos de teste desenvolvidos utilizando a ferramenta JUnit. Para validar a ferramenta e as métricas foram conduzidos estudos de casos com dois projetos de software de domínios diferentes que utilizaram métodos ágeis e testes de unidade.

Palavras-chave: Métodos ágeis, métodos ágeis de desenvolvimento, teste de software, métricas de software.

Abstract

Agile methods are appropriate techniques for software development subject to constant changes. These changes should not affect the project schedule, budget and must ensure meeting the clients needs. Several values, principles and practices of project development and driving are applied in agile projects with this goal. Some of these practices are related to software testing activity. This study aimed at characterizing the software testing activity applied to agile development methods, trying to eliminate unproductive testing aspects, identifying good practices and especially creating ways of tracking and continuously improve the test activity. From this activity characterization, it was proposed an adoption of metrics set to facilitate the monitoring and constant improvement of the activity. Some of these testing tracking metrics were implemented in the *Agile Testing Metrics Management Tool* (ATMM). The main goal of this tool is to manage the iterations of agile project development and, also show the metrics evolutions regarding the code that have been tested and the test cases developed using JUnit. The tool and metrics were validated by case studies that were conducted with two software projects of different domains which used agile methods and unit testing.

Keywords: Agile methods, agile software development methods, software testing, software metrics.

Sumário

Resumo **v**

Abstract **vii**

Lista de Figuras **xiii**

Lista de Tabelas **xv**

Lista de Abreviaturas **xvii**

1 Introdução **1**
1.1 Motivação . . . 3
1.2 Objetivos . . . 5
1.3 Organização . . . 5

2 Métodos Ágeis de Desenvolvimento **7**
2.1 Métodos Ágeis de Desenvolvimento . . . 8
2.2 *eXtreme Programming* (XP) . . . 12
2.2.1 Ciclo de Desenvolvimento do XP . . . 13
2.2.2 Valores, Princípios e Práticas do XP . . . 15
2.2.2.1 Valores do XP . . . 16
2.2.2.2 Princípios do XP . . . 17
2.2.2.3 Práticas do XP . . . 17
2.2.3 Papéis da Equipe de XP . . . 23
2.3 Scrum . . . 24
2.3.1 Ciclo de Desenvolvimento e Práticas do Scrum . . . 25
2.3.2 Papéis do Scrum . . . 27
2.4 *Lean Software Development* . . . 29
2.5 Outros Métodos Ágeis . . . 30
2.6 Considerações Finais . . . 33

3 Teste de Software **35**
3.1 Fundamentos do Teste de Software Tradicional . . . 36
3.1.1 Fases de Teste de Software . . . 37
3.1.2 Técnicas e Critérios de Teste . . . 38
3.2 Teste de Software em Métodos Ágeis . . . 43

3.2.1 Test Driven Development (TDD) 46
3.2.2 Testes de Aceitação (*Business Testing*) 49
3.2.3 Outras Práticas de Teste Ágil 51
3.2.4 Revisão Sistemática sobre Testes Ágeis 53
3.3 Ferramentas de Teste utilizadas em Projetos Ágeis 55
3.4 Considerações Finais 59

4 Métricas de Software 61
4.1 Conceitos Gerais 62
4.2 Abordagens para escolha de Métricas 64
4.2.1 Abordagem GQM (*Goal Question Metric*) 64
4.2.2 Abordagem Lean 64
4.3 Métricas de acompanhamento para Testes Ágeis 65
4.4 Métricas de Teste Ágil 66
4.4.1 Métricas de apoio a Testes de Unidade 68
4.4.2 Métricas de Teste de Aceitação (*Business Testing*) 72
4.4.3 Métricas de Teste Gerais 76
4.4.4 Área de Trabalho Informativa e Retrospectivas 78
4.5 Ferramenta ATMM (*Agile Testing Metrics Management Tool*) 80
4.5.1 Arquitetura Geral e Funcionamento da Ferramenta 80
4.6 Considerações Finais 84

5 Estudo de Caso 87
5.1 Método e Métricas Avaliadas 88
5.2 Caracterização dos Projetos Ágeis 89
5.2.1 Kalibro 91
5.2.2 Archimedes 95
5.3 Análise dos Resultados 98
5.3.1 Resultados do questionário de informações gerais do projeto 98
5.3.2 Análise das métricas de acompanhamento de teste de software 101
5.3.2.1 Evolução da quantidade de código e testes produzidos 101
5.3.2.2 Evolução do teste de unidade 106
5.3.2.3 Evolução da Cobertura de Código 108
5.4 Discussão e Conclusões 110

6 Conclusões e Trabalhos Futuros 115
6.1 Contribuições 117
6.2 Dificuldades e Limitações 118
6.3 Trabalhos Futuros 118
6.4 Publicações Esperadas 119

Referências Bibliográficas 121

A Questionário Informações Gerais (Projetos Ágeis) 131
A.1 Informações Gerais do Projeto e Contato 131
A.2 Fator Ergonômico 131
A.3 Fatores Tecnológicos 132
A.4 Fatores Geográficos 132
A.5 Fatores Sociológicos 132

A.6 Fatores Específicos do Projeto 133
A.7 Atividade de Teste de Software 133
A.8 Observações Finais / Sugestões 134

B Guia de Execução de *scripts JaBUTi* 135

Lista de Figuras

2.1 Ciclo de vida do eXtreme Programming 13
2.2 Ambiente XP - Área de Trabalho Informativa e Programação em pares 20
2.3 Ciclo de vida do processo Scrum 25

3.1 Processo de Teste de Software 36
3.2 Estratégia de Teste em Projetos Ágeis 44
3.3 Ciclo do Test Driven Development (TDD) 48
3.4 *JaBUTi*- Cobertura de casos de teste por critério 57
3.5 Tela da Ferramenta *JaBUTi* 58
3.6 Exemplo de Tabela Fit da Ferramenta FitNesse 59

4.1 Gráfico de evolução do número de testes de aceitação criados por iteração e informações sobre cobertura de código. 79
4.2 Arquitetura Geral da Ferramenta ATMM 82
4.3 Ferramenta ATMM - Criação e Informações gerais do Projeto (Tela Principal) 82
4.4 Ferramenta ATMM - Informações sobre cada Iteração de desenvolvimento e testes 83
4.5 Ferramenta ATMM - Exibição comparativa das métricas 84
4.6 Ferramenta ATMM - Exibição comparativa das métricas 85

5.1 Arquitetura geral do projeto Mezuro 91
5.2 Kalibro - Ferramenta de configuração e interpretação de métricas de código-fonte 92
5.3 Archimedes - Ferramenta CAD (*Computer Aided Design*) 95
5.4 Gráfico de valores - Aderência e Maturidade da atividade de teste no projeto Kalibro e Archimedes 100
5.5 Kalibro - Quantidade de linhas de código (LOC) e LOC / Classes 102
5.6 Kalibro - Fator de Teste (LOC SUT / LOC T) 103
5.7 Archimedes - Fator de Teste (LOC SUT / LOC T) 103
5.8 Archimedes - Quantidade de linhas de código (LOC) e LOC / Classes 104
5.9 Kalibro - Total de Casos de Teste e Assertivas (JUnit) 107
5.10 Archimedes - Total de Casos de Teste e Assertivas (JUnit) 107
5.11 Kalibro - Cobertura de Testes utilizando Critérios de Fluxo de Controle e Fluxo de Dados 109

Lista de Tabelas

2.1 Principais diferenças entre o desenvolvimento tradicional e o desenvolvimento ágil . . . 10
2.2 Comparativo entre Métodos Ágeis (Características Gerais) . . . 11

3.1 Comparativo sobre a atividade de Teste em Métodos Ágeis . . . 45
3.2 Comparativo entre Métodos Ágeis (Retrospectiva, Melhoria e Técnicas e Práticas de VV&T) . . . 47
3.3 Exemplo - Teste de Aceitação para História . . . 50
3.4 Tipos de Estudos Experimentais - TDD . . . 53
3.5 Resultados dos Estudos Experimentais - TDD . . . 54
3.6 Ferramentas de Teste de Cobertura para Linguagem Java . . . 56

4.1 Lista de verificação para o planejamento de uma métrica ágil . . . 67
4.2 Origem das métricas de acompanhamento de teste adotadas no trabalho . . . 67
4.3 Métricas de Acompanhamento de Testes implementada pela ATMM . . . 81

5.2 Kalibro - Fator ergonômico, fatores geográficos, específicos e sociológicos do projeto . . . 91
5.1 Kalibro - Informações gerais e fatores tecnológicos do projeto . . . 93
5.3 Kalibro - Práticas ágeis, práticas de teste do projeto e aderência a atividade de testes . . . 94
5.4 Kalibro - Maturidade da atividade de teste de software . . . 94
5.5 Archimedes - Informações gerais e fatores tecnológicos do projeto . . . 96
5.6 Archimedes - Fator ergonômico, fatores geográficos, específicos e sociológicos do projeto . . . 96
5.7 Archimedes - Fatores específicos, práticas ágeis e práticas de teste do projeto . 97
5.8 Archimedes - Aderência e Maturidade da atividade de teste de software . . . 97
5.9 Kalibro - Níveis de aderência e maturidade da atividade de testes . . . 99
5.10 Archimedes - Níveis de aderência e maturidade da atividade de testes . . . 99
5.11 Kalibro - Métricas de Acompanhamento de Testes (Quantidade de Código e Testes) . . . 102
5.12 Archimedes - Métricas de Acompanhamento de Testes (Quantidade de Código e Testes) . . . 103
5.13 Kalibro - Qualidade do código-fonte (AMLOC, MMLOC, LCOM4, ACC, COF) 105

5.14 Archimedes - Qualidade do código-fonte (AMLOC, MMLOC, LCOM4, ACC, COF) 106
5.15 Kalibro - Métricas de Acompanhamento de Testes (Testes de Unidade) 106
5.16 Archimedes - Métricas de Acompanhamento de Testes (Testes de Unidade) . . 108
5.17 Kalibro - Análise de Cobertura dos Testes (utilizando critérios estruturais) . . 109
5.18 Archimedes - Análise de Cobertura dos Testes (utilizando critérios estruturais) 110

Lista de Abreviaturas

ACC	*Afferent Connections per Class*
ACM	*Association for Computing Machinery*
AgilCoop	Cooperativa de Desenvolvimento Ágil de Software
AMLOC	*Average Lines per Method*
ASD	*Adaptative Software Development*
ATDD	*Acceptance Test Driven Development*
ATMM	*Agile Testing Metrics Management Tool*
BDD	*Behavior-Driven Development*
CAD	*Computer Aided Design*
CFG	Grafo de Fluxo de Controle
CMMI	*Capability Maturity Model Integration*
CBO	*Coupling Between Objects*
C3	*Chrysler Comprehensive Compensation System*
DBC	Desenvolvimento baseado em Componentes
DSDM	*Dynamic System Development Method*
ET	*Exploratory Testing*
FDD	*Feature Driven Development*
FIT	*Framework for Integrated Testing*
GQM	*Goal Question Metric*
GQS	Garantia de Qualidade de Software
GUI	*Graphic User Interface Graphic User Interface*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronic Engineers*
JaBUTi	*Java Bytecode Understanding and Testing*
JAD	*Joint Application Design*
LCOM	*Lack of Cohesion in Methods*
LEAN	*Lean Software Development*
MMLOC	*Max Method LOC*
MoSCoW	*Musthave, Should have, Could have e Want to have*
OO	Orientação a Objetos
PDCA	*Plan-Do-Check-Act*
QualiPSo	*Quality Platform for Open Source Software*
RAD	*Rapid Application Development*
ROI	*Return on Investment*

RTF	*Running Tested Features*
SEI	*Software Engineering Institute*
XML	*eXtensible Markup Language*
XP-EF	*Extreme Programming Framework*
SWEBOK	*Software Engineering Body of Knowledge*
SUT	*System Under Test*
TDD	*Test Driven Development*
VV&T	Verificação Validação e Teste
XP	eXtreme Programming

Capítulo

1

Introdução

Para enfrentar problemas com prazos e complexidade de métodos tradicionais da engenharia de software, diversos métodos ágeis de desenvolvimento estão sendo utilizados em projetos de software. Estes métodos possuem como principal objetivo a satisfação do cliente, preocupando-se com a entrega incremental de software desde as etapas iniciais de desenvolvimento, produtos de trabalho de engenharia de software minimizados e simplicidade global no desenvolvimento (Abrahamsson et al., 2002).

Métodos ágeis foram desenvolvidos para beneficiarem a entrega rápida de código que agregue valor ao cliente por meio do desenvolvimento em pequenos ciclos. Para atingir esse objetivo, esses métodos são focados na contínua interação entre desenvolvedores e clientes, que garantem que o software atenda as necessidades de mudança dos requisitos do cliente (Paetsch, 2003). Os métodos ágeis mudam o foco de artefatos complexos de projeto, fortemente utilizados em métodos tradicionais, para técnicas focadas no desenvolvimento de código-fonte e testes. A prototipação ágil também ajuda a acelerar a velocidade de desenvolvimento, reduzindo o excesso de planejamento e a documentação (Ambler e Jeffries, 2002).

Todo esse dinamismo dos métodos ágeis tem provocado um grande impacto na forma de se conduzir um projeto de software fortemente sensível a mudanças. Uma grande diversidade de métodos ágeis tem sido utilizada (Abrahamsson et al., 2002; Highsmith, 2002). Como exemplos mais conhecidos tem-se o XP (*eXtreme Programming*) (Beck, 2000; Beck e Andres, 2004), o *Scrum* (Schwaber, 1995, 2004; Schwaber e Beedle, 2001), a família *Crystal* (Cockburn, 2002, 2006), o FDD (*Feature Driven Development*) (Palmer e Felsing, 2001), o ASD (*Adaptative Software Development*) (Highsmith, 2002), o DSDM (*Dynamic System Develop-*

ment Method) (DSDM, 2009; Stapleton, 2003; Voigt, 2004) e o *Lean Software Development* (Poppendieck e Poppendieck, 2003, 2006). Esses métodos seguem princípios semelhantes, mas o que os diferencia são as suas práticas e a forma de condução do processo de desenvolvimento. O XP é o método mais conhecido, além de ser o método mais avaliado por estudos acadêmicos (Dingsøyr et al., 2008). Ele propõe um conjunto de valores, princípios e práticas em um cenário no qual os requisitos são vagos e mudam constantemente. Nesse contexto o objetivo do método é a excelência no desenvolvimento de software, visando baixo custo, poucos defeitos, alta produtividade e alto retorno de investimento. Outro método que vem se destacando é o Scrum, que é o mais utilizado na indústria de software atualmente (Version One, 2010), sendo mais focado em aspectos de gerenciamento de projeto.

Para satisfazer o cliente por meio da entrega rápida e contínua de software de qualidade (Huo et al., 2004) são utilizadas diversas práticas que serão detalhadas no Capítulo 2. Como exemplo pode-se citar a presença do cliente para ajudar na correção e refinamento de requisitos, a programação em pares utilizada para melhorar a qualidade do desenvolvimento e promover a diminuição de defeitos, e a refatoração (Fowler, 1999) que otimiza o código já existente. A agilidade descrita anteriormente, traz consigo uma forte preocupação com a melhoria constante do processo e do produto durante todo o ciclo de desenvolvimento do software, no qual o progresso do projeto é avaliado diariamente. Em projetos que utilizam métodos ágeis, a atividade de teste de software vem sendo considerada uma atividade primordial, com o objetivo de evitar que a qualidade do produto e a condução do projeto não sejam afetados por processos menos formais de documentação e projeto em relação aos métodos tradicionais (Simons, 2005).

A importância da atividade de teste em métodos ágeis pode ser constatada no método XP, que considera a atividade tão importante quanto a atividade de programação (Beck e Andres, 2004). No XP todo pedaço de código tem um conjunto de testes de unidade automatizados, que deve ser integrado ao repositório de código-fonte. Os resultados dos testes servem como uma forma de *feedback* instantâneo, no qual o desenvolvedor pode detectar em pouco tempo se o método desenvolvido ainda precisa ser modificado ou refatorado. O código é considerado completo apenas se passar por todos testes de unidade. Além disso, no fim de cada iteração, todos os testes de aceitação (*business testing)* que foram criados durante a fase de planejamento serão executados por usuários e clientes. Esses testes incluem os testes de iterações prévias e aqueles da última iteração, para determinar se as novas funcionalidades são aceitáveis e prevenir que novas mudanças causem efeitos colaterais em funcionalidades que estavam funcionando até o momento.

Entre as práticas de teste utilizadas em métodos ágeis pode-se citar os testes de unidade utilizando *test driven development* (TDD) (Beck, 2002), testes de integração contínuos, testes de aceitação com o cliente e testes de regressão associados a prática de refatoração. Para complementar essas práticas a equipe pode utilizar testes exploratórios, teste da interface gráfica (GUI) e teste de requisitos não-funcionais que podem envolver por exemplo, requi-

sitos de desempenho, carga ou *stress*. Todas essas práticas de testes devem ser executadas preferencialmente de forma automatizada, buscando a agilidade no processo de testes.

1.1 Motivação

Em 2002 Abrahamsson et al. (2002), destacaram em um de seus trabalhos a pequena quantidade de pesquisas acadêmicas na área, sendo que as principais publicações eram escritas apenas por poucos praticantes e consultores de métodos ágeis. Desde então esse cenário mudou e hoje pode-se observar uma grande sinergia entre a academia e a indústria na área de métodos ágeis, que pode ser constatada em eventos de larga importância como o *International Conference on XP* (`http://xp2010.org`), *Agile Conference* (`http://agile2010.agilealliance.org`) e mais recentemente em eventos nacionais e latino-americanos, como o *Ágiles* (`http://agiles2010.agiles.org`) e o *Agile Brazil* (`http://www.agilebrazil.com`).

Em relação à maturidade das pesquisas da área de métodos ágeis, Dingsøyr et al. (2008) afirmam que há um nível intermediário nas pesquisas em relação ao método XP, a programação em pares e TDD. Tópicos como: modelagem ágil, DSDM, Lean, Scrum e a educação de engenharia de software com métodos ágeis ainda são nascentes. Nesse contexto, Dingsøyr et al. (2008) sugerem que os pesquisadores pensem em dar prioridade a estudos que têm tido pouca atenção em pesquisas como o método Scrum. Além disso, os estudos devem ser conduzidos com organizações experientes na aplicação de métodos ágeis e devem explorar a utilização de práticas ágeis em diversos contextos, procurando encontrar como aplicar princípios ágeis em situações reais e aplicar práticas e princípios do desenvolvimento ágil em áreas já estabelecidas como engenharia de requisitos, teste de software ou arquitetura de software.

A utilização de métodos ágeis de desenvolvimento tem crescido sensivelmente. Segundo Sato (2007) esses métodos vêm sendo adotados em diversos contextos, em pequenas, médias e grandes empresas e até agências governamentais e universidades (Begel e Nagappan, 2007; Dybå e Dingsøyr, 2009; Freire, 2007; Jeffries e Melnik, 2007; Marchenko e Abrahamsson, 2008; Siniaalto e Abrahamsson, 2007). As pesquisas da área de métodos ágeis têm mostrado por meio de diversos estudos experimentais a efetividade e algumas deficiências das principais práticas de métodos ágeis tanto na indústria, quanto na academia (Dybå e Dingsøyr, 2008; Jeffries e Melnik, 2007; Layman et al., 2004; Madeyski, 2010).

Os benefícios da utilização de métodos ágeis envolvem a capacidade de reação a mudanças constantes, a colaboração com o cliente, processos eficientes para gerar produtos de qualidade (que atendem ao cronograma, possuem menos defeitos e que resultam em usuários mais satisfeitos), aprendizagem e melhoria contínua do projeto. Nesse contexto, estudos com o cliente relataram sua satisfação com oportunidades de *feedback* constante. Empresas de desenvolvimento que utilizam XP relatam que seus funcionários ficaram mais satisfeitos com

os seus trabalhos e com o produto final e estudantes universitários acreditam que a utilização de métodos ágeis melhora a produtividade do time (Dybå e Dingsøyr, 2009).

No entanto, Dybå e Dingsøyr (2008) identificaram por meio de diversos estudos já publicados, algumas limitações da utilização de métodos ágeis. Um dos exemplos é a dificuldade de se ter o cliente sempre presente por longos períodos e a dificuldade para introduzir métodos ágeis em projetos grandes e complexos. Além disso, ainda faltam estudos científicos que mostrem a efetividade de algumas práticas ágeis, como por exemplo o TDD, que apesar de comprovadamente resultar em um software de melhor qualidade, ainda produz resultados controversos em termos do impacto na produtividade do time e no projeto do código-fonte (Siniaalto e Abrahamsson, 2007).

A atividade de teste de software no contexto de métodos ágeis possui um papel bastante importante. Diferentemente de métodos tradicionais nos quais os testes ocorrem mais tarde no processo de desenvolvimento, os testes ágeis devem ocorrer de forma frequente, procurando detectar defeitos o mais cedo possível em ciclos de desenvolvimento iterativos e curtos, com um constante *feedback* do cliente. Além disso, a estratégia de desenvolvimento dirigido a testes (TDD) pode ser utilizada para explorar, projetar, desenvolver e testar o software, não devendo ser tratada apenas como uma atividade de testes (Janzen, 2006).

Para apoiar o *feedback* constante da equipe em projetos ágeis são utilizadas diversas práticas como reuniões diárias, reuniões de revisão e retrospectivas. Essas práticas tem como objetivo a melhoria da equipe de desenvolvimento, qualidade do produto e também do processo. Outra prática que apoia o *feedback* constante e a melhoria contínua é a area de trabalho informativa que deve fornecer instrumentos que forneçam dados sobre o andamento do projeto. Esses dados serão coletados por meio de métricas de software que devem medir o progresso, apontar melhorias e dificuldades durante todo o projeto (Crispin e Gregory, 2009).

Os trabalhos de Colette (2009); Crispin e Gregory (2009); Hartmann e Dymond (2006); Sato (2007); Williams et al. (2004b) propuseram abordagens para escolha e utilização de métricas específicas para projetos ágeis. No entanto, a utilização de métricas para acompanhamento da atividade de teste foi pouco explorada, havendo a necessidade de descrever melhor os objetivos de se utilizar cada métrica de teste e diversos aspectos que influenciam a utilização dessas métricas. Além disso, há uma carência de ferramentas que além de automatizar a coleta dessas métricas, forneçam a possibilidade de gerenciamento durante as iterações de desenvolvimento.

Como mencionado anteriormente, a atividade de teste fornece um *feedback* instantâneo a respeito do software que está sendo desenvolvido. Assim sendo, é importante que a equipe de desenvolvimento e teste utilize métricas para acompanhar essa atividade, focando principalmente na melhoria contínua do processo, das práticas e ferramentas de teste utilizadas. Por fim, essa equipe deve conduzir a atividade utilizando métricas para avaliar e estabelecer metas de qualidade para os artefatos de teste produzidos. A partir de um código de teste

de qualidade também será possível medir de forma eficiente a qualidade do software produzido. Nesse sentido, os objetivos derivados das motivações desta dissertação serão descritos a seguir.

1.2 Objetivos

A atividade automatizada de teste em métodos ágeis de desenvolvimento tem relação direta com a validação de novas funcionalidades e mudanças feitas em ciclos iterativos e curtos. Diversas abordagens, estratégias e práticas de teste foram criadas ou adaptadas para o contexto de projetos ágeis, apoiando a integração contínua e o desenvolvimento de soluções simples e que atendam as necessidades do cliente. Nesse sentido, o objetivo deste trabalho é investigar e descrever como a atividade de teste de software pode ser aplicada no contexto de métodos ágeis e, além disso, propor a adoção de um conjunto de métricas de teste que possam ser utilizadas para o acompanhamento dessa atividade. A partir desses objetivos, esperam-se os seguintes resultados:

- caracterização do processo, estratégias e práticas ligadas à atividade de teste de software no contexto de métodos ágeis de desenvolvimento.
- propor a adoção de um conjunto de métricas de teste de software para acompanhamento dessa atividade durante todo o ciclo de vida.
- desenvolvimento do protótipo da ferramenta ATMM (*Agile Testing Metrics Management Tool*) que automatiza algumas dessas métricas.
- estudos de casos com dois projetos de domínios diferentes que utilizaram métodos ágeis, estratégias e práticas de teste ágil, que tem como objetivo verificar o comportamento das métricas propostas neste trabalho no decorrer das iterações. As métricas foram analisadas após as iterações terem sido desenvolvidas.

1.3 Organização

Os próximos capítulos estão organizados da seguinte forma:

- **Capítulo 2 - Métodos Ágeis de Desenvolvimento:** nesse capítulo são descritas de forma geral a origem e as motivações para a utilização de métodos ágeis e também detalhes sobre os principais métodos. Os métodos ágeis são descritos principalmente sob o ponto de vista do fluxo de atividades e práticas conduzidas durante o desenvolvimento de software.

- **Capítulo 3 - Teste de Software:** esse capítulo apresenta conceitos básicos do teste de software tradicional (fases, técnicas e critérios). Também são caracterizados o processo, estratégias e práticas ligadas à atividade de teste no contexto de métodos ágeis. Por fim, são apresentados estudos experimentais que evidenciam os benefícios e dificuldades em termos da atividade de teste em métodos ágeis.

- **Capítulo 4 - Métricas de Software:** nesse capítulo são apresentados conceitos gerais de métricas de software e abordagens para sua escolha. Também são apresentadas as motivações e as formas de se utilizar métricas de acompanhamento de métodos ágeis. É apresentado um conjunto de métricas de teste que podem ser adotadas por uma equipe de desenvolvimento ágil. Por fim, são descritos aspectos gerais da ferramenta ATMM que implementa algumas dessas métricas de acompanhamento de testes e que foi utilizada para os estudos de casos com projetos ágeis.

- **Capítulo 5 - Estudo de Caso:** nesse capítulo são descritos os estudos de casos conduzidos em dois projetos ágeis. O objetivo de conduzi-los foi avaliar a aplicação das métricas de acompanhamento de teste que foram propostas neste trabalho. É apresentado o método que foi utilizado para analisar as métricas de acompanhamento de testes e as informações coletadas a partir do questionário sobre informações gerais dos projeto. Esses resultados são sintetizados, analisados e discutidos.

- **Capítulo 6 - Conclusões e Trabalhos Futuros:** esse capítulo conclui este trabalho, dando destaque às contribuições para a área de teste de software em métodos ágeis, análise das dificuldades e limitações e, por fim, a descrição de possíveis trabalhos futuros, além de publicações esperadas.

- **Apêndice A - Questionário Informações Gerais (Projetos Ágeis):** esse apêndice apresenta o questionário que foi utilizado para coletar informações gerais do projeto, incluindo a aderência e maturidade da atividade de teste em projetos ágeis.

- **Apêndice B - Guia de Execução de *scripts JaBUTi* :** esse apêndice apresenta um guia para execução de *scripts* da ferramenta *JaBUTi* para medir a cobertura de código de um projeto.

Capítulo

2

Métodos Ágeis de Desenvolvimento

O foco principal dos métodos ágeis de desenvolvimento de software, segundo seus proponentes, é a simplicidade e a velocidade (Abrahamsson et al., 2002). Nesse contexto, a equipe de desenvolvimento ágil concentra-se na entrega rápida de funcionalidades realmente necessárias, coletando *feedback* e reagindo às mudanças tecnológicas e no negócio (Abrahamsson et al., 2003). Além disso, o *feedback* do cliente permite o aumento de sua satisfação, já que ele passa a ter uma melhor visão do andamento do projeto e do sistema em desenvolvimento (Williams e Cockburn, 2003).

Neste capítulo as características de métodos ágeis são descritas, bem como os valores e princípios que guiam o desenvolvimento nesses métodos. Também são apresentadas as práticas de desenvolvimento e condução de projetos, que são atividades que garantem a agilidade e principalmente a qualidade no processo de desenvolvimento e no produto resultante. Na Seção 2.1 são discutidas a origem e a descrição geral sobre os valores e princípios comuns a todos os métodos ágeis. Nas Seções 2.2 a 2.5 os principais métodos ágeis são descritos e detalhados principalmente sob o ponto de vista do fluxo de atividades e práticas conduzidas durante o desenvolvimento de software. Por fim, na Seção 2.6 são apresentadas as considerações finais do capítulo .

2.1 Métodos Ágeis de Desenvolvimento

Segundo Baresi et al. (2006), historicamente engenheiros de software só poderiam passar para a fase de projeto e implementação após passar por uma fase exaustiva de elicitação e especificação de requisitos. Pesquisadores desenvolveram então métodos, técnicas e ferramentas para apoiarem uma evolução mais flexível de processo e produto, focando na qualidade do produto, no custo e na eficiência do projeto. Nesse contexto, foram introduzidos modelos de processo evolucionário, como o modelo incremental e o modelo baseado em prototipação. Mais recentemente, a ideia desses modelos foi incorporada em métodos ágeis.

A agilidade, para uma organização de desenvolvimento de software, é a habilidade de adotar e reagir rapidamente e apropriadamente a mudanças no seu ambiente e por exigências impostas pelos clientes (Nerur et al., 2005). Portanto, a agilidade não trata apenas do tamanho ou da velocidade das mudanças, mas é, principalmente, um meio de atingir a flexibilidade no projeto de software.

Segundo Pressman (2006), a agilidade pode ser aplicada a qualquer processo de software. No entanto, o autor enfatiza algumas práticas essenciais para que isso seja possível: (1) permitir à equipe de projeto adaptar tarefas e aperfeiçoá-las; (2) conduzir um planejamento para que se entenda a fluidez de uma abordagem de desenvolvimento ágil; (3) eliminar tudo, menos os produtos mais essenciais e mantê-los simples; (4) enfatizar uma estratégia de entrega incremental que forneça o software funcionando ao cliente o mais rápido possível para o tipo de produto e ambiente operacional desejados.

Abrantes e Travassos (2007) também descrevem as características de agilidade no contexto de métodos ágeis de desenvolvimento de software, a partir da condução de uma revisão sistemática de literatura:

> "*Um método, para ser caracterizado como ágil, deve apresentar, em um grau adequado ao contexto de desenvolvimento de software em que se insere, as características de adaptabilidade, incrementalidade, iteratividade, colaboratividade, cooperação, orientação a pessoas, parcimônia (leanness) e restrição de prazo.*" - Abrantes e Travassos (2007)

A criação do manifesto de desenvolvimento ágil de software é apontado como sendo o guia para todos os métodos de desenvolvimento considerados ágeis (Beck et al., 2001). Esse manifesto foi lançado em 2001 por um grupo de desenvolvedores experientes, consultores e líderes da comunidade de software. Foi construído para discutir ideias e procurar alternativas aos processos burocráticos e às práticas adotadas nas abordagens tradicionais de engenharia de software e gerência de projetos (Sato, 2007).

O manifesto define os valores a serem utilizados por todos os métodos ágeis:

- **Indivíduos e interações** são mais importantes que processos e ferramentas;
- **Software funcionando** é mais importante que documentação completa e detalhada;
- **Colaboração com o cliente** é mais importante que negociação de contratos;
- **Adaptação às mudanças** é mais importante que seguir um plano.

Os valores destacados são importantes em um projeto ágil, no entanto, ferramentas, processos e os demais itens não destacados também devem ser valorizados. Esses valores, são descritos a seguir (Highsmith e Cockburn, 2001):

- **Importância de indivíduos e suas iterações:** em projetos ágeis, os desenvolvedores e clientes representam papéis importantes em um projeto de software. Em métodos tradicionais, os clientes participam principalmente na especificação dos requisitos, com participação mínima em outras atividades. No desenvolvimento ágil, os clientes (que representam os usuários do sistema) trabalham em pequenos times com os desenvolvedores como membros ativos da equipe. Clientes e desenvolvedores, determinam de forma conjunta as funcionalidades a serem implementadas em cada ciclo de desenvolvimento. Posteriormente, os clientes também validam essas funcionalidades implementadas e testadas pelos desenvolvedores.
- **Software funcionando:** equipes ágeis buscam a entrega contínua de software completamente testado e funcionando. Isso deixa o cliente satisfeito com as novas funcionalidades entregues, desenvolvendo soluções simples e diminuindo assim a necessidade de uma documentação complexa.
- **Colaboração com o cliente:** a colaboração significa que o cliente e os desenvolvedores estão no mesmo time. Por meio da troca de diversas experiências e conhecimento entre o time, os requisitos se tornam mais claros e as mudanças de direção no desenvolvimento do software podem ocorrer rapidamente, produzindo resultados melhores e projetos que demandem menos recursos financeiros.
- **Adaptação às mudanças:** em vez de preocupar-se tanto com os planos do projeto de software, que muitas vezes podem estar desatualizados, é importante lidar com a adaptação eficiente às mudanças durante o desenvolvimento em projetos ágeis. Por meio de pequenos ciclos de desenvolvimento e diversas práticas, o desenvolvimento de software ágil reduz o custo de mudanças frequentes.

Highsmith (2002) sugere que as diferenças entre os métodos de desenvolvimento tradicionais e métodos ágeis são baseadas principalmente em suposições sobre os clientes. Em

processos tradicionais, os clientes não sabem realmente quais os requisitos necessários para o software. Portanto, em métodos tradicionais, os desenvolvedores desejam uma especificação detalhada, afirmando que eles somente desenvolveram o sistema seguindo o que foi especificado pelo cliente. Além disso, os desenvolvedores muitas vezes entregam funcionalidades extras para as futuras necessidades dos clientes. Em métodos ágeis, os clientes e desenvolvedores não possuem o total conhecimento a respeito dos requisitos no início do projeto. Clientes e desenvolvedores de uma equipe ágil entendem e descrevem os requisitos do sistema conforme o processo de desenvolvimento evolui, enfatizando também a simplicidade e o desenvolvimento somente do que o cliente necessita. As diferenças de filosofia entre métodos tradicionais e métodos ágeis resultam em diversas práticas diferentes como planejamento e controle, papel assumido entre os desenvolvedores, papel dos clientes e modo de condução de projeto. Essas diferenças são sintetizadas na Tabela 2.1.

Tabela 2.1: Principais diferenças entre o desenvolvimento tradicional e o desenvolvimento ágil (Nerur e Balijepally, 2007; Nerur et al., 2005)

	Desenvolvimento Tradicional	**Desenvolvimento Ágil**
Suposições Fundamentais	Sistemas são totalmente especificáveis, previsíveis e são feitos por meio de um planejamento meticuloso e extensivo.	Alta qualidade, software adaptativo, pode ser desenvolvido por equipes pequenas utilizando os princípios de melhoria contínua do projeto e testes baseados no *feedback* rápido e mudanças.
Processo envolvido no projeto	Deliberado e formal, com sequência linear de passos, formulação e implementação separadas, dirigida a regras.	Emergente, iterativo e exploratório, conhecimento e ação são inseparáveis, mais do que regras formais.
Modelo de desenvolvimento	Modelo de ciclo de vida (Cascata, Espiral ou alguma variação)	Modelo de entrega evolucionário
Controle	Centrado no processo	Centrado nas pessoas
Comunicação	Formal	Informal e bastante importante
Papel do Cliente	Importante	Crítico
Gerenciamento do Conhecimento	Explícito (documentação)	Tácito (pessoas)
Características Principais	Controle e direção Evitar conflitos Formaliza inovação Gerente é controlador Projeto precede a implementação	Colaboração e comunicação; integra diferentes visões de mundo Adota conflitos e argumentações Encoraja a exploração e criatividade; oportunismo Gerente é facilitador Projeto e implementação são inseparáveis e se desenvolvem iterativamente
Forma/Estrutura Organizacional Desejada	Burocrática, com alto grau de formalização (dirigido a grandes organizações).	Flexível e participativo, encorajando uma ação social cooperativa (dirigida a pequenas e médias organizações).

Os métodos ágeis de desenvolvimento tiveram origem nos Estados Unidos e Europa. O escopo e o tamanho da equipe nesses métodos podem variar, sendo que alguns dos métodos podem ser utilizados tanto por projetos pequenos, como também em grandes projetos. Métodos ágeis geralmente são indicados para equipes pequenas e times co-alocados. No entanto,

diversos métodos podem ser utilizados com grandes equipes e utilizando o desenvolvimento distribuído. Em relação a mecanismos de abstração, a maioria dos métodos funciona de maneira mais eficiente com o paradigma orientado a objetos e alguns com o desenvolvimento baseado em componentes (DBC). A partir da Tabela 2.2 são sintetizadas as características gerais dos métodos ágeis com base em alguns trabalhos que analisaram comparativamente esses métodos (Abrahamsson et al., 2002; Qumer e Henderson-Sellers, 2008; Strode, 2005; Thomas, 2006). As seguintes características são descritas na Tabela: (1) origem (ano de criação e país) e criadores, (2) escopo (tamanho de projeto) e tamanho da equipe, (3) estilo de desenvolvimento, (4) mecanismo de abstração (orientação a objetos - OO / desenvolvimento baseado em componentes - DBC), (5) ambiente (times co-alocados - CO / distribuídos - DI / não especificado - NE) e (6) principais contribuições.

Tabela 2.2: Comparativo entre Métodos Ágeis (Características Gerais)

	(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**
XP	1999 (EUA) Kent Beck	Pequeno e médio. Equipes <10	Iterativo, rápido	OO	CO e DI[1]	Valores e princípios de agilidade, e principalmente práticas como TDD, refatoração, programação em pares, histórias e integração contínua.
Scrum	1995, 2001 (EUA) Ken Schwaber e Jeff Sutherland	Pequeno, médio e escalável para grandes . Equipes <10 e múltiplos times	Iterativo, rápido	OO	NE	Gerenciamento de Projetos, equipes auto-organizadas e multifuncionais, reuniões, *Sprints* e Listas *Backlog*.
FDD	2001 (UK) Jeff De Luca e Peter Coad	Pequenos, médios e grandes (projetos e aplicações de negócio). Equipes sem limite (escalável).	Projeto e Construção Iterativa	OO	NE	Altamente escalável, frequente verificação, utilização de modelos, gerência de configuração, *builds* regulares, inspeções e testes de unidade.
Crystal	2002 (EUA e Europa) Alistair Cockburn	Pequeno e Médio Equipes de 6 pessoas (*Clear*) e múltiplos times de 40 e 80 pessoas (*Orange* e *Red*)	Iterativo, rápido	OO	NE	Práticas, produtos de trabalho, documentação e ferramentas de acordo com o método selecionado. Monitoramento do progresso e reuniões de reflexão, utilização testes de regressão automatizados.
ASD	2000 (EUA) Jim Highsmith	Projetos grandes e complexos. Não mencionado	Iterativo, rápido (desenvolvimento distribuído)	OO/ DBC	CO e DI	Modelo de gerenciamento adaptativo, inspeções, reuniões JAD e revisões de qualidade.
DSDM	1997 (UK) *DSDM Consortium*	Pequenos e grandes projetos (sistemas de informação). Mínimo de 2 e máximo de 6 (múltiplos times)	Iterativo, rápido e cooperativo	OO/ DBC	NE	Participação ativa do cliente, entregas frequentes, times com poder de decisão e os testes são integrados durante todo o ciclo de vida. Utilização de protótipos, *timeboxing* e *workshops*.

[1] O XP pressupõe time co-alocados, no entanto é possível aplicar o XP de forma distribuída, que pode resultar na limitação da iteração entre os desenvolvedores e clientes.

Em resumo, métodos tradicionais de desenvolvimento incorporam um planejamento extensivo, processos detalhados, e reutilização rigorosa, para fazer com que o desenvolvimento seja uma atividade eficiente e previsível que amadurece gradualmente em direção à perfeição (Boehm, 2002). Métodos ágeis, ao contrário, procuram enfrentar o desafio de um mundo não

previsível, enfatizando o valor da competência das pessoas e as suas relações decorrentes do desenvolvimento de software (Nerur et al., 2005).

2.2 *eXtreme Programming* (XP)

A Programação Extrema (*eXtreme Programming* – XP), é o método ágil mais conhecido e enfatiza a colaboração, criação rápida e precoce do software (Larman, 2003). Segundo Beck e Andres (2004), o XP é um método leve, que procura fazer o necessário para trazer valor para o cliente por meio do software funcionando. Os autores também enfatizam o desenvolvimento de software por meio de suas práticas de desenvolvimento que são apoiadas pelos valores e princípios. O método XP adapta-se a requisitos vagos ou a constante mudança e também funciona em times de qualquer tamanho. O objetivo principal do XP é a excelência no desenvolvimento de software, visando um baixo custo, poucos defeitos, alta produtividade e alto retorno de investimento (Sato, 2007).

O XP possui pequenas iterações com pequenos incrementos de versões e *feedbacks* rápidos. Seu planejamento ocorre de forma incremental, existe a participação próxima do cliente, constante comunicação e coordenação entre os integrantes da equipe. Além disso, o XP possui algumas práticas ligadas diretamente ao desenvolvimento de software, como a refatoração (melhoria de código), a integração e testes executados de forma contínua, o código coletivo e a programação em pares.

A maioria das características do XP, como a refatoração, a programação em pares, a adaptação às mudanças, a integração contínua, o desenvolvimento iterativo e a ênfase a atividade de teste foi originada a partir de elementos-chave presentes na cultura da comunidade da linguagem `Smalltalk` desde a década de 1980 (Sato, 2007). Kent Beck aplicou inicialmente essas idéias no projeto C3 (*Chrysler Comprehensive Compensation System*), sendo que a ideia que originou o nome Programação Extrema, era juntar boas práticas de programação já conhecidas pela indústria e utilizá-las ao extremo (Sato, 2007). A revisão de código, por exemplo, é feita por meio da programação em pares, na qual a revisão é feita em tempo integral. Os testes automatizados, nesse mesmo contexto, são levados ao extremo e são escritos antes do código e executados com bastante frequência. Os conceitos envolvidos na definição do XP, descritos por Beck e Andres (2004).

Os principais aspectos do método XP incluem (Beck e Andres, 2004):

- Uma filosofia para o desenvolvimento de software baseada nos valores de comunicação, *feedback*, simplicidade, coragem e respeito.

- Um conjunto de práticas comprovadamente úteis para melhorar o desenvolvimento de software. As práticas expressam os valores do XP e são técnicas utilizadas no cotidiano dos membros de uma equipe XP. A 2ª edição do XP é composta de 24 práticas (13 primárias e 11 corolárias).

- Um conjunto complementar de 14 princípios, que são técnicas intelectuais que auxiliam a tradução dos valores em práticas, úteis quando as práticas existentes não resolvem seu problema particular.
- Uma comunidade que compartilha os mesmos valores e muitas das mesmas práticas.

A seguir será descrito o funcionamento do ciclo de desenvolvimento do XP (Seção 2.2.1), seus valores, princípios e práticas (Seção 2.2.2). Por fim na seção 2.2.3 serão descritos os papéis da equipe de XP.

2.2.1 Ciclo de Desenvolvimento do XP

O XP é composto de seis fases: exploração, planejamento, iterações para as versões, produção, manutenção e morte (Abrahamsson et al., 2002; Beck, 2000). Os times de desenvolvimento que utilizam XP executam quase todas as atividades de desenvolvimento de software simultaneamente (Warden e Shore, 2007). Estas atividades são conduzidas por meio de iterações, que são incrementos semanais de trabalho. Toda semana, o time desenvolve um pouco dessas atividades. Eles trabalham em histórias (*user stories*): pequenas funcionalidades (*features*), ou parte de funcionalidades, que tem um valor ao cliente. No geral, toda semana, o time se compromete a entregar algumas histórias, que passam por todas as fases de desenvolvimento. Por fim, no final da semana, eles instalam o software para revisão interna, que em alguns casos pode incluir a instalação para os clientes do projeto.

A seguir descreve-se cada uma das fases do ciclo de desenvolvimento do XP, que são ilustradas na Figura 2.1:

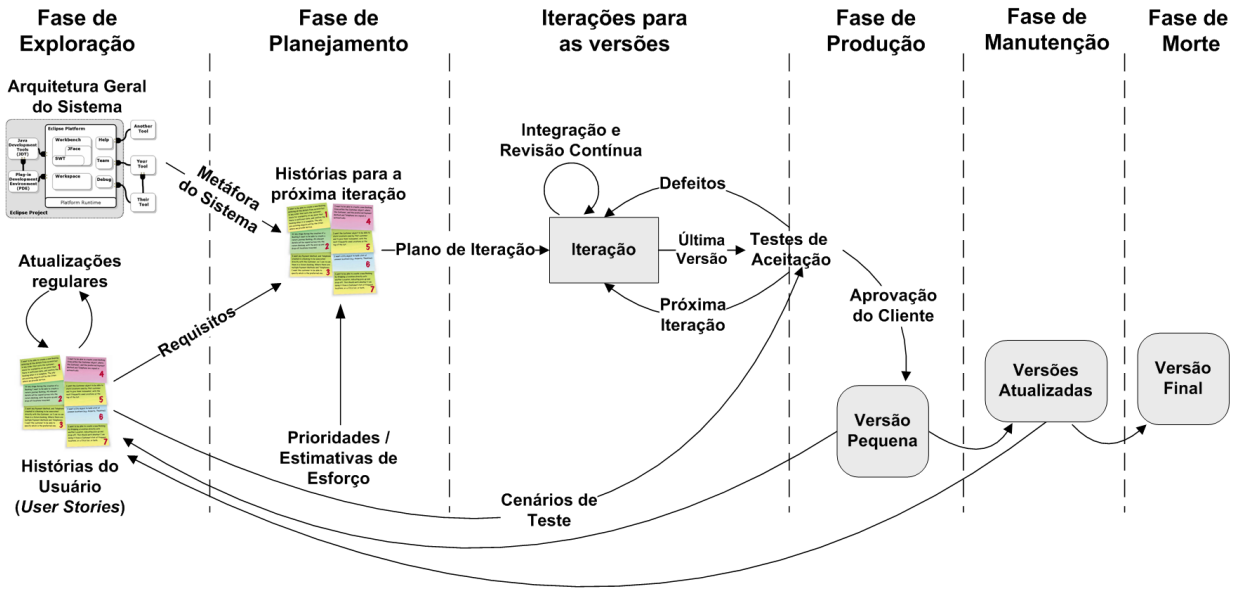


Figura 2.1: Ciclo de vida do eXtreme Programming (Adaptado de Abrahamsson et al. (2002); Don Wells (2006))

Fase de Exploração

Clientes escrevem histórias (também chamada de histórias do usuário (*user stories*)) descrevendo características e funcionalidades requeridas para o software a ser desenvolvido. Cada história é escrita pelo cliente e colocada em um cartão de indexação. Ao mesmo tempo os programadores exploram ferramentas, tecnologias e práticas que eles utilizarão.

Fase de Planejamento

A fase de planejamento leva apenas alguns dias, nos quais os clientes priorizam as histórias e os programadores estimam o esforço para cada história. A priorização das histórias é feita de acordo com um valor (isto é, uma prioridade) para história, com base no valor do negócio global da característica ou da função, podendo inclusive depender também da presença de outra história (Pressman, 2006). O esforço para cada história é medido em semanas de desenvolvimento e se passarem mais do que três semanas, o cliente deve dividir a história em histórias menores e a atribuição de valor e custo ocorre novamente.

Um ou mais testes automatizados de aceitação devem ser criados para verificar se cada história do usuário foi implementada corretamente de acordo com as necessidades do cliente e também podem ser utilizados caso algum dos requisitos sejam difíceis de entender. Esses testes descrevem necessidades de negócio dos clientes por meio de exemplos concretos de negócio, que devem ser testados de forma automatizada no final da fase de iteração (Mugridge, 2008). Os clientes e a equipe de desenvolvimento devem concordar com a data para a primeira versão e um conjunto de histórias para esta versão.

Fase de Iterações para as versões

Nessa fase os programadores escolhem histórias para uma iteração que pode durar de uma a três semanas (Abrahamsson et al., 2002). Por meio da seleção de histórias para a primeira iteração é estabelecida a estrutura do sistema de forma geral, concebendo assim uma arquitetura inicial do sistema. O cliente decide que histórias serão selecionadas para cada iteração. O conjunto de histórias podem ser implementadas imediatamente (em poucas semanas), ou as histórias com valor mais alto são implementadas ou as de maior risco (Pressman, 2006).

O XP utiliza um projeto e uma arquitetura incremental para criar e melhorar continuamente o projeto em pequenos passos (Warden e Shore, 2007). Esse trabalho é feito pelo desenvolvimento dirigido a testes (ou *test-driven development* - TDD), uma atividade de desenvolvimento que une testes, codificação, projeto e arquitetura. Para dar suporte a esse processo, programadores trabalham em pares, o que aumenta a capacidade intelectual exercida sobre cada tarefa e garante que uma pessoa em cada par sempre tenha tempo para pensar sobre questões de projeto de forma ampla. Programadores também são responsáveis por gerenciar o ambiente de desenvolvimento. Eles utilizam um sistema de controle de

versões para o gerenciamento de configuração e mantém sua própria compilação (*build*) de forma automatizada. Programadores integram seu código em poucas horas e garantem que cada integração está tecnicamente pronta para ser instalada.

Para apoiar este esforço, os programadores também procuram manter padrões de codificação e compartilham a autoria do código. Os testes funcionais (de aceitação) criados pelo cliente são executados no fim de cada iteração. No fim da última iteração o sistema está pronto para a fase de produção (Abrahamsson et al., 2002).

Fase de Produção

Antes do sistema ser entregue ao cliente, é necessário testes extras para provar a qualidade do sistema e principalmente checar a sua performance. Mudanças em histórias também podem ser necessárias, e devem ser incluídas na versão atual somente depois de discussão com o cliente. Estas funcionalidades também podem fazer parte de versões futuras. As sugestões e ideias adiadas são documentadas para uma posterior implementação, ou seja, para a fase de manutenção.

Fase de Manutenção e Fase de Morte

Na fase de manutenção, depois da entrega da primeira versão novas funcionalidades são desenvolvidas e o sistema continua em operação. Novas pessoas podem ser incorporadas no time de desenvolvimento quando a estrutura do time é mudada para se encaixar com a produção atual. A fase de morte inicia-se quando os clientes não conseguirem mais gerar uma nova história. Nessa fase a documentação necessária é escrita, e pode ser dado treinamento para que outra equipe realize manutenções posteriores. Outra razão para a morte do projeto é quando ele se torna extremamente caro ou se ele não atingir os resultados esperados.

2.2.2 Valores, Princípios e Práticas do XP

O XP, segundo Beck e Andres (2004), é um estilo de desenvolvimento de software, focado na aplicação de técnicas de programação, comunicação clara e um time de desenvolvimento que permite a execução de tarefas até então não imaginadas. O método possui três componentes essenciais: valores, princípios e práticas. Os valores e práticas são complementares, sendo que os valores dão razão às práticas, enquanto as práticas devem evidenciar os valores. Para auxiliar na tradução dos valores em práticas são utilizados os princípios, que são técnicas utilizadas quando as práticas propostas não se aplicam em uma situação específica. Os valores, princípios e práticas serão descritos nessa Seção.

2.2.2.1 Valores do XP

Na filosofia de desenvolvimento de software do XP, as equipes devem valorizar a comunicação entre as pessoas ao invés de documentos escritos, o *feedback* constante e contínuo para guiar a qualidade e a adaptação durante o processo, a simplicidade do software e do processo sempre procurando atender a necessidade do cliente com mais urgência, a coragem de trabalhar de forma rápida e desenvolver novamente se necessário e, por fim, o respeito que serve como base para os outros quatro valores, sendo importante que o time de desenvolvimento reconheça que a excelência no desenvolvimento de software depende das pessoas e elas devem se respeitar para conseguir extrair o máximo de seu potencial (Beck e Andres, 2004). Esses valores serão as bases a serem utilizadas como critério de julgamento do que a equipe de desenvolvimento deve ou não fazer, dando propósito e direção as práticas do XP. Os cinco valores do XP são descritos abaixo:

- **Comunicação:** o XP prioriza a comunicação pessoal e oral, acreditando que falar é melhor do que escrever e aceitando a observação de que problemas na comunicação são a causa da maioria das dificuldades de projeto. A comunicação deve ser feita entre os programadores por meio da programação em pares, reuniões diárias e jogo do planejamento, e também deve promover envolvimento do cliente em escrever testes de aceitação e participar de reuniões de avaliação das iterações.

- **Simplicidade:** o segundo valor se refere à preocupação em desenvolver soluções simples para resolver os problemas de hoje, evitando desperdícios com soluções genéricas para modificações futuras. A simplicidade será aplicada não somente ao projeto do software, mas também em outros elementos como os requisitos ou ferramentas de gerência. A simplicidade e a comunicação são valores complementares. Quanto mais simples o sistema, menos comunicação ocorre e quanto maior a comunicação, maior o entendimento e portanto mais fácil será a manutenção da qualidade do projeto.

- ***Feedback*:** Esse valor guia a qualidade e adaptabilidade do projeto de software (Larman, 2003). O XP promove ciclos curtos e constantes de *feedback*, nos mais variados aspectos do desenvolvimento de software. As ferramentas de *feedback* são o contato de perto com o cliente e a disponibilidade de um conjunto de testes automatizados, que são desenvolvidos junto com sistema. O *feedback* é um componente importante da comunicação, no qual é necessário comunicar-se para ter o *feedback*.

- **Coragem:** segundo Marchesi (2004) todos os métodos e processos são ferramentas para cuidar e reduzir nossos medos. Quanto mais medo os desenvolvedores tiverem de um projeto de software, métodos mais complexos serão necessários. A comunicação, simplicidade e *feedback* permitem cuidar com coragem até grandes mudanças nos requisitos e refatorações substanciais no sistema.

- **Respeito:** Se os membros da equipe de desenvolvimento não se importarem uns com os outros e o seu trabalho, o XP não vai funcionar. É importante respeitar os colegas do time e suas contribuições, a organização e as pessoas cuja vida é afetada pelo sistema que o time está desenvolvendo.

2.2.2.2 Princípios do XP

Os princípios são ferramentas que traduzirão os valores em práticas concretas, e que serão úteis quando as práticas existentes não resolverem um problema particular. Um exemplo citado por Beck e Andres (2004) é a comunicação por intermédio de conversas diárias ou a utilização de documentos longos. Qual seria a forma mais eficiente? Nesse caso, o princípio da humanidade sugere que a conversa satisfaz melhor as necessidades humanas de relacionamento. Por outro lado, a comunicação escrita permite alcançar uma larga audiência, sendo uma forma de comunicação de apenas uma via, permitindo uma maior clareza, *feedback* imediato, a exposição de diversas idéias, entre outros aspectos que não podem ser feitos com um documento escrito.

Em um projeto XP deve-se criar oportunidades para o crescimento, contribuição, participação e relacionamento entre o time de desenvolvimento. A equipe de desenvolvimento deve incluir diferentes conhecimentos, habilidades e características, procurar continuamente a melhoria e refinar as atividades ao longo do tempo. Todas as atividades do XP devem trazer benefícios aos envolvidos e deve-se aplicar a estrutura de uma solução em outros contextos e inclusive escalas.

No contexto de melhoria, a equipe também deve se preocupar com a reflexão e um fluxo contínuo de atividades de software que agregam valor ao negócio e maximizam o valor do projeto. Membros da equipe XP devem aceitar responsabilidades e procurar sempre executar passos pequenos na direção correta, procurando dividir tarefas complexas e diminuir os riscos. Problemas devem ser vistos como uma oportunidade para melhoria e aprendizagem, e devem ser resolvidos agregando diversas práticas e soluções diferentes. Por fim, a qualidade deve ser sempre idealizada, refletindo na melhoria de outras características do sistema como a produtividade, eficiência e motivação. Um projeto XP não considera a qualidade como uma das variáveis de controle. O custo e o tempo também são geralmente fixos, deixando o escopo como principal variável na negociação com o cliente (Beck e Fowler, 2000). Além disso, a simplicidade de soluções deve ser tratado como atributo de qualidade.

2.2.2.3 Práticas do XP

As práticas são técnicas utilizadas no cotidiano dos membros de uma equipe XP, para que se atinja o estado ideal de um desenvolvimento efetivo. Essas práticas complementam uma às outras, ampliando os seus efeitos. Enquanto as práticas evidenciarão os valores, os valores darão razão às práticas.

As práticas do XP, segundo Beck e Andres (2004) são dependentes da situação, sendo que a equipe deve escolher diferentes práticas para atender a essas condições, no entanto os valores não devem mudar para se adaptar a uma nova situação. Na segunda versão do XP, as práticas foram divididas em 13 práticas primárias e 11 práticas corolárias, totalizando 24 práticas que podem ser adotadas de forma incremental (Freire, 2007). As práticas primárias são independentes e podem ser aplicadas individualmente de maneira segura, provendo melhorias imediatas à equipe de desenvolvimento. As práticas corolárias são mais difíceis ou arriscadas de se implementar antes de se ter domínio e experiência prévia com as práticas primárias.

Práticas primárias do XP

As práticas primárias podem ser aplicadas no início da adoção do XP para melhorar o desenvolvimento de software em uma organização. Segundo Freire (2007), essas práticas podem ser facilmente implantadas pois são seguras e devem ser introduzidas em pequenos passos, para evitar uma mudança muito rápida na cultura da organização. A adoção depende completamente no ambiente de desenvolvimento da equipe ágil e também o que a equipe visualiza como sendo uma oportunidade para melhoria (Beck e Andres, 2004).

(1) Análise de Requisitos e Planejamento

- **Histórias:** as funcionalidades do sistema são descritas em histórias, pequenas descrições das necessidades do cliente. Os cartões de história são apenas lembretes de diálogos da equipe de desenvolvedores do XP com o cliente e é relacionada diretamente a área de trabalho informativa.

- **Ciclo Semanais:** essa prática do XP sugere que se deve planejar o trabalho de cada iteração uma semana por vez. No início de cada semana haverá uma reunião na qual é feita uma revisão do progresso até a data, os clientes escolhem histórias que devem ser implementadas na semana e as histórias são quebradas em tarefas. Segundo Beck e Andres (2004), o começo da semana acontece com a escrita de testes automatizados que devem passar quando as histórias forem completadas. O objetivo é ter o sistema desenvolvido e testado no fim de cada semana podendo celebrar o progresso do projeto.

- **Ciclo Trimestral:** em uma escala de tempo maior, as versões são planejadas a cada trimestre. Isto é feito por intermédio de reflexões no time, projeto, progresso e o seu alinhamento com objetivos gerais. Esse plano é feito em mais alto nível, geralmente representado por um tema. Durante o planejamento do trimestre, o time identifica gargalos (principalmente externos à equipe), inicia reparos e escolhe as histórias mais alinhadas ao tema e que serão implementadas no trimestre (Beck e Andres, 2004).

- **Folga:** para evitar atrasos ou a necessidade de renegociação de escopo, o planejamento deve conter explicitamente espaços de folga. Como consequência, é criado um vínculo de confiança e responsabilidade entre a equipe e cliente evitando que eventuais atrasos atrapalhem a entrega da iteração ou da versão.

(2) Time de Desenvolvimento e Fatores Humanos

- **Sentar Junto:** times de desenvolvimento devem trabalhar em um espaço amplo, capaz de acomodar todo o time, maximizando a comunicação. Essa prática incentiva a troca de tradicionais cubículos por áreas de uso comum nas quais os membros da equipe possam se agrupar para discutir no quadro branco, sentar juntos para trabalhar em pares e espalhar gráficos e informações na parede (Sato, 2007).

- **Time Completo:** a equipe precisa de pessoas com todas as habilidades e perspectivas necessárias para o sucesso do projeto. O time completo segundo Beck e Andres (2004) deve ser dinâmico o bastante para adaptar-se às habilidades e atitudes que foram importantes ao estado atual do projeto. Todos devem trabalhar em um espírito de contribuição para a equipe, visando o bom andamento do projeto.

- **Área de Trabalho Informativa:** o local de trabalho deve fornecer instrumentos informativos para fornecer informações a respeito do andamento do projeto e de tarefas a serem realizadas. Alguns exemplos desses instrumentos são cartões com histórias em um mural, quadros brancos, notas em papel nas paredes, e gráficos como o *Work Burn-Down* proposto pelo Scrum (Schwaber, 2004; Schwaber e Beedle, 2001) para acompanhar a velocidade da equipe. Do ponto de vista humano, a área de trabalho deve ser um ambiente agradável, contribuindo com a prática de trabalho energizado.

- **Trabalho Energizado:** a prática diz que os desenvolvedores devem estar renovados e o trabalho deve durar quantas horas o time puder se manter produtivo e energizado. O trabalho energizado tem relação com o planejamento, no qual o número de horas dedicadas ao projeto deve ser definido realisticamente. Não adianta desgastar sua equipe com trabalhos em hora-extra, pois a produtividade irá cair inevitavelmente.

- **Programação em pares:** é uma das práticas mais conhecidas do XP. Essa prática sugere que o código seja escrito (analisado, desenvolvido e testado) sempre por dois programadores, promovendo o trabalho coletivo e colaborativo, unindo a equipe, melhorando a comunicação e visando principalmente a qualidade do código. A programação em pares valoriza principalmente a comunicação. Os pares são trocados regularmente e geralmente, a seleção dos pares depende da tarefa a ser realizada, da disponibilidade dos membros da equipe e da experiência de cada um. O objetivo principal é espalhar o conhecimento da equipe do sistema a equipe inteira, compartilhando

técnicas, competências entre os membros da equipe e produzindo código de melhor qualidade devido a revisão executada pelos pares.

A Figura 2.2 ilustra um ambiente de desenvolvimento XP, que possui diversos instrumentos informativos a respeito do estágio atual do desenvolvimento e também fornece infraestrutura para que a equipe desenvolva utilizando a programação em pares.

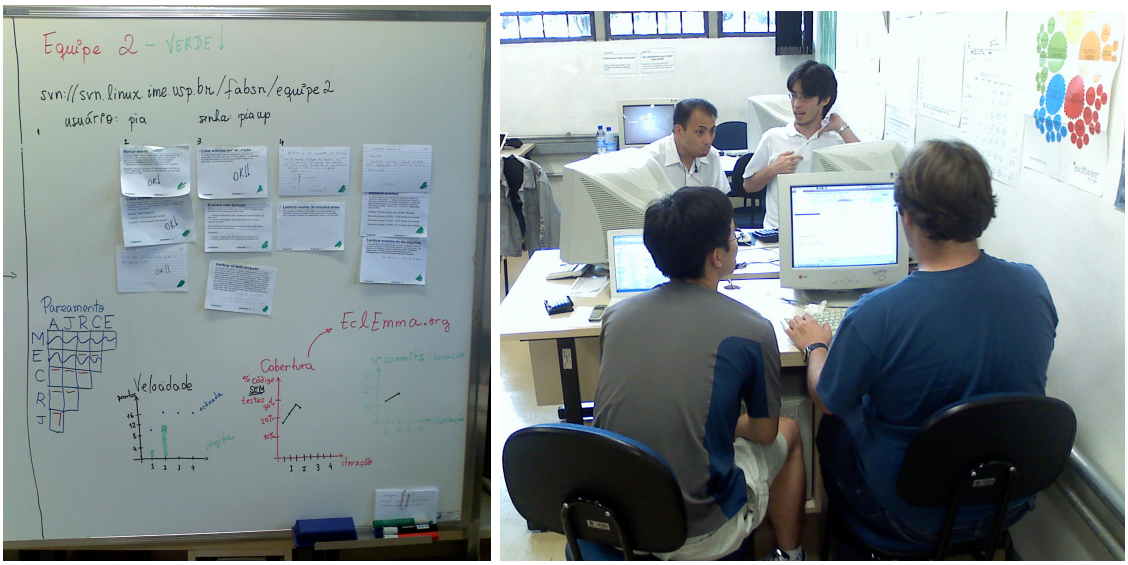


Figura 2.2: Ambiente de Desenvolvimento XP - Área de Trabalho Informativa e Programação em pares (Curso de Verão IME/USP 2009).

(3) *Design*[1]

- ***Design* Incremental:** o XP se opõe a produzir ao *design* completo logo no começo. O *design* é indispensável para um bom código, a questão é quando ele deve ocorrer. Beck e Andres (2004) sugerem que ele seja feito incrementalmente durante o código. Para minimizar o custo com mudanças no futuro, os desenvolvedores devem sempre implementar o *design* mais simples, com o mínimo da complexidade e flexibilidade necessária para atender às necessidades de negócio atuais (Sato, 2007).

- **Desenvolvimento Dirigido por Testes (TDD)**: antes de atualizar e adicionar um código, é necessário escrever testes para verificar o código e trabalhar na melhoria do código por meio da refatoração. Mais detalhes a respeito desta prática, assim como os testes de aceitação que complementam esta prática serão descritos na Seção 3.2.

(4) Codificação e Versões (*Releasing*)

- ***Build* em 10 Minutos:** exige que o sistema deve ser compilado por completo e todos os testes devem ser executados, de maneira automatizada, no máximo em 10 minutos. O objetivo dessa prática é aumentar os ciclos de *feedback* e diminuir ao máximo o

[1]Nesta dissertação o termo "*design*" se refere ao termo "projeto"

tempo entre a introdução e descoberta de um defeito. Essa prática tem relação direta com o fluxo de desenvolvimento sugerido pelo XP.

- **Integração Contínua:** o objetivo dessa prática é garantir que todo código produzido será integrado a um repositório comum com a maior frequência possível, após algumas horas de desenvolvimento ou em um dia no máximo. Isto facilitará a entrega de versões de software com todas as mudanças recentes ao cliente, tornando-se uma atividade simples e automatizada e maximizando o *feedback* do cliente e do time de desenvolvimento. Segundo Freire (2007), esta prática pode ser aplicada de diversas maneiras, sendo importante manter o processo de *build* do sistema rápido e automático para que a o código integrado possa ser compilado e os testes executados sem muita perda de tempo. Dessa forma, o conhecimento do sistema se espalha por toda a equipe mais facilmente e a dificuldade de realizar uma integração grande se dilui em diversas integrações pequenas e frequêntes.

Práticas corolárias do XP

As práticas corolárias do XP são difíceis ou perigosas de serem implementadas, e também devem ser aplicadas incrementalmente. Antes que a equipe XP comece a aplicar as práticas corolárias, deve-se já ter um domínio das práticas primárias. Beck e Andres (2004) citam o exemplo da implantação diária, que deve ser aplicada somente quando a taxa de defeitos esteja baixa (com a aplicação da programação em pares, integração contínua e o TDD).

(1) Análise de Requisitos e Planejamento

- **Envolvimento Real do Cliente:** as pessoas cujas vidas são afetadas pelo sistema devem se tornar parte do time. Além disso, os clientes devem contribuir com as atividades de planejamento, como escrever histórias, definir prioridades e testes de aceitação e responder eventuais dúvidas sobre as funcionalidades desejadas.

- **Implantação Incremental:** quando substituir um sistema legado, comece substituindo somente algumas funcionalidades e gradualmente troque todo o sistema. O importante é manter o sistema funcionando e ter segurança durante a migração.

- **Contrato de Escopo Negociável:** contratos para desenvolvimento de software devem fixar o tempo, custos e a qualidade, mas manter o escopo negociável para garantir que a equipe esteja sempre trabalhando no que é mais importante para o cliente.

- **Pague pelo Uso:** essa prática sugere que o cliente pague por toda vez que receber uma versão do sistema. No entanto, ela pode gerar conflitos entre o desenvolvedor e o cliente, que desejará um menor número de versões, com o maior número de funcionalidades. Essa prática conecta o fluxo econômico ao desenvolvimento de softawre

provendo informações precisas e atualizadas para direcionar as melhorias no sistema (Marchesi, 2004).

(2) Time de Desenvolvimento e Fatores Humanos

- **Continuidade da Equipe:** os times de desenvolvimento que se mostraram eficientes, devem continuar trabalhando junto em outros projetos. A relação que eles compartilham durante o projeto é importante e não deve ser dispersada.
- **Diminuição da Equipe:** assim que o time tornar-se mais capaz e produtivo, mantenha o ritmo constante mas diminua o seu tamanho, retirando membros para formar novas equipes. Tentar fazer com que todos os membros pareçam ocupados pode possivelmente esconder um excesso de recursos na equipe (Sato, 2007).

(3) *Design*

- **Análise da Causa Inicial:** toda vez que for encontrado um defeito, elimine ele e suas causas. Dessa maneira, você não estará eliminando apenas o defeito, como também irá prevenir de ocorrer os mesmos defeitos novamente. Para auxiliar na descoberta da causa inicial do defeito devem ser utilizados testes de unidade e testes de aceitação.

(4) Codificação e Versões (*Releasing*)

- **Código e Testes:** somente o código e testes são artefatos permanentes e devem ser preservados. Os outros documentos podem ser gerados a partir do código e testes. Por esse motivo, a principal forma de documentação do XP é feita por meio de conversas e de artefatos de código e testes.
- **Código Compartilhado:** qualquer um no time de desenvolvimento deve ter a capacidade de modificar qualquer parte do sistema a qualquer momento. Ao invés de identificar responsáveis por cada parte do código, o time inteiro é responsável pelo sistema inteiro.
- **Repositório de Código Unificado:** complementa a prática de código compartilhado e vai além, dizendo que todo o código deve estar contido em um único repositório. Ramificações (*branches*) podem existir, mas devem ser evitadas.
- **Implantação Diária:** toda noite deve-se colocar um novo software em produção, para que os usuários possam usufruir de benefícios o quanto antes. Essa prática depende de uma baixa taxa de defeitos e de um processo automático de implantação, com habilidade para implantação incremental, inclusive voltar uma versão caso seja necessário.

Durante o processo de adoção do XP, é importante adaptar XP ao contexto local, possivelmente alterando algumas práticas, criando novas práticas, ou até mesmo utilizando práticas de outros métodos ágeis. Nesse contexto melhorias e adaptações são feitas por equipes ágeis. O Grupo AgilCoop (Cooperativa de Desenvolvimento Ágil de Software)[2] por exemplo, inseriu diversas práticas ao XP: *stand-up meetings* (originado do Scrum Schwaber e Beedle (2001)) que consiste em encontros diários para micro-planejamento e comunicação da equipe e retrospectivas (originada da Família Crystal) que consiste em uma atividade executada após o término de uma iteração ou *release*, na qual todos os envolvidos no processo param para pensar no próprio processo (o que foi bom, como foi efetuado, o que não queremos esquecer e também o que não foi muito bom e deseja-se melhorar). Outras práticas utilizadas em projetos XP são descritas com mais detalhes em (Freire, 2007; Sato et al., 2007a).

2.2.3 Papéis da Equipe de XP

Um projeto XP depende fortemente da prática do Time Completo, na qual diversas pessoas, com diferentes características e habilidades devem contribuir para o sucesso do projeto. Em um time maduro de XP, os papéis não são fixos, nem rígidos, sendo que uma pessoa pode assumir mais de um papel e os papéis podem ser revezados entre as pessoas.

Na área de gerência, gerentes e executivos devem atuar conectando pessoas e estimulando a equipe a melhoria contínua do projeto. Outros três papéis do XP desempenham funções relacionadas principalmente no processo de descrição das necessidades do cliente, os usuários, projetistas de iteração e escritores técnicos. Por fim, há três papéis (programadores, testadores e arquitetos de software) que são responsáveis principalmente pelo ciclo iterativo de desenvolvimento do XP. A seguir são descritos os principais papéis da equipe XP, que são os usuários, programadores, testadores e arquitetos de software:

- **Usuários:** ajudam a equipe XP a escrever e escolher histórias, além de tomar decisões de domínio durante o desenvolvimento.

- **Programadores:** fornecem estimativas de histórias e tarefas, quebram histórias em tarefas, escrevem testes e código-fonte, automatizam processos, sugerem alternativas e ajudam os clientes a criarem um plano possível de ser cumprido. Sato (2007) cita ainda dois papéis especiais para programadores. O primeiro é o *coach* (treinador), que, verifica e auxilia a aplicação das práticas técnicas do XP. O segundo é o *tracker* que coleta e compartilha dados importantes sobre o andamento do projeto e do processo (Sato, 2007). O *tracker* é responsável por criar e espalhar cartazes e gráficos na área de trabalho informativa. Juntamente com a área gerencial, programadores e testadores devem definir e acompanhar as métricas em relação ao processo e ao produto.

[2] AgilCoop - `http://ccsl.ime.usp.br/agilcoop`

- **Testadores:** ajudam os clientes a identificar falhas nos requisitos e colaborar com a escrita de testes de aceitação, definindo cenários de sucesso e erro em uma história. Eles também colaboram com o cliente, para que eles considerem todas as possibilidades quando eles prevendo o produto e escrevendo as histórias. Além disso, eles treinam os programadores em técnicas e ferramentas de teste. Caso algum programador encontre um problema de teste complicado, ele pode contar com a colaboração de um testador para que esse problema seja resolvido.

- **Arquitetos de software:** procuram e executam refatorações em larga escala, escrevem testes de carga para estressar a arquitetura e guiam a equipe para o *design* incremental. Além disso, arquitetos de software também direcionam a evolução da arquitetura, mostram formas de simplificar *designs* complexos e particionam o sistema com o objetivo de simplificá-lo.

2.3 Scrum

Scrum é um processo ou *framework* de gerenciamento de projetos de software. O Scrum (nome derivado de uma atividade que ocorre durante um jogo de *rugby*[3]) foi desenvolvido por Ken Shawaber, Jeff Sutherland e Mike Beedle nas décadas de 80 e 90, sendo apresentado oficialmente no *OOPSLA'95* (*International Conference on Object-oriented Programming Systems, Languages and Applications*) - (Schwaber, 1995)

O Scrum se concentra principalmente nos aspectos gerenciais do desenvolvimento de software em ambientes de negócio que mudam constantemente. Esse gerenciamento é feito por meio de iterações de duas semanas ou 30 dias (chamados *Sprints*), com monitoramento diário por intermédio de reuniões em pé (ou *stand-up meetings*) (Sato, 2007). Segundo Abrahamsson et al. (2003), o Scrum deixa os desenvolvedores livres para escolherem técnicas de desenvolvimento de software específicas, métodos e práticas para implementação do processo. O Scrum é idealmente indicado para projetos com mudança rápida e requisitos que mudam durante a execução do projeto e incorpora um conjunto de padrões de processo que enfatizam prioridades de projeto, comunicação e *feedback* freqüente do cliente.

Recentemente o processo Scrum tem se popularizado, e vem sendo adotado por grandes empresas de software como a Yahoo!, Microsoft, Intel, Google e Nokia (Marchenko e Abrahamsson, 2008). Uma pesquisa recente conduzida pela empresa Version One em parceria com diversas organizações da área de métodos ágeis com mais de 3000 entrevistados de 80países constatou que quase 50% dos entrevistados utilizam Scrum em suas empresas (Version One, 2008). Apesar da grande adoção de métodos ágeis por empresas, menos de 5% de estudos experimentais com evidências cientificamente válidas correspondem a estudos

[3]Um grupo de jogadores se forma ao redor da bola e os companheiros de equipe trabalham juntos para mover a bola pelo campo

com o processo Scrum (Dybå e Dingsøyr, 2008). Outro aspecto importante do Scrum é que ele vem sendo aplicado em organizações que procuram certificações de qualidade como o *CMMi* (Jakobsen e Sutherland, 2009) e também vem sendo utilizado em conjunto com as práticas do XP (Vriens e Barto, 2008).

2.3.1 Ciclo de Desenvolvimento e Práticas do Scrum

O Scrum enfatiza o uso de um conjunto de "padrões de processo de software", que são efetivos para projetos com prazos apertados, requisitos que mudam constantemente e criticalidade de negócio. Cada um desses padrões de processo define um conjunto de atividades de desenvolvimento. Essas atividades serão conduzidas pelo *Product Owner*, *ScrumMaster* e a Equipe Scrum. O *Product Owner* é um funcionário da organização que utilizará o software e garante que o produto entregue atenda aos anseios do patrocinador do projeto e o *ScrumMaster* guia o a equipe Scrum que representa todos os responsáveis por desenvolver as funcionalidades do projeto. As atividades do Scrum são apresentadas na **Figura 2.3** e descritas logo a seguir.

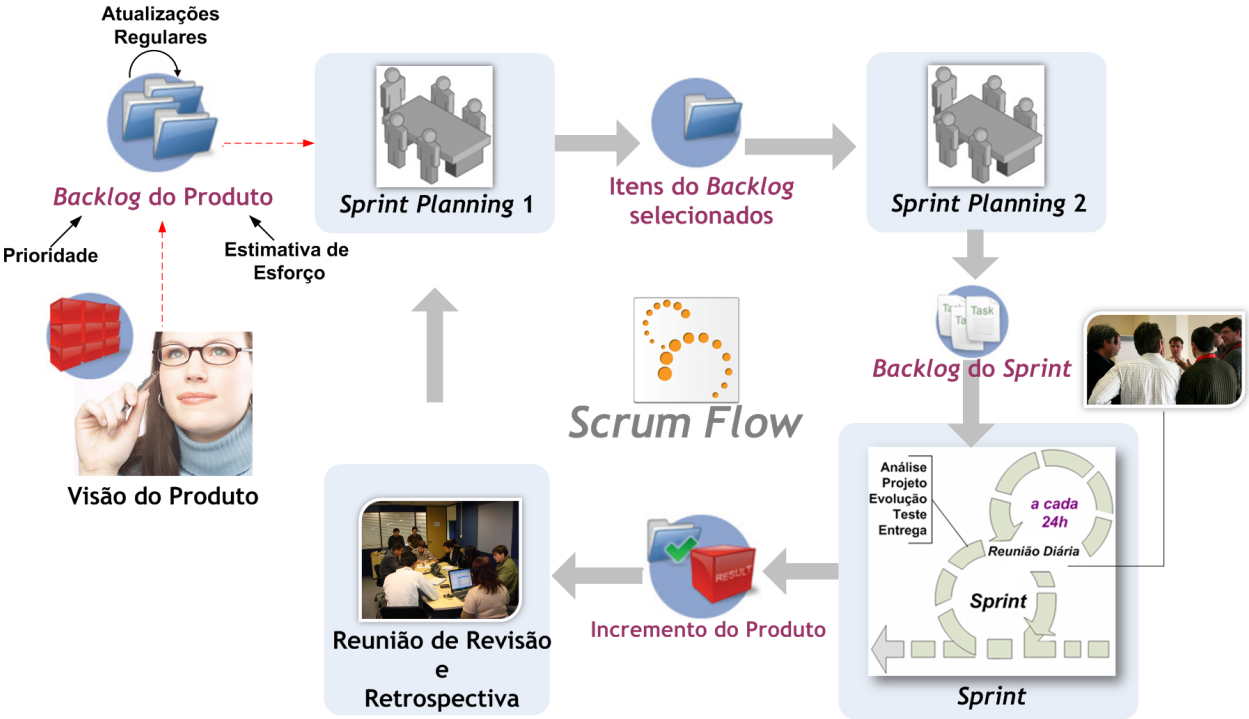


Figura 2.3: Ciclo de vida do processo Scrum

Visão do Produto e Definição do *Backlog* do Produto

O desenvolvimento inicia-se com a elaboração da visão do produto a ser desenvolvido, contendo suas características, premissas e restrições. Em seguida, o trabalho a ser feito em um projeto Scrum é listado em uma lista de pendências (*product backlog*), que é uma lista de todos os requisitos ou características do projeto que fornecem valor de negócio ao cliente. Esse

documento servirá como entrada para o processo iterativo e incremental de desenvolvimento.

Planejamento do *Sprint*

No começo de cada *Sprint* é feita uma reunião de planejamento do *Sprint* (*Sprint Planning Meeting*). Ela é dividida em duas partes e segundo um dos criadores do Scrum cada uma das reuniões deve demorar até quatro horas:

- **Reunião de Planejamento I (Seleção do *Backlog*):** o *Product Owner* apresenta os itens de *backlog* com maior prioridade ao time de desenvolvimento. É definido o objetivo do *Sprint*, e é discutido em seguida quantos desses itens podem ser selecionados para que seja possível entregar um produto funcional no fim do próximo *Sprint*. O time seleciona o quanto eles acreditam que podem fazer.

- **Reunião de Planejamento II (Seleção do *Backlog* do *Sprint*):** o time define a arquitetura e o projeto das funcionalidades que eles selecionaram e então definem o trabalho, ou tarefas, para desenvolver esta funcionalidade durante o *Sprint*. Essas tarefas também são estimadas em termos do tempo necessário para desenvolvê-las. A estimativa é informada conforme a performance em *Sprints* anteriores, capacidade de desenvolvimento para o próximo *sprint* e a complexidade das tarefas necessárias para alcançar o objetivo do *Sprint*. Caso a equipe constate que o tempo estimado no planejamento do *Sprint* não correspondeu ao tempo efetivamente gasto, esses poderão ser discutidos na retrospectiva do *Sprint* após o fim do *Sprint*,

Sprint de Desenvolvimento

Depois da reunião de planejamento, o time realiza a fase de desenvolvimento do produto que pode levar de uma a quatro semanas (*Sprint*). Durante o *Sprint* nenhuma interferência é permitida no *backlog do Sprint* por parte do *Product Owner* ou dos outros *stakeholders*. Mudanças, adições e priorização de requisitos só podem ser feitas por parte do *Product Owner* durante a reunião de planejamento. No entanto, caso mudanças no ambiente de negócios forcem mudanças nos requisitos, o *Product Owner*, juntamente com o *ScrumMaster*, podem cancelar o *sprint* atual imediatamente e realizar uma nova reunião de planejamento. Com isso, o Scrum mantém a visibilidade da produtividade realizada pelo time, que seria seriamente prejudicada caso requisitos pudessem ser modificados e adicionados durante um *Sprint*, ao mesmo tempo que provê um mecanismo de adaptação e correção para os requisitos do produto colocados pelo *Product Owner*.

A cada dia durante o *Sprint* são conduzidas breves reuniões diárias chamadas de *daily scrum* que ajudam o time a verificar o andamento do projeto. Essas reuniões são curtas e duram normalmente 15 minutos. Nelas os membros da equipe respondem a três questões básicas: (1) O que você fez desde a última reunião Scrum? (2) Você teve algum obstáculo?

e (3) O que você vai fazer antes da próxima reunião?. O líder da equipe, chamado de *ScrumMaster*, lidera a reunião e avalia as respostas de cada pessoa. Essas reuniões diárias ajudam a equipe a descobrir problemas potenciais tão cedo quanto possível. Elas também levam à "socialização do conhecimento", promovendo assim uma estrutura de equipe auto-organizada. Os impedimentos são registrados no *impediments backlog*, um artefato que registra todos os entraves encontrados pelo time. É responsabilidade do *ScrumMaster* trabalhar para resolver esses impedimentos e cuidar para que o time consiga desenvolver com a máxima eficiência.

Reunião de Revisão do *Sprint* e Retrospectiva

No final de cada *Sprint* o time demonstra a funcionalidade completa em uma reunião de revisão de *Sprint* (*sprint review meeting*). Essa reunião tem como objetivo melhorar o processo, as capacidades e aptidões do time e o desenvolvimento do produto para o próximo *sprint*. Nessa reunião os *stakeholders* podem identificar funcionalidades que não foram entregues, ou que não foram entregues conforme o esperado e requisitar que tais funcionalidades sejam colocadas no *backlog* do produto para priorização no próximo *sprint*. Por meio da apresentação, os *stakeholders* também podem identificar novas funcionalidades e requisitá-las.

O monitoramento do progresso do projeto é realizado por meio de dois gráficos principais: *product burndown* e *sprint burndown*. Esses gráficos mostram ao longo do tempo a quantidade de trabalho que ainda resta para realizar para o produto e a iteração, respectivamente. Com isso, observa-se a quantidade de trabalho que falta ser feita (em qualquer ponto) e o progresso do time do projeto em reduzir esse trabalho.

É feito também uma reunião de retrospectiva (*sprint retrospective*), que é um mecanismo importante que permite que todo a equipe se avalia continuamente e melhore durante a vida do projeto. Essa reunião é facilitada pelo *ScrumMaster* no qual o Time discute o *Sprint* recentemente concluído e determina o que pode ser feito para que o próximo *sprint* seja melhorado e mais produtivo. Tudo o que afeta o time que desenvolve o software é aberto para debate, e deve incluir: processos, práticas, comunicação e o ambiente. A revisão do *Sprint* se preocupa com "o que" o time vem desenvolvendo enquanto a retrospectiva se preocupa em discutir "como" está sendo desenvolvido.

2.3.2 Papéis do Scrum

No Scrum todas as responsabilidades de gerenciamento do projeto são divididas em três papéis: *Product Owner*, Equipe Scrum (*Scrum Team*) e *ScrumMaster*:

- ***Product Owner*:** Ele define a visão do produto e é responsável pelo retorno sobre o investimento do projeto (ROI). O Product Owner é geralmente um funcionário da organização que utilizará o software. Seu papel é garantir que o produto entregue

atenda aos anseios do patrocinador do projeto, priorizar quais funcionalidades devem ser entregues e quais agregam mais valor ao projeto. O *product owner* tem as seguintes responsabilidades: (1) definir o objetivo do *Sprint*, (2) gerenciar o *backlog* (define as funcionalidades do produto, decide a data da entrega da versão e o seu conteúdo), (3) agregar *idéias* de usuários, *stakeholders* e outras partes interessadas para formar uma visão única dos requisitos priorizados para o sistema, (4) é responsável pelo lucro gerado pelo produto (ROI) e priorização de funcionalidades de acordo com o valor para o mercado, (5) mudar funcionalidades e priorizar os itens do *backlog* a cada 30 dias, (6) aceitar ou rejeitar resultados do trabalho da equipe de desenvolvimento ao fim de cada *sprint*, juntamente com os *stakeholders*.

- **Equipe Scrum:** São todos os responsáveis por desenvolver funcionalidades. Os membros de uma equipe Scrum são multifuncionais, auto-organizáveis e auto-gerenciáveis. Eles são responsáveis por desenvolver o produto alvo do projeto, de acordo com suas próprias descrições para alcançar os objetivos estabelecidos. Podem fazer parte da equipe, analistas, *designers*, pessoal de qualidade, desenvolvedor, pessoas responsáveis pela documentação, além de outros especialistas que a equipe necessitar para transformar requisitos em produto. Geralmente são equipes pequenas (de até 10 pessoas), sendo que equipes grandes geralmente se comportam como várias equipes pequenas. A equipe Scrum tem as seguintes responsabilidades: (1) seleciona o objetivo da iteração e especificam os resultados do trabalho, (2) organiza-se a si mesma e o seu trabalho, (3) demonstra os resultados do trabalho ao *Product Owner* e aos *stakeholders*.

- ***ScrumMaster*:** É responsável por guiar o time, por meio do conhecimento do processo, intermediando negociações entre o *product owner* e a equipe. Seu principal papel é ensinar e acompanhar a utilização do Scrum e também remover impedimentos do projeto. O ScrumMaster é um líder e facilitador responsável por: (1) melhorar a vida e a produtividade dos desenvolvedores, (2) possibilitar a cooperação próxima entre todos os papéis e funções, removendo barreiras entre eles, (3) proteger a equipe de interferências externas e remove "impedimentos", (4) garantir que o processo está sendo seguindo, (5) conduzir as reuniões diárias (*daily scrum*) e revisões do *sprint* (*sprint review*), (6) convidar as pessoas certas para as reuniões diárias, revisões de iteração e reuniões de planejamento, (7) ensinar o cliente como maximizar o ROI e atender aos seus objetivos por meio do Scrum, (8) procurar formas de melhorar as práticas de engenharia e ferramentas.

2.4 *Lean Software Development*

Lean Software Development é uma adaptação de princípios do sistema de produção da Toyota, no qual *Mary* e *Tom Poppendieck* traçam paralelos entre valores e práticas do Lean com o desenvolvimento de software (Poppendieck e Poppendieck, 2003, 2006). O Lean pode ser sintetizado em 7 princípios, sendo que cada princípio indicam práticas que funcionam como guias para que estes princípios sejam aplicados:

Elimine Desperdícios: a equipe deve aprender a identificar desperdícios, pensando sempre em desenvolver software que agregue valor ao cliente. Neste contexto, são citados 7 desperdícios em um projeto de software: (1) trabalhos incompletos ("em progresso"), (2) processos a mais, (3) funcionalidades a mais, (4) troca de tarefas (desenvolvedores participando de vários projetos), (5) espera (atrasos em atividades), (6) movimentação de integrantes do projeto e artefatos, e (7) defeitos. Além disso, é sugerida a utilização de um modelo de fluxo de valores durante as fases de desenvolvimento, que demonstram o tempo efetivo de cada atividade e os seus atrasos.

Crie conhecimento: para melhoria constante do processo a equipe deve se preocupar com instrumentos informativos que forneçam o *feedback* constante do andamento do projeto no fim de cada iteração, que devem ser curtas. Para resolução de problemas é sugerido a utilização do ciclo PDCA (*Plan-Do-Check-Act*) que deve identificar o problema, procurar a raiz do problema, propor uma solução, especificar os resultados esperados, implementar a solução, verificar os resultados, analisar e adaptar os padrões.

Adie comprometimentos: decisões irreversíveis devem ser tomadas o mais tarde possível (*last responsible moment*), além disso a equipe deve saber quando essas decisões devem ser tomadas. Geralmente o momento certo para tomada de decisão é quando a equipe tiver informações suficientes. Uma prática sugerida pelo Lean para tomada de decisão é o "*Design* baseado em conjunto (*set-based*)", que deve experimentar diversas soluções para encontrar a melhor.

Entregue rápido: devem ser estabelecidas práticas para que o ciclo de entregas seja mais rápido. Utilize sistemas *pull* para o software, no qual a produção é priorizada conforme a demanda atual dos clientes. É interessante também que a equipe utilize irradiadores de informação (*Kanban*[4] ou Reuniões Scrum) para que seja possível que a equipe possa ser auto-dirigida. A equipe deve estabelecer pequenas unidades de trabalho, limitando o trabalho à sua capacidade.

[4]O Kanban utiliza controles visuais (cartões) para controlar o fluxo de funcionalidades (histórias) em desenvolvimento, demonstrando o seu *status* (por ex. não iniciado, em andamento, impedimento, em publicação ou concluido).

Respeite as Pessoas: um time engajado, motivado e pensante fornece uma grande vantagem competitiva. O time também deve ser multidisciplinar. A liderança é outro aspecto essencial. Líderes devem direcionar os objetivos do time e possibilitar a sua motivação. Deve-se pensar na educação e auto-aprendizado do time, e também deve ser oferecido um treinamento contínuo para o time e também para os líderes.

Visualize o Todo: é preciso olhar para o processo todo, se focando em todo o fluxo de valores (da concepção ao dinheiro, e de um pedido do cliente ao software entregue). Quando forem detectados problemas, é preciso resolver a sua causa e não resolver os seus sintomas. A equipe deve focar-se no produto como um todo e não somente no software. A utilização de medidas é muito importante para acompanhamento do progresso do desenvolvimento, métricas devem ser medidas a nível de time e não de indivíduos.

Construa com Integridade: a integridade de um sistema é alcançada quando a totalidade do produto atinge um equilíbrio de funcionamento, usabilidade, confiabilidade, economia e deixam os usuários satisfeitos. Do ponto de vista conceitual, essa integridade significa que o sistema é coeso e alcança um balanço efetivo de flexibilidade, manuntenabilidade, eficiência e rapidez. Essa integridade é alcançada por meio de um fluxo excelente e detalhado de informações entre o cliente, usuários e equipe de desenvolvimento. O Lean sugere três práticas para que a integridade seja alcançada: o *refactoring*, que faz com que o código do sistema mantenha-se simples, o *design* que deve ser construído de forma incremental e os testes, que colaboram com a comunicação e *feedback* da equipe de desenvolvimento e facilitam a manutenção do software.

2.5 Outros Métodos Ágeis

Nessa seção são descritos resumidamente outros métodos ágeis com algum destaque na comunidade de desenvolvedores.

Dynamic System Development Method (DSDM)

Foi desenvolvido inicialmente por um consórcio iniciado em janeiro de 1994, formado por organizações britânicas, podendo ser visto como o primeiro método de desenvolvimento ágil de verdade (Abrahamsson et al., 2003). A intenção do grupo era criar um modelo independente para o desenvolvimento rápido de aplicações baseando-se nas idéias do *Rapid Application Development* (RAD) e no desenvolvimento iterativo e incremental (Stapleton, 2003). Desde então o modelo vem sendo desenvolvido e refinado, mas os conceitos básicos têm permanecido (DSDM, 2009).

O objetivo principal do DSDM visa a entrega rápida de soluções de qualidade, no prazo e orçamento previstos (DSDM, 2009). Para atingir esse objetivo, de forma semelhante ao mé-

todo XP, o DSDM descreve valores e princípios que servirão como guia ao desenvolvimento. Os princípios baseiam-se no envolvimento do usuário, fortalecimento da equipe de projeto, entregas frequentes, atendimento das necessidades atuais do negócio, desenvolvimento iterativo e incremental, permissão para reverter requisitos, escopo em alto-nível que é fixado depois que o projeto inicia-se, testes durante todo o ciclo de vida, comunicação efetiva e eficiente. Além dos testes serem conduzidos durante todo o ciclo de vida, eles devem ser executados o mais cedo possível, inclusive inspecionando documentos de entrevista por meio de checagem cruzada com grupos controlados ou técnicas similares (Voigt, 2004).

O ciclo de vida DSDM define três ciclos iterativos diferentes, precedidos por duas atividades adicionais de ciclo de vida. O método começa com duas atividades que são o estudo de viabilidade e o estudo de negócio que define as prioridades do negócio por meio de um documento de requisitos e uma arquitetura inicial. O restante do processo é formado por três fases: (1) iteração do modelo funcional: define um modelo funcional (análise e protótipo), uma lista priorizada de funcionalidades e documentos de revisão do protótipo funcional, requisitos não-funcionais e documento de riscos do projeto, além de alguns protótipos iniciais; (2) iteração de projeto e construção: é feito o projeto e a implementação de forma iterativa, na qual os protótipos da fase anterior são completados, combinados e testados; (3) implementação: na ultima fase é feito a implementação para entrega, orientações e aprovação do usuário, implantação do produto e revisão de negócio

O DSDM possui um conjunto de técnicas: (1) *timeboxing*: divisão do projeto em porções, cada um com orçamento fixo e data de entrega. Cada porção do *timeboxing* terá um número de requisitos que são selecionados e priorizados de acordo com o princípio *MoSCoW* (*Must have*, *Should have*, *Could have* e *Want to have*; (2) protótipos: no DSDM poderão ser desenvolvidos protótipos de negócio para checar a evolução do sistema, para checar a usabilidade, performance, e para testar possíveis soluções/técnicas; (3) *workshops*: utilizados para possibilitar que o cliente esteja envolvido com o projeto, ajudando por exemplo, na definição de requisitos, testes de aceitação, *design* e informações de negócio.

Feature Driven Development (FDD)

O *Feature Driven Development* (Desenvolvimento Guiado por Funcionalidades) é um processo de desenvolvimento de software interativo e incremental, criado em 1997 por Peter Coad e Jeff De Luca em um grande projeto em Java para o *United Overseas Bank*, em Singapura. O processo combina o desenvolvimento guiado por modelos (*model-driven*) e o desenvolvimento ágil, com ênfase em um modelo de objetos inicial, divisão do trabalho em características, e um projeto e desenvolvimento iterativo para cada funcionalidade. O FDD não cobre todo o processo de software, sendo focado apenas nas fases de concepção e projeto e desenvolvimento por funcionalidade (Palmer e Felsing, 2001).

Segundo Nascimento (2007), o que diferencia o FDD da maioria dos métodos ágeis é o fato de ele ser altamente escalável, podendo ser aplicado a projetos com equipes numerosas,

que podem chegar a centenas de indivíduos. Para dar suporte a escalabilidade, o FDD enfatiza a criação inicial de um modelo global de projeto e uma lista de funcionalidades. O FDD utiliza um conjunto de boas práticas já conhecidas da engenharia de software (Abrahamsson et al., 2002), como o modelo de objetos do domínio que fornece a visão geral do projeto, inspeções de projeto e código, testes de unidade, gerência de configuração, *builds* regulares e a visibilidade do progresso e resultados por meio de relatórios de progresso do projeto.

Família *Crystal*

A Família Crystal é um conjunto de métodos para times co-alocados propostos por Alistair Cockburn (Cockburn, 2002, 2006) que podem ser adotados para características específicas de projeto (projetos de diferentes tamanhos e complexidades). Cada membro da família possui elementos centrais: entregas frequentes, reflexão e comunicação, além de papéis, padrões de processo, ferramentas, produtos de trabalho e práticas específicas a cada uma. Podendo inclusive ser adotadas práticas de outros métodos, como o XP ou Scrum (Cockburn, 2002). O Crystal sugere a escolha de um método de cor apropriada (*Clear*, *Yellow*, *Orange*, *Red*, *Blue*) de acordo com o tamanho e criticidade do projeto, sendo que projetos grandes precisam de mais coordenação e métodos mais pesados (Cockburn, 2002).

Um conjunto de práticas auxiliam no desenvolvimento do projeto e do software. Nascimento (2007) descreve resumidamente cada uma dessas práticas que podem ser utilizadas pela família Crystal: *staging* (define os requisitos que serão desenvolvidos no próximo incremento), monitoramento do progresso, revisão e inspeção dos requisitos, fluxo e paralelismo (garante que atividades possam ocorrer em paralelo), estratégia de diversidade holística (estratégias para tornar a equipe multifuncional), técnica para refinamento do método (aprendizagem referente a um incremento), parecer do usuário (para garantir que os requisitos foram desenvolvidos corretamente), *workshops* de reflexão (que tem o intuito de verificar possíveis problemas e soluções) e testes de regressão automatizados. Além disso, práticas de outras métodos podem ser integrados.

Adaptative Software Development (ASD)

O ASD foi proposto por Jim Highsmith, na mesma época do surgimento do método XP. O método encoraja o desenvolvimento iterativo e incremental, com constantes prototipações e a tentativa de promover um paradigma adaptativo no software (Highsmith, 2002). O ciclo de vida do ASD é dedicado à contínua aprendizagem e orientado a mudanças e reavaliações, olhando para um futuro incerto, com intensa colaboração entre os desenvolvedores, gerentes de projeto e clientes. Além disso, o ciclo de desenvolvimento possui seis características básicas que demonstram a natureza adaptativa do ASD: foco na missão, desenvolvimento baseado

em funcionalidades, iterações, *timeboxing*[5], análise de riscos e tolerância às mudanças. O ASD propõe três fases não lineares que podem ser sobrepostas: especulação, colaboração e aprendizagem, na qual a equipe por meio de iterações curtas cria o conhecimento cometendo pequenas falhas, causadas por falsas premissas, e corrigindo-as aos poucos.

Um aspecto importante no ASD é o ciclo colaborativo que é apoiado por sessões de grupo estruturadas - JAD (*Joint Application Design*) na qual o líder de reunião guia usuários e analistas para que todos projetem o sistema juntos. Todos os envolvidos com o projeto devem colaborar para solução dos problemas técnicos, requisitos de negócio e rápidas tomadas de decisão. No ASD também são utilizados instrumentos para prover uma área de trabalho informativa. No fim do desenvolvimento das funcionalidades são executadas revisões de qualidade para possibilitar a aprendizagem para o próximo ciclo. Nessas revisões, devem ser observadas a qualidade do produto na visão do usuário final e na visão técnica, a performance da equipe responsável pelas entregas, as práticas que a equipe utiliza e o estado do projeto.

2.6 Considerações Finais

Neste capítulo foram descritos os métodos ágeis de desenvolvimento, principalmente do ponto de vista de valores e princípios comuns a esses métodos, assim como suas práticas individuais. Também foi descrito o relacionamento dessas práticas com o ciclo de vida de cada método. Optou-se por descrever com maiores detalhes o método XP, Scrum e Lean, por esses serem os métodos mais aplicados no contexto atual da área de desenvolvimento ágil, além de terem relação com a utilização de métricas de software seja por meio de suas práticas ou pelo seu processo e papéis envolvidos no projeto de software.

Os conhecimentos adquiridos por intermédio deste estudo detalhado a respeito de métodos ágeis de desenvolvimento serviram como base para o estudo a respeito da atividade de teste de software, podendo descrever de forma geral quais os processos e práticas envolvidos nessa atividade, que são essenciais para um bom andamento do projeto e para a qualidade do produto final.

No próximo capítulo é apresentada uma visão geral da área de teste de software tradicional, com a descrição de fases, técnicas e critérios de teste. Como assunto principal do capítulo são descritas as principais estratégias, técnicas e ferramentas utilizadas pela atividade de teste executada no contexto de métodos ágeis e também são apresentados estudos experimentais na área de testes ágeis.

[5] *Timeboxing* é uma técnica de gerenciamento de tempo comum no planejamento de projetos (tipicamente para projetos de software), aonde o plano é dividido em períodos de tempo, sendo que cada parte possui os itens que serão entregues, a data de entrega e o orçamento.

Capítulo

3

Teste de Software

Teste de Software, segundo Bertolino (2007), é um termo circundado por um amplo espectro de diferentes atividades, do teste de pequenos pedaços de código pelo desenvolvedor (teste de unidade) até a validação do consumidor em grandes sistemas de informação (teste de aceitação). Essas atividades também podem abranger o monitoramento em tempo de execução de uma aplicação em rede ou orientada a serviços. Em vários estágios, os casos de teste devem ser planejados de acordo com diferentes objetivos, expondo incompatibilidade com os requisitos do usuário, ou estimando a conformidade com um padrão específico, ou avaliando a robustez para condições de *stress* ou entradas maliciosas, ou medindo outros atributos, como performance, usabilidade, ou estimando a confiabilidade operacional, entre outras coisas.

A história da engenharia de software tem demonstrado uma evolução progressiva de sistemas com limites específicos e requisitos estáticos em um ambiente fechado, para sistemas flexíveis em contínua evolução, em nível de produto e de processo (Baresi et al., 2006). A área de teste de software também tem evoluído, preocupando-se com a aplicação de testes em sistemas cada vez mais dinâmicos e reutilizáveis com propostas para o aumento de qualidade e produtividade de software. Como exemplos têm-se o teste de sistemas críticos e sistemas embarcados, teste de linha de produtos de software, serviços web, programas orientados a aspectos e teste de componentes.

Nesse capítulo são apresentados conceitos básicos relacionados ao teste de software tradicional e também estratégias, práticas e ferramentas utilizadas em projetos que utilizam métodos ágeis. Na Seção 3.1 é apresentada uma visão geral da área de teste de software, como é realizado o processo de teste de software e quais as suas fases, e por fim são descritas

brevemente as técnicas de teste funcional, estrutural, baseado em defeitos. Na Seção 3.2 são apresentadas estratégias e práticas de teste utilizadas em projetos ágeis e são apresentados os resultados de uma revisão sistemática que teve como objetivo verificar como a atividade de teste vem sendo tratada em métodos ágeis (Vicente et al., 2009). Na Seção 3.3 são apresentadas algumas ferramentas de teste de software utilizadas em projetos ágeis e na Seção 3.4 são apresentadas as considerações finais do capítulo.

3.1 Fundamentos do Teste de Software Tradicional

Do ponto de vista dos fornecedores de software, qualidade não é mais um fator de vantagem no mercado, mas uma condição necessária e pode-se dizer indispensável para que seja possível competir com sucesso. A atividade de teste de software desempenha um papel central em atividades de garantia de qualidade de software (GQS) (Tian, 2005) e tem o objetivo de revelar a presença de erros ou defeitos no produto e aumentar a confiança de que o produto esteja correto (Myers et al., 2004). O teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa que está sendo testado falhe.

A atividade de testes (conduzida da forma incremental) consiste em cinco etapas executadas após o planejamento dos testes, conforme ilustrado na Figura 3.1: (1) projeto dos casos de teste, (2) preparação dos dados de teste, (3) execução dos casos de teste e (4) avaliação dos testes por meio da comparação do resultado dos testes com o resultado esperado (Myers et al., 2004; Pressman, 2006; Sommerville, 2006). Os objetos submetidos ao teste podem incluir programas, especificação de requisitos e de projeto, estruturas de dados e quaisquer outros artefatos conceitualmente executáveis utilizados na implementação da aplicação. Esses objetos representam uma função (casos de teste) que descreve a relação entre um elemento de entrada (chamado de elemento do domínio) e um elemento de saída. Os casos de teste são feitos a partir da comparação do comportamento desta função com o comportamento esperado.

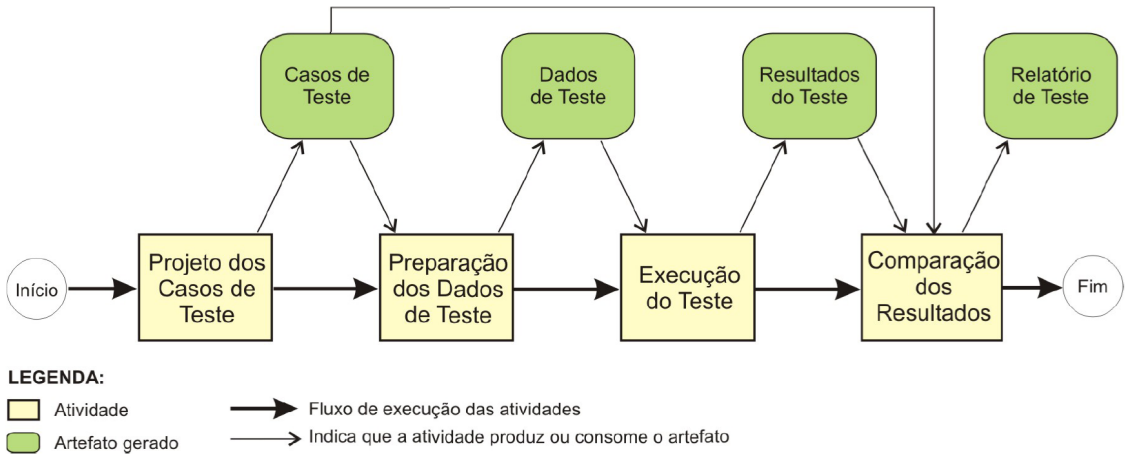


Figura 3.1: Processo de Teste de Software (Nakagawa, 2006; Sommerville, 2006)

Outro ponto bastante importante na atividade de teste é a automatização das técnicas e critérios de teste (que serão discutidos nas próximas seções), com o objetivo de contribuir com a qualidade e produtividade dos testes. Além disso, este apoio à atividade de testes segundo Barbosa (2004) viabiliza a realização de estudos empíricos, auxilia a condução de testes de regressão e também apoia o processo de ensino e aprendizado envolvendo a aplicação prática de conceitos de teste.

Na Seção 3.1.1 serão descritas as fases do teste de software sob a abordagem incremental utilizada em projetos tradicionais. Na Seção 3.1.2 é apresentada uma visão geral sobre as técnicas e critérios de teste de software propostos para projetos que utilizam métodos tradicionais, mas que no entanto também são utilizados em projetos ágeis.

3.1.1 Fases de Teste de Software

Em projetos tradicionais de desenvolvimento a atividade de teste ocorre basicamente por meio de duas estratégias: o sistema é testado somente quando ele está totalmente construído na esperança de encontrar erros ou utiliza-se a estratégia de teste incremental. A estratégia de teste incremental ocorre da seguinte forma (Binder, 1999; Myers et al., 2004; Pressman, 2006):

- **Teste de Unidade:** tem como foco as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. O objetivo desta fase é revelar defeitos de lógica e implementação em cada unidade, fornecendo evidências de que a unidade testada funciona adequadamente de forma isolada. Como cada unidade é testada separadamente, o teste de unidade pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio desenvolvedor, sem a necessidade de dispor-se do sistema totalmente finalizado (Delamaro et al., 2007b).

- **Teste de Integração:** após serem testadas as unidades individualmente, o teste de integração deve ser realizado, sendo que a ênfase é dada na construção da estrutura do sistema. Nesta fase, as diversas partes do software são integradas e executadas, verificando se a interação entre elas funciona de maneira adequada e não leva a erros. Para verificar esta iteração é necessário um grande conhecimento das estruturas de código internas e das iterações existentes entre as partes do sistema.

- **Teste de Sistema:** é iniciado após se ter o sistema completo, com todas as suas partes integradas. O objetivo é verificar se as funcionalidades especificadas nos documentos de requisitos estão todas implementadas, combinando-se todos os elementos do sistema. Aspectos de correção, completude e coerência devem ser explorados, bem como requisitos não funcionais como segurança, performance e robustez.

- **Teste de regressão:** é realizada durante a manutenção do software. A cada modificação efetuada no sistema, após a sua liberação, corre-se o risco de que novos defeitos

sejam introduzidos. Por esse motivo, é necessário, após a manutenção, realizar testes que mostrem que as modificações efetuadas estejam corretas, ou seja, que os novos requisitos implementados (se for esse o caso) funcionem como o esperado e que os requisitos anteriormente testados continuam válidos (Delamaro et al., 2007b).

No contexto de sistemas orientados a objeto (OO), (Vincenzi, 2004) apresenta algumas variações nas fases descritas acima. Em cada uma das fases são executados testes conforme a menor unidade a ser testada em um programa OO, que pode ser o método (Harrold e Rothermel, 1994) ou a classe (Binder, 1999). Considerando o método como menor unidade e a classe a qual o método pertence o *driver* do método, a atividade de testes OO poderia ser dividida da seguinte maneira (Vincenzi, 2004):

- **Teste de Unidade:** teste de cada método, testado de forma isolada, chamado de teste *intra-método*.
- **Teste de Integração:** considerando uma única classe é possível dividir o teste de integração em três tipos de teste: (1) no teste *inter-método* deve ser testada a integração entre os métodos de uma mesma classe; (2) no teste *intra-classe* são testadas interações entre métodos públicos fazendo chamada a esses métodos em diferentes seqüências, identificando-se possíveis seqüências de ativação de métodos inválidos que levem um objeto a um estado inconsistente; (3) no teste *inter-classe* o mesmo conceito de invocação de métodos públicos em diferentes seqüências é utilizado, no entanto esses métodos públicos não necessitam estar na mesma classe.
- **Teste de Sistema:** executado após os testes de unidade e integração terem sido realizados, e também com o sistema integrado como um todo. Essa fase do teste é baseado em critérios funcionais.

3.1.2 Técnicas e Critérios de Teste

A escolha do conjunto de técnicas de teste deve levar em conta as restrições de qualidade, custo, prazo e recursos no desenvolvimento de um produto em particular. Como descrito anteriormente, as técnicas e critérios de teste fornecem uma abordagem sistemática e teoricamente fundamentada para se conduzir e avaliar a qualidade do teste de software.

Critérios de teste selecionam e avaliam casos de teste com o intuito de revelar a presença de defeitos ou estabelecer um nível elevado de confiança na correção do produto quando erros não são revelados (Maldonado e Fabbri, 2001). Esses critérios são geralmente derivados a partir de quatro técnicas: funcional, estrutural, baseada em defeitos e baseada em estados, que diferem pela abordagem de teste utilizada para gerar e avaliar casos de teste (Zhu et al., 1997). A seguir serão apresentadas as técnicas funcional, estrutural e baseada em defeitos

juntamente com os respectivos critérios[1].

Técnica de Teste Funcional

O teste funcional utiliza a especificação de requisitos do programa ou do componente para derivar os seus casos de teste, sendo que o conteúdo do software é desconhecido, sendo possível visualizar apenas as respostas produzidas a partir de dados de entrada fornecidos (Fabbri et al., 2007; Maldonado e Fabbri, 2001).

Fabbri et al. (2007) apresentam dois problemas da utilização de teste funcional. O primeiro problema é referente a especificações ausentes ou mesmo incompletas tornarão difícil a aplicação dos critérios funcionais. O segundo problema é referente à limitação dos critérios de teste funcional, que não garantem que partes essenciais ou críticas do produto em teste sejam exercitadas. Por outro lado, os critérios funcionais podem ser aplicados em todas as fases de testes e em produtos desenvolvidos com qualquer paradigma de programação, inclusive componentes caixa preta, pois não levam em consideração detalhes de implementação (Vincenzi et al., 2003).

Os critérios mais conhecidos da técnica de teste funcional são o particionamento em classes de equivalência (Binder, 1999), análise do valor limite (Myers et al., 2004), grafo causa-efeito (Howden, 1986) e *error-guessing* (Fabbri et al., 2007). Além desses, outros critérios são utilizados para o teste funcional, por exemplo: teste funcional sistemático, *syntax testing*, *state transition testing* e *graph matrix*. Os critérios de particionamento em classes de equivalência, análise do valor limite, grafo causa-efeito são descritos abaixo:

- **Particionamento em Classes de Equivalência:** consiste em particionar o domínio de entrada de um programa em classes de equivalência (válidas e inválidas) a partir das condições de entrada de dados identificadas na especificação. O número de classes de equivalência deve ser finito de modo que o teste de um elemento representativo de uma classe seja equivalente ao teste de qualquer outro elemento da mesma classe. Assim se um caso de teste detectar um erro, todos os outros casos de teste da mesma classe irão detectar o mesmo erro. De forma equivalente, se um casos de teste não detectar nenhum erro, espera-se que nenhum outro casos de teste na mesma classe o revele (Lemos, 2005). Esse critério reduz significativamente o número de casos de teste em relação ao teste exaustivo, sendo mais adequado para o teste de produtos com domínios de entrada divididos em intervalos ou conjuntos.

- **Análise do Valor Limite:** é um critério funcional que complementa o particionamento em classes de equivalência, exigindo que existam casos de teste para testar os limites de cada classe de equivalência (um ponto abaixo do limite,o limite e um ponto acima do limite). Os limites concentram um grande número de erros.

[1]Outras técnicas de teste são utilizadas em projetos de software, como por exemplo teste baseado em estados ou testes combinatoriais, no entanto estas técnicas não serão descritas nesse trabalho.

- **Grafo de Causa-Efeito:** explora a fraqueza dos outros critérios funcionais que não combinam as circunstâncias da entrada, mas apenas exigem o teste de um subconjunto delas. Primeiramente para geração de casos de teste deve-se identificar as possíveis condições de entrada (causas) e possíveis ações (efeitos) do programa. Em seguida é construído um grafo relacionando as causas e efeitos identificados. Este grafo é convertido em uma tabela de decisão da qual são gerados os casos de teste (Maldonado e Fabbri, 2001).

Técnica de Teste Estrutural

A técnica estrutural (caixa-branca) é vista como complementar à técnica funcional e baseia-se na estrutura de um programa para derivar os seus casos de teste. Em geral, os critérios dessa técnica utilizam um tipo de representação chamado de grafo de fluxo de controle (CFG) ou grafo de programa, que mostra o fluxo lógico do programa. Um grafo de fluxo de controle é um grafo dirigido, com um único nó de entrada e um único nó de saída, no qual cada arco representa um possível desvio de um bloco para outro. Cada bloco tem as seguintes características: uma vez que o primeiro comando do bloco é executado, todos os demais são executados sequencialmente; e não existe desvio de execução em nenhum comando dentro do bloco (Domingues, 2002). Por intermédio do grafo de fluxo de controle podem ser escolhidos os componentes que devem ser executados.

Os critérios baseados no fluxo de controle estruturais utilizam apenas características de controle de execução do programa, como comandos ou desvios, para derivar os requisitos de teste necessários. Os critérios mais conhecidos desta classe são (Barbosa, 2004; Domingues, 2002; Lemos, 2005; Myers et al., 2004):

- **Todos-Nós:** exige que cada comando do programa seja executado ao menos uma vez (também chamado de cobertura de comandos - *statement coverage*), ou seja, que cada nó do grafo de programa seja coberto. Este critério é considerado fraco, pois há a necessidade de se cobrir cada sentença, no entanto na prática não revela mesmo os erros mais simples (Myers et al., 2004).

- **Todos-Arcos (Arestas):** requer que cada desvio do fluxo de controle do programa seja exercitado pelo menos uma vez, ou seja, que cada arco do grafo seja coberto.

- **Todos-Caminhos:** exige que todos os caminhos possíveis do programa sejam executados. Este critério geralmente é considerado impraticável, pois mesmo em um programa simples, pode ser astronomicamente grande (possivelmente infinito).

Os critérios baseados em fluxo de dados utilizam a análise do fluxo de dados como fonte de informação para derivar requisitos de teste. A derivação de casos de teste destes critérios baseiam-se nas associações entre a definição de uma variável e seus possíveis usos subsequentes. A definição de uma variável ocorre toda vez que um valor é atribuído a ele. O uso de

uma variável pode ser de dois tipos: quando a variável é usada em uma computação (*c-uso*) e quando a variável é usada em uma condição seu uso é predicativo (*p-uso*).

Uma motivação para introdução destes critérios, segundo Barbosa et al. (2007), foi a indicação de que, mesmo para programas pequenos, o teste baseado unicamente no fluxo de controle não era eficaz para revelar a presença mesmo de defeitos simples e triviais. Nesse sentido, a introdução destes critérios procurou estabelecer uma hierarquia entre os critérios *Todas-Arestas* e *Todos-Caminhos*.

Dentre os critérios baseados em fluxo de dados destacam-se os critérios de Rapps e Weyuker (1985, 1982). Os principais critérios desta família são (Barbosa et al., 2007):

- **Todas-Definições:** requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por uso computacional ou por uso predicativo.

- **Todos-Usos:** requer que todas as associações entre uma definição e seus subseqüentes usos (c-usos e p-usos) sejam exercitadas pelos casos de teste, por meio de pelo menos um caminho livre de definição (um caminho em que a variável não é definida). Os critérios *Todos-p-Usos*, *Todos-p-Usos/Alguns-c-Usos*, *Todos-c-Usos/Alguns-p-Usos* representam variações do critério *Todos-Usos*.

- **Todos-Du-Caminhos:** requer que toda associação entre uma definição de variável e subseqüentes *c-usos* ou *p-usos* dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

A família de critérios *potenciais-usos* propostos por Maldonado (1991) corresponde à família de critérios executáveis, obtida pela eliminação dos caminhos e associações não executáveis. A família de critérios potenciais-usos requerem que caminhos livres de definição a partir da definição de uma determinada variável sejam executados, independentemente de ocorrer um uso desta variável neste caminho.

Os critérios de teste estrutural que foram apresentados nesta seção são utilizados no teste de programas procedimentais, e também são utilizados no teste intra-método de software OO. Em (Vincenzi, 2004) são apresentados quatro critérios baseados em análise de fluxo de controle e quatro baseados em análise de fluxo de dados para programas OO.

Diversas ferramentas implementam os critérios de teste apresentados para verificar a porcentagem de cobertura do código-fonte em projetos de software. A maioria das ferramentas implementam critérios de fluxo de controle, sendo que poucas ferramentas implementam critérios de fluxo de dados. A utilização dessas ferramentas cobertura de código são bastante importantes para que a equipe tenha um guia para a qualidade dos casos de teste de unidade que são desenvolvidos utilizando a estratégia TDD. A ferramenta *JaBUTi* (*Java Bytecode Understanding and Testing*) (Vincenzi et al., 2005) será utilizada no estudo de caso que foi conduzido neste trabalho, mais detalhes sobre a ferramenta serão descritos na Seção 3.3

Técnica de Teste Baseada em Defeitos

A técnica de teste baseada em defeitos utiliza informações sobre os erros mais frequentes cometidos pelo programador ou projetista no processo de desenvolvimento de software (Budd, 1981; DeMillo et al., 1988). Inicialmente utilizada para teste de programas em nível de unidade e mais recentemente para testes de integração (Delamaro et al., 2001). Os critérios típicos do teste baseado em defeitos são utilizados para medir a qualidade do conjunto de casos de teste de acordo com a sua habilidade para detectar defeitos (Zhu et al., 1997). Dentre eles pode-se citar:

- **Semeadura de Defeitos:** neste critério, uma quantidade conhecida de erros são introduzidos artificialmente no programa, ou seja, defeitos artificiais são introduzidos aleatoriamente no programa a ser testado e de modo desconhecido para o testador. Após o teste, do total de erros encontrados verificam-se quais são naturais e quais são artificiais. Usando estimativas de probabilidade, o número de erros naturais ainda existentes no programa pode ser calculado (Budd, 1981).

- **Análise de Mutantes:** é um critério que utiliza um conjunto de programa ligeiramente modificados, obtidos a partir de um determinado programa P, para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é encontrar um conjunto de casos de teste capaz de revelar as diferenças de comportamento existentes entre P e seus mutantes. (Delamaro, 1993; DeMillo et al., 1988)

No contexto de teste de especificações, a análise de mutantes já foi utilizada em testes de Redes de Petri, *Statecharts*, Máquinas de estados finitos e Estelle (Delamaro et al., 2007a). Recentemente, pesquisadores tem investigado o uso do teste de mutação para o paradigma orientado a objetos (Ma et al., 2005), e também para programas orientados a aspectos (Ferrari et al., 2008). Algumas ferramentas para o apoio ao teste de mutação foram desenvolvidas para algumas linguagens de programação: Mothra e Proteum (Delamaro e Maldonado, 1996) desenvolvidas para Fortran e C respectivamente, além das ferramentas Jester, Jumble, MuJava, JavaMut, Response Injection (RI) e Judy descritas comparativamente em (Madeyski e Radyk, 2010).

No contexto de testes em projetos ágeis, o teste de mutação já foi aplicado para análise de códigos de teste utilizando a estratégia TDD (Madeyski, 2010).

3.2 Teste de Software em Métodos Ágeis

Diferentemente de métodos tradicionais na qual a atividade de teste ocorre mais tarde no processo de desenvolvimento, os testes ágeis devem ocorrer de forma frequente, procurando detectar defeitos o mais cedo possível em de ciclos de desenvolvimento iterativos e curtos, com um constante *feedback* do cliente. Equipes de desenvolvimento ágeis podem parar de desenvolver novas funcionalidades durante um período, mudando o foco para a estabilidade do código, corrigindo defeitos, refatorando código e executando casos de teste de regressão para assegurar-se de que não houve nenhuma alteração no código que afetou o funcionamento do sistema em desenvolvimento (Champion, 2008). O teste de regressão necessita essencialmente de ferramentas e dos casos de testes armazenados em um repositório.

Em projetos ágeis também há uma preocupação em testar sob o ponto de vista do cliente por meio de testes de aceitação. Para criação de testes de aceitação, há um esforço colaborativo entre um especialista do negócio ou domínio e o desenvolvedor, analista ou qualquer outro membro do time de desenvolvimento. Cada história do cliente deve ter um ou mais casos de teste de aceitação associados, que idealmente são baseados em exemplos fornecidos pelo cliente e além de ajudar na compreensão das histórias, validam o software desenvolvido. Um aspecto enfatizado em projetos ágeis é abordagem do time completo (prática introduzida pelo método XP), no qual toda a equipe é responsável pela qualidade, e os testadores devem ser integrados à equipe. A sinergia entre desenvolvedores e testadores pode ocorrer por meio do desenvolvimento de soluções em pares, melhorando assim a comunicação a respeito da qualidade do produto. Neste sentido, é desejável que todos os integrantes da equipe reconheçam a importância dos testes, desde o nível de unidade a níveis mais altos, direcionando o desenvolvimento do código, ajudando a equipe a entender como a aplicação deve funcionar, e servindo como *feedback* sobre que tarefas ou histórias concluídas (Crispin e Gregory, 2009).

A qualidade de software possui diversas dimensões, cada uma requer uma diferente abordagem de testes. Neste contexto, Brian Marick dividiu a atividade de teste em métodos ágeis em quatro categorias de testes: (1) TDD (unidade e integração), (2) testes de negócio (*business/customer-facing tests*), (3) testes de negócio em relação ao produto desenvolvido (testes exploratórios, usabilidade, cenários, entre outros) e (4) testes relacionados à performance, carga e segurança) (Crispin e Gregory, 2009).

A estratégia de teste de software em projetos ágeis é aplicada conforme o método ágil e as práticas de teste utilizadas. Uma das estratégias que pode ser aplicada é descrita por Crispin e Gregory (2009), que vem sendo aplicada com sucesso na indústria e consiste nas seguintes atividades (Figura 3.2): (1) Visão Geral do Produto e Planejamento: nessa fase a equipe de teste é responsável por colaborar no entendimento do projeto e na criação de histórias, além de se preocupar com a criação de um plano de teste que será aplicado durante o projeto, (2) Ciclo Iterativo: no ciclo iterativo as tarefas do projeto são estimadas e as funcionalidades são desenvolvidas utilizando TDD (testadores podem programar em pares

com desenvolvedores). Nessa fase também são escritos e executados testes em mais alto-nível para as histórias (também chamados de *story tests* ou *business tests*), são executados testes de regressão automatizados e testes de carga. Os testadores também podem participar de demonstrações de funcionalidades ao cliente durante o desenvolvimento. (3) Testes de Sistema: nessa fase podem ser utilizados testes exploratórios, testes de aceitação, testes ligados a requisitos não funcionais como segurança e performance. Também são utilizados testes de fumaça (*smoke testing*)[2] para funcionalidades novas ou que foram alteradas e todo o conjunto de casos de teste deve ser executado, (4) Nova versão do Produto / Suporte: nessa fase os testadores participam das reuniões de retrospectivas e da liberação do código para produção. Além disso, durante todas as fases devem ser definidas e coletadas métricas relacionadas a atividade de teste, para facilitar o acompanhamento dos testes automatizados e também para facilitar a melhoria do código de teste e do código do sistema. Além disso, essas métricas devem ficar visíveis para que possam ser utilizadas durante as iterações e também em reuniões durante o desenvolvimento e retrospectivas.

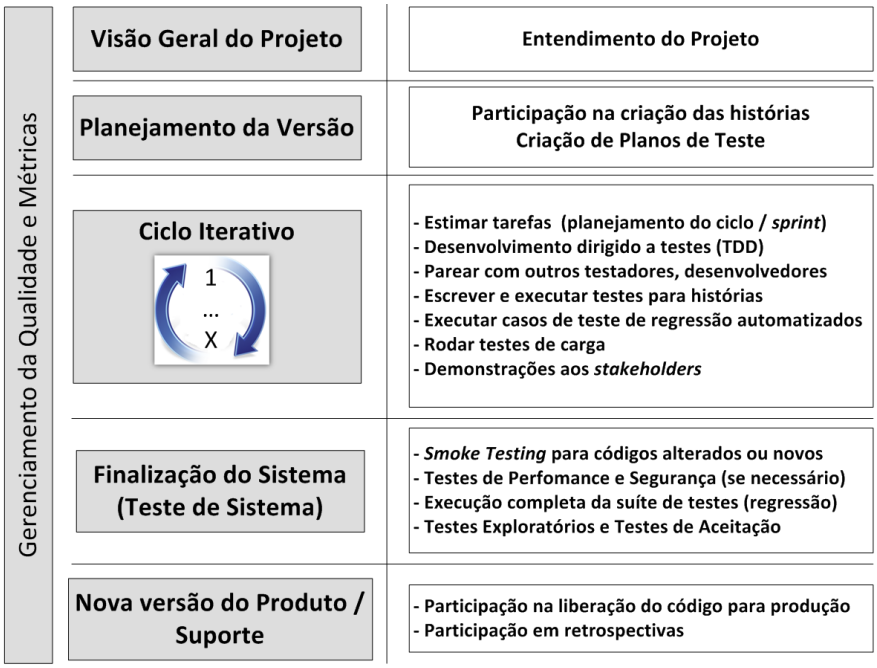


Figura 3.2: Estratégia de Teste em Projetos Ágeis - Adaptado de (Crispin e Gregory, 2009)

A utilização de ferramentas é bastante enfatizada em métodos ágeis, principalmente para facilitar no desenvolvimento e execução de um grande conjunto de testes de forma eficiente, que forneçam resultados rápidos a respeito do estado do sistema (Janzen e Saiedian, 2005).

Uma das grandes motivações para adoção de métodos ágeis é a qualidade em projetos de software. A qualidade vem sendo alcançada em projetos ágeis e é apontada como um dos grandes benefícios da adoção de métodos ágeis em empresas de software (Begel e Nagappan, 2007; Layman et al., 2004; Version One, 2010). A melhoria contínua em projetos ágeis é

[2] *Smoke testing* é uma abordagem na qual todas as áreas da aplicação são testadas sem profundidade.

alcançada por meio de diversos princípios e práticas, como por exemplo: ciclos de *feedback* curtos, integração e entrega contínua de funcionalidades que agregam valor ao cliente, além de reuniões entre a equipe de projeto e clientes e diversas práticas de desenvolvimento como a programação em pares e o TDD. Alguns métodos ágeis também sugerem a utilização de instrumentos informativos, compostos por exemplo de um quadro de histórias, gráficos como o *Sprint Burndown* (sugerido pelo Scrum) ou também por meio da visibilidade do progresso e resultados utilizando relatórios.

A atividade de teste assume papéis essenciais no processo de desenvolvimento de software utilizando métodos ágeis. O teste apoia a comunicação entre desenvolvedores e clientes, fornece *feedback* sobre quais as funcionalidades do sistema estão funcionando e apoia a manutenção do sistema, pois as mudanças tendem a ser mais seguras se a equipe tiver um bom conjunto de testes para o sistema. Pressman (2006) descreve duas estratégias utilizadas para o teste de software. Em um extremo, o time de desenvolvimento apenas testa o sistema quando ele esteja totalmente construído com o objetivo de encontrar defeitos. Entre os dois extremos está a estratégia de teste incremental (geralmente utilizada em abordagens de teste tradicionais). E por fim, no outro extremo, a equipe conduz testes diariamente, sempre que qualquer parte do sistema é construída. Essa prática vem sendo utilizada em métodos ágeis por meio da prática do Test Driven Development (TDD) e tem se mostrado bastante efetiva.

Os métodos ágeis tratam as fases de teste de maneiras diferente, sendo que as fases mais enfatizadas são as de teste de unidade e teste de aceitação. Os ciclos de desenvolvimento em projetos ágeis são iterativos e curtos e devido a isso, os testes são executados frequentemente, durante todo o ciclo de vida do software (conceito introduzido pelo método DSDM). Na Tabela 3.1 é apresentado um comparativo entre métodos ágeis relacionado as fases de teste, o suporte as atividades de gerenciamento dos testes e também à existência de um guia concreto que detalhe como os testes devem ser conduzidos. Esse comparativo foi feito com base em dois trabalhos (Abrahamsson et al., 2003) e (Strode, 2005).

Tabela 3.1: Comparativo sobre a atividade de Teste em Métodos Ágeis.

	DSDM	XP	Scrum	Crystal	ASD[2]	FDD[2]
Teste de Unidade [1]	• • ○	○ • •	• ○ ○	• • ○	• • ○	• • ○
Teste de Integração [1]	• • ○	○ • •	• • •	• • ○	• • ○	• • ○
Teste de Sistema [1]	• • ○	○ • •	• • •	• • ○	• • ○	• • ○
Teste de Aceitação [1]	• • ○	○ • •	• ○ ○	• • ○	○ ○ ○	○ ○ ○
Testes durante todo o ciclo de vida	√	√	√	√	-	√
Testes de Unidade Automatizados	-	√	√	√	-	√
Testes de Regressão Automatizados	-	√	√	√	-	-

[1] [• sim][○ não] (1°) Gerenciamento de Projeto (2°) Processo Descrito (3°) Guia Concreto (práticas/atividades/produtos)
[2] Utilizam Inspeção de Código para validar o produto

Diferentes práticas e estratégias são utilizadas pelos diversos métodos ágeis. No método XP, todo trecho de código tem um conjunto de casos de teste de unidade automatizados,

entregue dentro do repositório de código-fonte. Os resultados dos testes servem como uma forma de *feedback* instantâneo, no qual o desenvolvedor pode detectar rapidamente se o método desenvolvido ainda precisa ser modificado ou refatorado. O código é considerado completo apenas se passar por todos os testes de unidade. Além disso, no fim de cada iteração, todos os testes de aceitação que foram criados durante a fase de planejamento são executados por usuários e clientes. Esses testes incluem os testes de iterações prévias e da última iteração, para determinar se as novas funcionalidades são aceitáveis, prevenindo que novas mudanças não causem efeitos colaterais em funcionalidades prévias que estavam funcionando.

No método Scrum, os testes são executados como tarefas em *Sprints* de desenvolvimento (incrementos do software), sendo que um *Sprint* pode ter como objetivo exclusivo a execução de tarefas relacionadas a testes, melhoria do código e correção de defeitos. A funcionalidade esta pronta para ser entregue ao cliente somente após ter passado por testes de integração e de sistema. Nos métodos FDD e ASD, além da atividade de teste, são executados inspeções de projeto e de código. No método FDD podem ser executados testes baseado em modelos, em consequência da criação de modelos de análise e projeto orientados a objeto, como diagrama de classes e diagramas de atividades.

Além dos testes, outra prática para apoiar a melhoria contínua e as mudanças que podem ocorrer no decorrer do projeto são as reuniões. As reuniões ocorrem durante os ciclos de desenvolvimento e também a cada fim de iteração ou versão. Na Tabela 3.2 é apresentado um comparativo entre os métodos ágeis em termos de práticas para retrospectiva e melhoria durante o projeto, além das práticas de VV&T aplicadas em projetos ágeis.

Entre as práticas de teste existentes em métodos ágeis, pode-se citar testes utilizando Test Driven Development (TDD) (Beck, 2002), testes de integração contínuos, testes de aceitação com o cliente e testes de regressão associados à prática de refatoração (melhoria do código). Todas essas práticas de testes devem ser executadas preferencialmente de forma automatizada, buscando a agilidade no processo de testes. Essas estratégias e técnicas de teste utilizadas em projetos ágeis serão apresentadas nas seções seguintes.

3.2.1 Test Driven Development (TDD)

O Desenvolvimento Dirigido a Testes (*Test Driven Development* – TDD), introduzido por Beck (2002), de forma geral significa escrever testes de unidade antes de desenvolver o código-fonte em iterações pequenas e rápidas. Segundo Janzen e Saiedian (2005), essa unidade em programas orientados a objetos se refere a classe ou método. Com a utilização do TDD um conjunto de testes de unidade é mantido e irá guiar o desenvolvimento durante todo o processo de implementação.

A ideia pregada pelo TDD força os programadores a pensarem em diversos aspectos da funcionalidade e projeto antes da escrita do código-fonte. Além disso, a utilização do TDD

Tabela 3.2: Comparativo entre Métodos Ágeis (Reuniões, Retrospectiva, Melhoria e Técnicas e Práticas de VV&T) (Vicente et al., 2009)

	Retrospectiva e Melhoria	**Técnicas e Práticas de VV&T**
XP	- Ciclos semanais e trimestrais	- Integração contínua e testes - *Test-Driven Development* - Testes de aceitação associados a histórias do usuário
Scrum	- Reuniões diárias (*Daily Scrum*), - Reunião de revisão (*Sprint Review*) - Retrospectiva	- *Builds* diários e testes (todos os tipos) - Testes durante o *sprint* e Teste de sistema
FDD	- Relato e visibilidade dos resultados - Inspeções de projeto e código	- Inspeção do projeto e do código - Teste de unidade
Crystal	- Monitoramento do progresso - *Workshops* de reflexão	- Teste de regressão de funcionalidades automatizados
ASD	- Revisões de qualidade	- Teste faz parte da construção concorrente de componentes - Inspeções de código - Revisões para melhorar a qualidade do produto (Testes de Aceitação e Sistema)
DSDM	- *Workshops* com o cliente	- Testes integrados durante todo o ciclo de vida - Todo componente de software é testado pelos desenvolvedores e usuários assim que são desenvolvidos

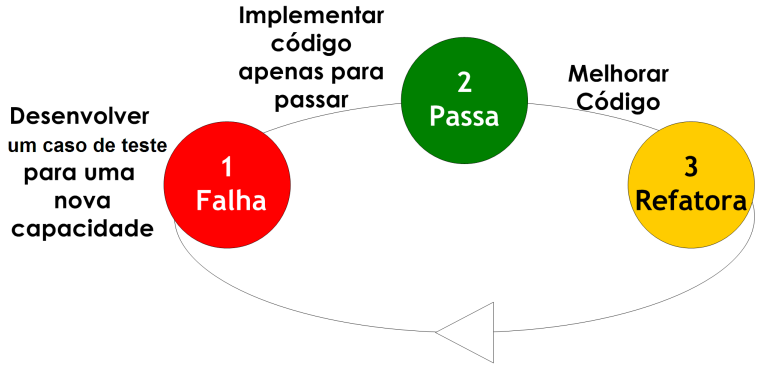
fornece um conjunto de casos de teste que pode ser executado a cada atualização do código, garantindo que a refatoração, atualização ou um novo código não faça com que funcionalidade já existente pare de funcionar.

As atividades do TDD, ilustradas na Figura 3.3 são descritas a seguir: (1) pensar sobre uma pequena funcionalidade que deve ser implementada; (2) desenvolver um caso de teste que falhe para esta nova funcionalidade, especificando como o programa deve invocar esta funcionalidade e qual o resultado esperado. O teste irá falhar, mostrando que a funcionalidade ainda não está presente; (3) implementar o código para que o teste seja satisfeito e além disso deve-se verificar que todos os testes anteriores continuam sendo satisfeitos e (4) Refatorar o código, o código é revisado e o seu *design* é melhorado. A refatoração simplifica e deixa mais claro o código-fonte e o código de testes.

O TDD pode ser utilizado para explorar, projetar, desenvolver e testar o software, não devendo ser tratado apenas como uma atividade de testes (Janzen, 2006; Jeffries e Melnik, 2007). Essa estratégia foi aplicada de diversas formas por várias décadas, no entanto ganhou força como sendo um das práticas mais importantes do XP (Jeffries e Melnik, 2007).

Os benefícios do TDD vão além da conveniência de se executar uma grande massa de testes em pouco tempo (Martin, 2007):

- **Confiança:** os casos de teste validam o comportamento do sistema que está sendo desenvolvido.



O ciclo do *test-driven-development*
(1) desenvolver um teste que falhe;
(2) implementar código para passar no teste e
(3) melhorar o projeto via *refactoring*.

Figura 3.3: Ciclo do Test Driven Development (TDD) (Jeffries e Melnik, 2007)

- **Flexibilidade:** o TDD permite que o desenvolvedor melhore seu código por meio da refatoração, sem se preocupar que estará provocando erros em alguma parte do software.

- **Documentação:** testes podem documentar o software de forma não ambígua, estando sempre atualizados de acordo com o desenvolvimento do software e podendo ser executados rapidamente a qualquer hora. No entanto, para que isso se torne realidade os testes devem ser fáceis de localizar e ler, deixando-os no mesmo nível de qualidade que o código fonte.

- **Minimização de Depuração:** se os desenvolvedores estiverem seguindo o ciclo do TDD de forma correta, nunca irá demorar muitos tempo para rodar todos os seus casos de teste. Isso significa que a detecção de um defeito que foi inserido no sistema pode ser detectado em pouco tempo. Não será necessário executar durante muito tempo a atividade de depuração, pois o defeito detectado geralmente foi inserido recentemente.

- **Melhor projeto (acoplamento e coesão):** um dos indícios de problemas com o *desing* do software é a dificuldade para escrever o teste. Códigos considerados testáveis devem ser desacoplados e independentes de chamadas a outros métodos.

Esses benefícios podem ser utilizados para se desenvolver código limpo e flexível, que funcione e seja entregue a tempo (Martin, 2007):

- **Código limpo e flexível:** o TDD sugere a refatoração para produção de um código de fácil entendimento (limpo). Além disso, o código deverá fácil de ser mantido (flexível), ajudando a eliminar o medo de melhorar um código que poderia gerar diversos defeitos no software. Um teste rápido poderá dar o *feedback* ao desenvolvedor se o novo código é mesmo eficiente.

- **Código que funciona:** sempre a equipe de desenvolvimento saberá as funcionalidades do sistema que estão passando ou não pelos testes, sejam eles testes automatizados utilizando TDD, teste manual, exploratório ou teste de sistema. Por fim, nenhum programador profissional irá entregar um código à equipe de qualidade, esperando uma lista de erros.

- **Código entregue em tempo:** o TDD não garante que tudo será entregue conforme o tempo previsto, no entanto com ele é possível eliminar variáveis. É mais fácil estimar o tempo de uma tarefa quando não for perdido muito tempo com depuração. A equipe terá o *status* de uma tarefa conforme a quantidade de testes que já passaram.

Apesar do TDD poder melhorar a qualidade do software de forma significativa, a produtividade e o impacto do TDD em termos de projeto do código ainda são pouco evidentes, demonstrando a necessidade de mais estudos sobre o assunto (Siniaalto e Abrahamsson, 2007). Conforme apontado por uma pesquisa divulgada recentemente, os testes de unidade vêm sendo aplicados na grande maioria dos projetos ágeis, enquanto a estratégia TDD e testes de aceitação automatizados ainda não vêm sendo utilizados na mesma intensidade (Version One, 2010). Uma das explicações para esse cenário é a falta de experiência dos desenvolvedores com a atividade de testes e a mudança do paradigma de testes tradicionais para o TDD (Puleio, 2006). Nesse contexto, o treinamento de programadores e estudantes para utilização do TDD é essencial para a aplicação efetiva dessa estratégia de testes.

A estratégia TDD para o teste de unidade e integração (voltado ao desenvolvedor) também pode ser aplicada em testes voltados aos requisitos do cliente, que são chamados de testes de aceitação, testes de negócio ou "*story tests*". A prática de testes de aceitação será discutida na seção a seguir.

3.2.2 Testes de Aceitação (*Business Testing*)

Outra prática de teste utilizada em métodos ágeis é o Desenvolvimento Dirigido a Testes de Aceitação (*Acceptance Test Driven Development*- ATDD) (Melnik, 2007; Mugridge, 2008). Essa estratégia também é chamada de testes de aceitação, teste de negócios (*business testing*), teste de histórias (*story testing*) ou teste para os clientes (*customer testing*) (Deng, 2007). O ATDD possui o mesmo processo do TDD, porém, no ATDD são utilizados testes ligados a requisitos de negócio ao invés de testes de unidade e integração como no TDD tradicional. Os testes de aceitação são criados pelo cliente e são associados a histórias do usuário. O objetivo desses testes é encorajar uma comunicação clara das necessidades de negócio essenciais (e formas de atender a essas necessidades) utilizando exemplos concretos de funcionalidades do software.

Clientes podem descrever histórias como esta: *"Como comprador, quero adicionar itens ao meu carrinho de compras, então pagarei por todos eles no final"*. Essa história pode ser

complementada por um exemplo: *"Haverá diversos itens na página. Comprarei um item 1 por R$ 20,25 e vou colocá-lo no carrinho de compras. Clicarei para continuar comprando e selecionarei mais um segundo item de R$ 5,38 e colocarei no carrinho de compras. Por fim eu clicarei na opção finalizar compra, e será mostrado os dois itens no carrinho de compras com o valor de cada um e o custo total da compra".* O teste para esta história poderá ser expresso no formato apresentado na Tabela 3.3 e tem como função capturar o exemplo em um formato executável. Mais testes podem ser escritos para classes de equivalência, limites, e outros cenários.

Tabela 3.3: Exemplo - Teste de Aceitação para História (Crispin e Gregory, 2009)

	Entradas		**Resultados Esperados**	
ID	**Item**	**Preço**	**Custo Total**	**# de Itens**
001	Item A	20.25	20.25	1
002	Item D	0.01	20.26	2
003	Item F	100.99	121.25	3

A execução dos testes de forma automatizada pode ajudar os desenvolvedores a determinar quando novas funcionalidades estão completas ou se alguma funcionalidade existente está com algum defeito. Além disso, os clientes podem verificar se as funcionalidades do software estão produzindo as respostas que eles esperavam. Uma ferramenta bastante utilizada para testes de aceitação, são as ferramentas derivadas do *framework* FIT (*Framework for Integrated Test*) (Deng, 2007). Ela se adapta bem a atividade de testes sob uma perspectiva de negócios, utilizando tabelas para representar os testes e automaticamente documentar os seus resultados.

Uma outra estratégia que pode ser aplicada no contexto de testes de aceitação é a estratégia *Behavior-driven Development* (BDD) (North, 2006). O BDD também apoia a criação testes de aceitação no formato de cenários que especificam os testes em linguagem natural. Os cenários são descritos no seguinte formato: *"Dado (Given) um contexto inicial, quando (When) um evento ocorre, então (Then) assegure alguns resultados".* Um cenário para um cliente que deseja sacar dinheiro de um ATM pode ser o seguinte:

Cenário 1: *Conta tem crédito*
Dado que a conta possui crédito
E o cartão é válido
E a máquina contém dinheiro
Quando o cliente requisitar dinheiro
Então assegure que o valor sacado seja debitado na conta
E assegure que o dinheiro é entregue
E assegure que o cartão é retornado.

Mais detalhes sobre ferramentas de apoio a testes de aceitação utilizando tabelas Fit e cenários BDD serão descritos na Seção 3.3.

3.2.3 Outras Práticas de Teste Ágil

Nessa seção são apresentadas outras práticas de teste utilizadas em projetos ágeis: O teste exploratório que complementa os testes utilizando TDD e os testes de aceitação, além dos testes de usabilidade e testes de sistema.

Teste Exploratório

O Teste Exploratório (*Exploratory Testing* - ET) é uma prática importante para a atividade de testes em métodos ágeis, complementando o TDD e o teste de aceitação. Segundo o guia SWEBOK (*Software Engineering Body of Knowledge*) (Abran et al., 2004), o ET é uma abordagem que combina aprendizagem, projeto e execução, no qual os testes não são definidos anteriormente em um plano de testes, mas são dinamicamente projetados, executados e modificados.

Itkonen e Rautiainen (2005) baseados na definição do SWEBOK, apresentam algumas propriedades que descrevem o teste exploratório: (1) os testes não são definidos a priori como um plano detalhado ou casos de teste, mas sim por objetivos gerais sem instruções específicas de como atingi-los; (2) ET é guiado por resultados e conhecimento gerado por testes executados anteriormente, além de informações de outras fontes como documentos de requisitos ou manuais do usuário; (3) o foco do ET é encontrar defeitos por meio da exploração, ao invés de produzir sistematicamente um conjunto de casos de teste; e (4) a efetividade dos testes dependem do conhecimento, habilidades e experiências do testador.

Testes Exploratórios não devem ser exaustivos, mas devem ser vistos como uma abordagem que verifica se as funcionalidades já implementadas realmente satisfazem o que foi requisitado pelo cliente (Crispin e Gregory, 2009). Diversas formas de utilização dos ETs foram identificadas por Itkonen e Rautiainen (2005):

- **Seções de teste exploratórios:** são planejadas utilizando pequenas descrições, que representam brevemente a tarefa de teste, os objetivos da seção, e o artefato a ser testado. Os resultados principais dessas seções são os defeitos, que são relatados no sistema de rastreamento de defeitos.

- **Testes funcionais exploratórios:** são executados imediatamente após uma funcionalidade individual ter sido implementada. Esses testes verificam se a implementação corresponde ou não as especificações da funcionalidade.

- **Testes exploratórios de fumaça (*smoke testing*):** tentam identificar defeitos em funcionalidades que foram consertadas e melhorias implementadas na última versão.

- **Testes de regressão exploratórios:** também verificam consertos e mudanças após a implementação de um único conserto. Esses testes de regressão não testam o sistema

inteiro exaustivamente, mas se baseiam na experiência do testador, explorando novos defeitos ou defeitos relacionados causados pelas mudanças.

- **Testes exploratórios executados por usuários:** verificam diversos cenários de funcionalidades do sistema.

A utilização de ETs trazem diversos benefícios, como revelar áreas do produto que podem utilizar mais casos de teste automatizados que podem não ter sido incluídos no plano de teste (Crispin e Gregory, 2009). Exemplos de tais testes podem incluir o teste de dependências de novas funcionalidades e funcionalidades existentes. Testar novamente defeitos que foram corrigidos também traz benefícios, possibilitando a exploração de novos defeitos e defeitos relacionados. Itkonen e Rautiainen (2005) apresentam resultados sobre a efetividade e eficiência de ETs detectarem defeitos em um pequeno espaço de tempo. O artigo relata que uma maior quantidade de defeitos foi encontrada utilizando o teste exploratório de que o teste baseado em casos de teste. Apesar dos benefícios, é difícil estimar a cobertura dos testes exploratórios. Apesar dos ETs terem seus objetivos gerais é dificil estabelecer quando a aplicação foi suficientemente testada segundo tais objetivos. Além disso, testar funcionalidades de sistemas complexos pode demandar bastante tempo.

Teste de Usabilidade e Teste de Sistema

O teste de usabilidade deve ser focado nos grupos de usuário que utilizam o sistema. Devem ser feitas entrevistas para descobrir como eles utilizarão o sistema e quais as suas reações. A navegação é outro aspecto importante do teste de usabilidade. Outras aplicações similares podem ser comparadas para verificar como elas realizam certas tarefas.

Requisitos de performance, segurança, interação com outros sistemas, e outros atributos não funcionais devem ser pensados na fase de projeto e desenvolvimento do sistema. Alguns desses atributos podem ser mais importantes que uma funcionalidade, como por exemplo o tempo de espera em um sistema de vendas pela Internet. A escalabilidade do sistema como um todo também deve verificar se a aplicação continua confiável quando mais usuários utilizarem o sistema. Teste de performance geralmente são utilizados para identificar pontos de estrangulamento no sistema. Teste de carga e *stress* devem verificar o comportamento do sistema conforme cresce o número de usuários, além de verificar a robustez da aplicação em condições extremas. Diversas ferramentas automatizadas podem apoiar os testes de performance e carga, facilitando o processo de avaliação desses aspectos.

3.2.4 Revisão Sistemática sobre Testes Ágeis

Essa seção apresenta uma revisão sistemática, que teve como objetivo verificar como a atividade de teste vem sendo tratada em métodos ágeis (Vicente et al., 2009)[3]. Para isso, como questão primária procurou-se identificar e analisar quais estratégias, técnicas e critérios de teste são utilizados no contexto de tais métodos. Adicionalmente, foram protocoladas três questões secundárias: (1) qual a importância da atividade de teste em métodos ágeis, (2) quais ferramentas de apoio são utilizadas para testes automatizados em métodos ágeis e (3) quais os resultados de estudos experimentais para a atividade de teste em métodos ágeis que validam a abordagens ágeis de teste nestes métodos.

As fontes selecionadas foram bases de dados eletrônicas indexadas (*Springer*, *ACM Digital Library*, *IEEEXplore* e *Science Direct*). Foram recuperados um total de 473 referências, das quais 268 artigos (56,6%) foram da IEEE Xplore, 72 da ACM Digital Library (15,22%), 119 trabalhos da SpringerLink (25,16%), 14 da ScienceDirect (2,96%). Na etapa de seleção final foram identificados e analisados 29 trabalhos relevantes para os objetivos da revisão.

A importância da atividade de testes em métodos ágeis pode ser constatada pela maioria dos métodos ágeis que consideram a atividade de testes durante todo o seu ciclo de vida. A utilização de testes durante o desenvolvimento segundo os artigos selecionados facilita a compreensão do código-fonte, apoia a documentação do software e a manutenção do código-fonte (acaba com o medo de mudanças), além de resultar em um código-fonte de melhor qualidade.

Dos 29 estudos selecionados, 20 trabalhos (69%) discutem a respeito da estratégia TDD, 3 trabalhos (10%) discutem o ensino de TDD e os 6 trabalhos restantes (21%) falam a respeito de testes de aceitação ágeis. Dos 23 estudos sobre TDD (incluindo estudos na área de ensino), a sua grande maioria (21 trabalhos) trazem algum tipo de estudo experimental conduzido na indústria ou academia. Esses estudos foram categorizados em experimentos controlados, experimentos quasi-controlados (um experimento na qual as condições das unidades não são determinadas randomicamente) e estudos de caso, conforme a Tabela 3.4 a seguir.

Tabela 3.4: Tipos de Estudos Experimentais - TDD(Vicente et al., 2009)

	Academia	**Industrial**
Exp. Controlado	3	1
Exp. Quasi-Controlado	1	3
Estudo de Caso	4	6
Diversos Estudos Exp.	3	3

Os atributos avaliados nestes estudos são descritos na Tabela 3.5. Esses estudos mostram diversas melhorias como consequência da utilização do TDD, como código de melhor qualidade devido à menor quantidade de defeitos e à maior cobertura de código. Além

[3]Mais detalhes sobre os estudos selecionados e outras estatísticas desta revisão sistemática estão disponíveis em: `http://www.labes.icmc.usp.br/avicente/revisao-testesageis`.

disso apontaram maior reúso, melhoria do código, menos tempo de depuração e usuários mais satisfeitos. A maioria dos estudos que tratavam sobre a produtividade utilizando TDD apontaram uma melhoria neste aspecto, contrariando os estudos descritos no trabalho de Jeffries e Melnik (2007) que apresentam uma queda na produtividade da equipe. Foram realizados experimentos em relação a métricas de código orientados a objeto, demonstrando em alguns casos nenhuma diferença na qualidade e pouca diferença em termos de acoplamento e coesão. Um dos estudos também apontou uma complexidade ciclomática constante durante as iterações utilizando-se TDD. Por fim, alguns estudos detectaram uma melhoria na qualidade, resultando em módulos de software menores, menos complexos e mais testados que módulos produzidos com testes tradicionais. Um dos estudos fez uma análise de diversos experimentos da academia e indústria com TDD. Os resultados da academia são bastante controversos, o que se deve principalmente à inexperiência dos estudantes na utilização do TDD.

Tabela 3.5: Resultados dos Estudos Experimentais - TDD (Vicente et al., 2009)

	Melhoria	**Neutro**	**Negativo**
Acoplamento e Coesão		1	
Custos Financeiros			1
Cobertura	4		
Complexidade Ciclomática	1		
Defeitos	7		
Entendimento do Programa	2		
Maior Reúso	2		
Melhoria de Código	2		
Menos tempo de Depuração	1		
Métricas OO		2	
Performance melhor dos alunos	1		
Produtividade	6	1	3
Qualidade em Geral	4	3	
Usuários mais satisfeitos	1		

A revisão sistemática também selecionou 6 trabalhos (21% dos trabalhos selecionados) que tratam de testes de aceitação com o cliente. A partir desses trabalhos chegou-se a conclusão de que essa técnica de teste, apesar de demandar um grande esforço por parte dos clientes e desenvolvedores, proporciona uma melhor qualidade e entendimento dos requisitos. Além disso, melhora a comunicação entre clientes e equipe de desenvolvimento, serve como indicador do progresso do desenvolvimento e melhora a identificação e correção de defeitos.

O custo para escrita e manutenção de códigos de teste é desafiador e pode ser otimizado pelo uso de ferramentas. Os testes devem ser fáceis de codificar, de ler e manter, além disso, devem ser acompanhados utilizando-se métricas. Nos estudos selecionados foram citadas principalmente ferramentas de apoio à criação e execução de testes de unidade (como a ferramenta JUnit) e ferramentas de teste de aceitação (como o *framework* FIT e a ferramenta

Selenium). Também foram detectadas algumas iniciativas de novas ferramentas de apoio a testes ágeis como a FitNesse (teste de aceitação) e o Zorro (gerenciamento do processo TDD).

3.3 Ferramentas de Teste utilizadas em Projetos Ágeis

Ferramentas de teste vêm sendo desenvolvidas em diversos contextos, desde ferramentas que implementam os critérios de teste descritos anteriormente até ferramentas de teste de interface-gráfica (GUI), performance e segurança. Essas ferramentas são de fundamental importância para a qualidade e produtividade da atividade de teste, uma vez que a atividade de teste é propensa a erros, além de improdutiva, se aplicada manualmente (Delamaro et al., 2007b). Nessa seção serão apresentadas algumas ferramentas utilizadas em projetos ágeis.

Testes de Unidade e Gerência de Casos de Teste

Em projetos ágeis, ferramentas de teste de unidade executam frequentemente o conjunto de casos de teste para provar a estabilidade do sistema. Essas ferramentas fornecem um constante *feedback* nos ciclos iterativos do TDD, para novas funcionalidades implementadas e quando parte do código for modificado ou refatorado.

Atualmente uma gama de ferramentas para teste de unidade conhecidas como família XUnit (Meszaros, 2007) estão sendo utilizadas em projetos ágeis. Essas ferramentas utilizam classes de teste com diversos casos de teste criados na forma de assertivas. As classes de teste são escritas utilizando-se a linguagem do código que está sendo testado, contendo um ou mais métodos que especificam os casos de teste, podendo ser organizados de forma hierárquica. Essas ferramentas podem testar o sistema em partes separadas, integradas ou até mesmo todas as unidades. As principais ferramentas da família XUnit são as ferramentas JUnit e TestNG para Java, NUnit para .Net, PHPUnit para PHP e PYUnit para Pyton.

O teste de unidade deve-se focar em uma unidade, no entanto essas unidades possuem dependências entre si. Ferramentas como EasyMock e JMock podem ser utilizadas para implementar objetos *mock* que simulam o comportamento de objetos reais (isolando essas dependências) e *stubs* de teste. Além de isolar os testes de unidade, os objetos *mock* aceleram a preparação ou execução dos testes, permitindo testar uma funcionalidade mesmo que algum componente não esteja pronto ou disponível.

Para ajudar na gerência do processo de teste de software podem ser utilizadas ferramentas para o planejamento, gerenciamento e execução de todo o conjunto de casos de teste. Alguns exemplos de ferramentas para o gerenciamento dos testes são as ferramentas TestLink e a ferramenta Testopia que é integrada a ferramenta Bugzilla.

Ferramentas de Teste de Cobertura (Critérios Estruturais)

Ferramentas de cobertura de código utilizam critérios de teste estrutural, sejam eles fluxo de controle ou fluxo de dados. Por meio desses critérios, essas ferramentas avaliam quanto do código foi exercitado pelos testes. Diversas ferramentas de cobertura são utilizadas em projetos ágeis, sendo que a cobertura pode ser utilizada como uma métrica, indicando pontos do código que não foram testados pelo conjunto de casos de teste. Grande parte das ferramentas são integradas a ambientes de desenvolvimento como o Eclipse, e tem integração com ferramentas de apoio a implantação do sistema, como as ferramentas Ant e Maven. Como pode ser constatado na Tabela 3.6, que faz um comparativo entre diversas ferramentas de teste, a maioria das ferramentas utiliza critérios de fluxo de controle, enquanto somente a ferramenta *JaBUTi*, Coverlipse e a *Data Flow Testing Tool* (Bluemke e Rembiszewski, 2009) implementam critérios baseado em fluxo de dados. A ferramenta *JaBUTi* será utilizada para a coleta da métrica de cobertura dos casos de teste no estudo de caso deste trabalho.

Tabela 3.6: Ferramentas de Teste de Cobertura para Linguagem Java

Ferramenta	Fluxo de Controle	Fluxo de Dados	Sobre a Ferramenta
Cobertura	√		**Execução:** Modo texto, suporte a Ant e Maven **Licença:** GPL **Site:** `http://cobertura.sourceforge.net`
Code Cover	√		**Execução:** Modo texto, *Plug-in* Eclipse, suporte a Ant **Licença:** GPL **Site:** `http://cobertura.sourceforge.net`
Coverlipse	√	√	**Execução:** *Plug-in* Eclipse **Licença:** CPL **Site:** `http://coverlipse.sourceforge.net`
Coverlipse	-	√	**Execução:** *Plug-in* Eclipse **Licença:** CPL **Site:** `http://coverlipse.sourceforge.net`
Data Flow Testing Tool (DFC)	-	√	**Execução:** *Plug-in* Eclipse **Licença:** - **Referência:** (Bluemke e Rembiszewski, 2009)
Eclma	√[1]		**Execução:** *Plug-in* Eclipse **Licença:** EPL **Site:** `http://www.eclemma.org`
JaBUTi	√	√	**Execução:** Modo texto, *Desktop* e Serviço Web **Licença:** GPL **Site:** `http://incubadora.fapesp.br/projects/jabuti/`

[1] Somente suporte a cobertura de linhas de código

A ferramenta *JaBUTi* (Vincenzi, 2004) foi desenvolvida no ICMC/USP. A ferramenta tem como intuito, ser um ambiente completo para o entendimento e teste de programas e componentes Java. A idéia da ferramenta é viabilizar o teste de programas Java em nível

de *bytecode*, possibilitando, com isso, não somente o teste de programas Java para os quais o código-fonte esteja disponível, mas também o teste de componentes Java. A *JaBUTi* fornece diferentes critérios estruturais para análise de cobertura, um conjunto de métricas estáticas para avaliar a complexidade das classes que compõem o programa/componente, e implementa ainda algumas heurísticas de *slicing* de programas que visam a auxiliar a localização de defeitos. A cobertura de casos de teste pode ser visualizada de três maneiras pelo testador: cobertura por critério, por cada método e também por cada caso de teste. A Figura 3.4 ilustra a cobertura dos casos de teste por critério.

Overall Coverage Summary by Criterion

Testing Criterion	Coverage	Percentage
All-Nodes-ei	19 of 30	63%
All-Nodes-ed	0 of 1	0%
All-Edges-ei	20 of 32	62%
All-Edges-ed	0 of 6	0%
All-Uses-ei	37 of 63	58%
All-Uses-ed	0 of 14	0%

Figura 3.4: *JaBUTi*- Cobertura de casos de teste por critério (Extraído de (Vincenzi, 2004))

A ferramenta *JaBUTi* faz a representação da estrutura do programa por meio do grafo de fluxo de controle (GFC) é usado para representar o fluxo de controle do programa, no qual cada nó representa uma sentença ou um bloco de sentenças executado sequencialmente e cada aresta representa o fluxo de controle de uma sentença ou bloco para outro bloco. Além disso, esse grafo pode ser estendido com informações a respeito das definições e usos das variáveis dos programas em cada nó e aresta do GFC para permitir a definição de critérios de teste baseados em fluxo de dados.

A partir da determinação do conjunto de requisitos de teste de cada critério, tais requisitos podem ser utilizados para avaliar a qualidade de um conjunto de teste existente e para desenvolver novos casos de teste visando a melhorar a cobertura dos requisitos pelo conjunto de teste. Por meio da Figura 3.5 visualiza-se um exemplo de um grafo *Def-Uso*, a partir do critério *Todos-Nós-Primários*.

A ferramenta também foi estendida para o teste de programas orientado a aspectos (*JaBUTi/AJ*) (Lemos, 2005; Masiero et al., 2006), testes de aplicações com conexão a banco de dados, teste de programas implementados utilizando agentes móveis e teste de aplicações J2ME. Além das versões desktop (*stand-alone*), recentemente foi implementada uma versão da ferramenta que permite que diversas funcionalidades da ferramenta sejam utilizadas por meio de serviços Web (Eler et al., 2009).

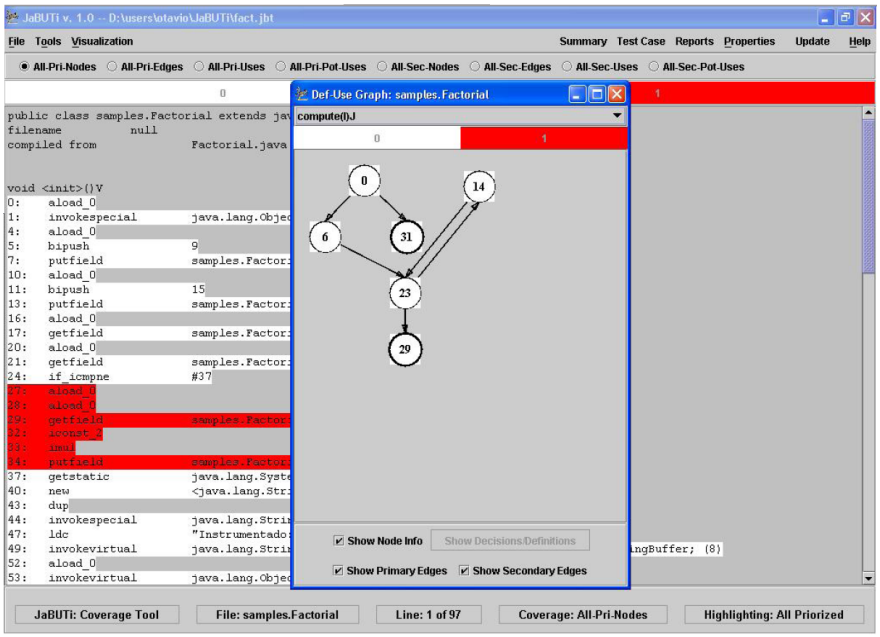


Figura 3.5: *JaBUTi*- Grafo de *Def-Uso*, a partir do critério *Todos-Nós-Primários* (Extraído de (Lemos, 2005))

Ferramentas de Teste de Aceitação (*Business Testing*)

O *framework* Fit (*Framework for Integrated Testing*) é um dos *frameworks* mais populares para teste de aceitação e foi desenvolvido para que clientes estivessem aptos a entender e até mesmo escrever testes (Deng, 2007).Os testes Fit são escritos em forma de tabelas com assertivas dentro das células da tabela. Código de suporte (*fixtures*) são desenvolvidos pelos programadores para executar tabelas de lógica de negócio, mapeando o conteúdo da tabela para serem chamadas pelo sistema. Os resultados do teste são mostrados em três cores diferentes: verde para testes que passaram; amarelo para testes que não puderam ser executados e vermelho para testes que falharam. Tabelas de teste Fit são criadas com ferramentas de negócio comuns como planilhas eletrônicas ou processadores de texto. Várias linguagens podem ser utilizadas para escrever as *fixtures*, incluindo Java, C#, Ruby, C++, Python, Objective C, Perl e Small talk (Mugridge e Cunningham, 2005). Um exemplo de tabela Fit e o seu respectivo código *fixture* é ilustrado na Figura 3.6.

Para apoiar o BDD (*Behavior-Driven Development*), ferramentas como JBehave para Java, Cucumber e RSpec para linguagem Ruby podem ser criados e executados diversos cenários de teste de forma automatizada. Os passos de cada cenário são especificados na mesma linguagem de programação e podem ser executados por essas ferramentas que fazem a ligação desses cenários com as funcionalidades já implementadas.

Outras Ferramentas de Teste

Os testes em projetos ágeis também podem ser executados por meio de entradas na interface gráfica do sistema (GUI). A criação dos casos de teste é feita por meio de recursos de gravação e execução de navegações pelo sistema, que devem gerar *scripts* que podem ser

ca.easytix.fixtures.CalculateDiscount				
tickets	senior	student	employee	discount?
9	no	no	no	0.00
10	no	no	no	0.05
20	no	no	no	0.10
1	no	no	yes	0.10
10	no	no	yes	0.10
20	no	no	yes	0.10
1	yes	no	yes	0.15
1	no	yes	yes	0.15
1	yes	yes	yes	error
1	yes	no	no	0.15
1	no	yes	no	0.15
1000	no	no	no	0.40 *expected* / 0.1 *actual*

```
public class   CalculateDiscount
           extends ColumnFixture {

  public int tickets;

  public boolean senior;
  public boolean student;
  public boolean employee;


  public float discount ()
     throws DiscountException {
     return
     ca.easytix.core.DiscountRule.
     getDiscount(tickets,
        senior, student, employee);
  }

}
```

Figura 3.6: Exemplo de Tabela Fit da Ferramenta FitNesse

editados e executados a qualquer momento pelo desenvolvedor. Exemplos de ferramentas para teste de interface desktop são as ferramentas Fest e Jemmy e para teste de interfaces Web podem ser utilizadas a ferramenta Watir (Ruby), Selenium e Canoo Web Test.

Para testes de desempenho e carga podem ser utilizadas ferramentas que verificam gargalos do sistema, como as ferramentas de código aberto: Apache JMeter, JProfiler, OpenWebLoad e JUnitPerf.

3.4 Considerações Finais

Neste capítulo primeiramente foi apresentada uma visão geral da área de teste de software tradicional e a sua importância no desenvolvimento de software, assim como estratégias, práticas e ferramentas utilizadas em projetos ágeis. Também foram apresentados estudos experimentais que evidenciam os benefícios e dificuldades em termos da atividade de teste em métodos ágeis.

As práticas de teste apoiam diversos valores e outras práticas ágeis, como por exemplo a integração contínua, o projeto de soluções simples e de qualidade, a refatoração e principalmente o apoio para efetuar mudanças durante o desenvolvimento. Todos incrementos de software devem ser validados por conjuntos de casos de teste de unidade, integração e o teste de aceitação para validar as funcionalidades conforme as necessidades do cliente. Para complementar esses testes, também são utilizados testes exploratórios e testes ligados ao funcionamento do sistema como um todo. O apoio automatizado da atividade de teste em métodos ágeis se torna ainda mais fundamental devido às constantes mudanças no soft-

ware que demandam consequentemente a execução rápida de testes que verifiquem que estas mudanças não afetaram o que estava funcionando corretamente.

No próximo capítulo serão apresentados conceitos gerais sobre métricas de software e métricas de teste, além da importância da utilização e acompanhamento de métricas em projetos ágeis durante todo o ciclo de vida. As métricas de teste de software para acompanhamento de projetos ágeis propostos nesse trabalho serão descritas e também será apresentado o protótipo da ferramenta ATMM (*Agile Testing Metrics Management Tool*) que implementa algumas das métricas propostas neste trabalho.

Capítulo

4

Métricas de Software

Métricas de software são padrões quantitativos de medidas de vários aspectos do projeto de software. A medição dessas métricas em um projeto de software pode apoiar estimativas, o controle de qualidade, a produtividade da equipe e o controle do projeto (Pressman, 2006). Além disso, um conjunto de métricas bem projetado pode ser utilizado para medir a qualidade de produtos de trabalho, dar suporte à tomada de decisão dos gerentes de projeto e aumentar o retorno de investimento do produto (Kulik, 2000). Em métodos ágeis de desenvolvimento, a utilização de métricas apoia a medição contínua do produto e do processo, permitindo que os ciclos de desenvolvimento sejam constantemente inspecionados, adaptados e melhorados (Hartmann e Dymond, 2006).

Equipes de desenvolvimento ágil são auto-organizadas e podem ser dirigidas por meio de entrevistas com o cliente, reuniões diária, *workshops* após incrementos ou revisões de projeto. Tais retrospectivas se concentram, tipicamente, no estabelecimento de mecanismos de rastreamento do estado atual do projeto, métricas e experiências da equipe de desenvolvimento (Wege, 2004). Além de mecanismos de retrospectiva, o método XP possui um papel específico chamado *tracker*. O *tracker* é responsável por prover informações do projeto para todos os seus integrantes do projeto, utilizando as métricas apropriadas para destacar a melhoria ou aspectos que merecem atenção da equipe (Jeffries et al., 2000). Além disso, ele deve atualizar regularmente essas informações em gráficos, pôsteres e tabelas na área de trabalho informativa, que Cockburn (2006) chama de irradiadores de informação.

Neste capítulo são apresentados conceitos gerais ligados a métricas de software e como essas métricas apoiam o acompanhamento de métodos ágeis, principalmente em relação a atividade de teste de software. Por fim, são apresentadas as métricas de acompanhamento

de teste e são descritos aspectos gerais da ferramenta ATMM (*Agile Testing Metrics Management Tool*) que implementou algumas dessas métricas e foi utilizada para os estudos de casos descritos no próximo capítulo.

4.1 Conceitos Gerais

Diversas características do software e de projetos de software podem ser medidas, como por exemplo, o seu tamanho, complexidade, qualidade, confiabilidade e aderência ao processo. Medidas, métricas e indicadores são conceitos importantes na área de métricas de software e muitas vezes são confundidos. Ragland (1995) descreve esses conceitos a partir de definições da IEEE (*Institute of Electrical and Electronics Engineers, Inc.*) e SEI (*Software Engineering Institute*):

- **Medida:** uma avaliação comparada a um padrão, ela fornece uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo do processo ou do produto. Exemplo - linhas de código, que podem variar conforme a linguagem ou regras para o cálculo de linhas de código.

- **Métrica:** um método quantitativo para determinar um certo atributo em relação ao sistema, componente ou processo. Ela geralmente é calculada ou composta por duas ou mais medidas. Exemplo - quantidade de casos de teste produzidos em uma iteração. As medidas que compõem essa métrica são o número de casos de teste produzidos e a iteração em que esses casos de teste foram produzidos.

- **Indicador:** um aparelho ou variável que pode ser configurado para um determinado estado conforme a ocorrência de um processo ou a ocorrência de uma situação especificada. Um indicador é algo que chama a atenção de uma pessoa para uma situação particular e pode considerar um comparativo entre métricas. Exemplo - aumento substancial de defeitos encontrados na última versão, que pode ser um indicador de que a qualidade do software piorou. Indicadores permitem avaliar o *status* de um projeto em andamento, acompanhar riscos em potencial, descobrir áreas-problema, ajustar o fluxo de trabalho ou tarefas e avaliar a capacidade da equipe de controlar a qualidade dos produtos produzidos (Kan, 2002).

Métricas de software podem ser classificadas em três categorias: métricas de processo, métricas de projeto e métricas de produto (Kan, 2002). Métricas do processo são utilizadas para a melhoria no desenvolvimento do software e manutenção. Exemplos de métricas de processo incluem a taxa de descoberta de defeitos, ou o tempo gasto para consertar defeitos encontrados. Métricas de projeto descrevem características do projeto e sua execução, e são utilizadas para minimizar atrasos, riscos e para avaliar a qualidade durante o

desenvolvimento. Exemplos de métricas de projeto incluem o número de desenvolvedores, custos, cronograma e produtividade. Métricas de produto descrevem características do produto como tamanho, complexidade, características ligado a projeto do código, performance e nível de qualidade.

As métricas de qualidade de software são um subconjunto de métricas que focam em aspectos de qualidade do produto, processo e projeto. Essas métricas podem ser divididas em métricas de qualidade do produto final ou métricas de qualidade calculadas durante todo o ciclo de desenvolvimento (chamadas de *in-process metrics*). Segundo Kan (2002), a essência da engenharia de qualidade de software é investigar as relações entre as métricas *in-process*, características do projeto, e a qualidade do produto final, e baseado nessas descobertas, melhorar a qualidade do processo e do produto.

Sato (2007) apresenta algumas das possíveis classificações para métricas que um *tracker* de um projeto deve considerar quando utilizar uma métrica:

- **Objetiva/Subjetiva:** Uma métrica objetiva depende somente do objeto que está sendo avaliado e não do ponto de vista de quem está interpretando. Um exemplo é o número de casos de testes de unidade ou o número de *commits* em um repositório que podem ser obtidos de forma automatizada. Uma métrica subjetiva depende do objeto em questão e também do ponto de vista de quem está interpretando. A qualidade do código medida numa escala de 0% a 100% é subjetiva, pois apesar de definir um intervalo numérico, ainda depende do ponto de vista de quem está avaliando.

- **Quantitativa/Qualitativa:** métricas quantitativas pertencem a um intervalo de uma certa magnitude e geralmente são calculados por um número. Por exemplo, o número de defeitos encontrados ou a quantidade de linhas de código. Por outro lado, métricas qualitativas são representadas por valores representados por palavras, símbolos ou figuras em vez de números, como por exemplo o humor da equipe ou a satisfação do cliente.

Hartmann e Dymond (2006) sugerem outra categoria de classificação para projetos de desenvolvimento ágil. Eles distinguem métricas organizacionais, que medem a quantidade de valor de negócio entregue ao cliente e métricas de acompanhamento que fornecem informações que ajudam o time no entendimento e melhoria do processo que produz valor de negócio. As métricas organizacionais podem envolver questões de direcionamento econômico definidos por executivos da empresa. No entanto, em projetos de código aberto, em que o objetivo final nem sempre é financeiro, outros fatores de sucesso devem ser utilizados como o número de usuários ou sua satisfação (Sato, 2007). Métricas de acompanhamento auxiliam por meio de medições locais para que a equipe atinja os objetivos mais amplos definidos por métricas organizacionais.

4.2 Abordagens para escolha de Métricas

Nessa seção são apresentadas duas abordagens para escolha de métricas. Essas abordagens foram utilizadas na lista de verificação para definição de métricas propostas neste trabalho, que será discutido na Seção 4.4.

4.2.1 Abordagem GQM (*Goal Question Metric*)

A abordagem GQM baseia-se na suposição de que para se medir de maneira eficaz alguns objetivos devem ser estabelecidos para que estes sirvam de rota para o estabelecimento de questões que orientarão a definição de métricas em um contexto particular. A abordagem GQM, definida por Basili et al. (1994) é um método bastante aceito pela comunidade de engenharia de software para estudos experimentais e tem sido adotado para medir e melhorar a qualidade em organizações de desenvolvimento de software. Essa abordagem define um conjunto de métricas sob uma abordagem *top-down*:

- **Objetivo (conceitual):** são identificados objetivos para o produto/processo/recurso.
- **Pergunta (operacional):** determina questões que auxiliem na caracterização do objeto de estudo e como ele deve ser enxergado dentro do contexto de qualidade.
- **Métrica (quantitativo):** definição de métricas que irão fornecer uma resposta quantitativa de cada questão. Métricas podem ser objetivas ou subjetivas).

Um exemplo do uso do GQM poderia ser um gerente de projeto que tenha o objetivo de entregar um produto de software que atenda às expectativas do cliente daquela funcionalidade. Uma pergunta poderia ser, quanto o software entregue ao usuário desvia dos requisitos do cliente. Poderiam ser derivadas algumas métricas para responder essa pergunta, como por exemplo a quantidade de defeitos encontrados pelo cliente ou a grau de satisfação do mesmo.

4.2.2 Abordagem Lean

O *Lean Software Development* (Poppendieck e Poppendieck, 2003, 2006) pode ser sintetizado em sete princípios e um dentre esses é o "otimizar o todo". Esse princípio faz algumas observações sobre métricas, uma delas é de que as medições não devem preocupar apenas com aspectos locais. Um exemplo citado em (Poppendieck e Poppendieck, 2003) é relacionado a uma equipe de testes que deseja manter uma alta taxa de trabalho, evitando a contratação de novos testadores que abaixem essa alta taxa de trabalho. Apesar da taxa de trabalho se manter alta, outros setores da equipe terão que esperar bastante tempo para os

testes, aumentando o tempo de um ciclo de desenvolvimento, reduzindo o *feedback*, e gerando produtos de menor qualidade, com mais defeitos.

Além disso, o Lean sugere que as métricas não devem ser utilizadas para avaliar apenas indivíduos. Essa prática pode resultar em um indivíduo que se preocupa apenas em otimizar uma certa métrica, deixando de pensar no sistema como um todo. A avaliação deve levar em conta o time todo ou a empresa, gerando incentivos para que os indivíduos trabalhem de forma colaborativa para atingir um resultado comum. Métricas organizacionais devem ser reduzidas para contemplar objetivos amplos, e as métricas de acompanhamento devem ocultar o desempenho individual, não gerando incentivos errados.

4.3 Métricas de acompanhamento para Testes Ágeis

Hartmann e Dymond (2006) definem um conjunto de sugestões para que uma equipe de desenvolvimento ágil defina boas métricas para sua equipe. De forma geral, os autores sugerem que as métricas reforcem princípios ágeis de colaboração com o cliente, entrega de valor, simplicidade e não se preocupem com os números gerados pelas métricas e sim com as tendências demonstradas. Outro ponto que os autores destacam são pontos que são a minimização do número de métricas (já discutido pelo método Lean) e a facilidade para serem coletadas (de preferência utilizando ferramentas). Também deve ser deixado claro os fatores que influenciam essa métrica, evitando manipulações e facilitando a melhoria. Essas métricas também devem fornecer um *feedback* frequente e regular. Além disso, as métricas devem ser atualizadas, discutidas e disponibilizadas na área informativa, encorajando a melhoria e um alto nível de qualidade.

Métricas ágeis fornecem um guia para a equipe ágil, medindo o progresso do projeto e procurando apontar quando a equipe está desviando dos objetivos da equipe, ou fornecendo o *feedback* de que a equipe está no caminho correto. Crispin e Gregory (2009) afirmam que métricas podem nos alertar a respeito de problemas, mas analisadas de forma isolada geralmente não fornecem valor algum. Um exemplo desse cenário é a diminuição da cobertura de código de um projeto, que pode gerar preocupações se for utilizada de forma isolada. Uma das razões da diminuição da cobertura de código, por exemplo, poderia ser pela remoção de código que não estava sendo utilizado e estava coberto por testes. Se removermos esse código e os seus respectivos testes, a cobertura irá diminuir. Se utilizadas da forma correta, métricas podem estimular o time, que também deve focar na qualidade, não somente em números.

No contexto ágil, métricas de qualidade de código e de projeto oferecem conselhos objetivos, por exemplo, na identificação de áreas da aplicação candidatas à refatoração, assim como métricas de cobertura de código fornecem um guia necessário para o TDD e também à refatoração (Knoernschild, 2006).

Alguns trabalhos relacionados propõem métricas para projetos ágeis, um deles é um *framework* de avaliação de projetos em XP (*XP Evaluation Framework* – XP-EF) que sugere diversas métricas para projetos XP (Williams et al., 2004a,b). O *framework* define fatores de contexto, que envolvem aspectos ligados a classificação do software, aspectos sociológicos, específicos do projeto, ergonomia e tecnológicos. Também são utilizadas métricas de resultado, ligadas a objetivos de negócio e métricas de aderência que é composto por medidas objetivas e subjetivas. Algumas das métricas propostas por esse trabalho foram utilizadas para caracterização dos projetos e também para a definição das métricas de testes ágeis.

Além de Hartmann e Dymond (2006) que propuseram uma lista de verificação baseada na abordagem GQM e na abordagem Lean, para criação de métricas ágeis. Essa lista de verificação (descrita na tabela 4.1) foi utilizada para a definição das métricas de teste apresentadas neste trabalho. Sato (2007) propõe um conjunto de métricas organizacionais e de acompanhamento para testes ágeis e aplica essas métricas a um estudo de caso com projetos acadêmicos e governamentais. Colette (2009) também apresenta algumas métricas para projetos ágeis, criadas a partir de características de métodos ágeis citadas pelo manifesto ágil.

Diversos trabalhos avaliam projetos ágeis por meio de métricas de programas orientados a objeto (Ambu et al., 2006; Sato et al., 2007b). Também há trabalhos que propõe a automação de métricas de software, como por exemplo a ferramenta GERT *(Empirical Reliability Estimation and Testing Feedback Tool*) uma ferramenta de apoio ao TDD por meio do acompanhamento métricas de teste, que apoiam os ciclos de *feedback* criados pelos testes contínuos (Davidsson et al., 2004).

Na próxima Seção serão apresentadas e categorizadas as métricas de acompanhamento de testes propostas neste trabalho.

4.4 Métricas de Teste Ágil

As métricas de teste de software propostas neste trabalho foram criadas a partir de uma lista de verificação, proposta por Hartmann e Dymond (2006), que é apresentada na Tabela 4.1. O objetivo dessas métricas é facilitar o acompanhamento da atividade de teste, visando a melhoria contínua dos testes e consequentemente a qualidade do código-fonte, com menos defeitos, com um bom *design* e mais fácil de ser mantido. As métricas foram categorizadas em métricas de apoio ao teste de unidade, métricas de apoio ao teste de aceitação e métricas de teste gerais. Algumas das métricas foram propostas nesse trabalho, outras métricas foram adotadas de outros trabalhos e descritas segundo a lista de verificação da Tabela 4.1. A origem das métricas de acompanhamento de teste deste trabalho e quais métricas foram implementadas pela ferramenta ATMM são resumidas na Tabela 4.2.

Tabela 4.1: Lista de verificação para o planejamento de uma métrica ágil(Hartmann e Dymond (2006))

Característica	Descrição
Nome	Deve ser bem escolhido para evitar confusão ou ambiguidade.
Classificação	Objetiva / Subjetiva, Quantitativa / Qualitativa e Organizacional / Acompanhamento.
Objetivo	Motivação, preocupação ou tópico, um objeto e um ponto de vista (GQM).
Pergunta	Associada a uma pergunta específica (GQM).
Base de medição	Uma clara definição das medidas utilizadas para o cálculo da métrica.
Suposições	Devem ser identificadas para um claro entendimento do que os dados estão representando.
Tendência esperada	Uma idéia do comportamento esperado para a métrica.
Quando utilizar	Deve esclarecer os motivos que levaram à criação da métrica e, caso a métrica já tenha sito utilizada anteriormente, mostrar um pouco do seu histórico.
Quando parar de utilizar	É importante saber até quando uma métrica será útil, antes mesmo de utilizá-la, principalmente para métricas de acompanhamento.
Formas de manipulação	Deve esclarecer como as pessoas tentarão alterar seu comportamento em função da métrica gerar números "mais favoráveis".
Cuidados e observações	Recomendações sobre outras métricas similares, limites no uso e perigos associados à má utilização da métrica.

Tabela 4.2: Origem das métricas de acompanhamento de teste adotadas no trabalho

Métrica	Origem	Implementada
M1. Cobertura de Código	XP-EF (Williams et al., 2004a) e Nagappan et al. (2005)	SIM
M2. Fator de Teste	Extraída de Sato (2007)	SIM
M3. Quantidade de Casos de Teste e Assertivas	XP-EF (Williams et al., 2004a) e Nagappan et al. (2005)	SIM
M4. Porcentagem de Assertivas de Teste de Unidade Passando e Falhando	Proposta neste trabalho	SIM
M5. Quantidade de Testes de Aceitação por Funcionalidades	(Nagappan, 2004)	NÃO
M6. Porcentagem de assertivas de teste de aceitação passando e falhando	Proposta neste trabalho	NÃO
M7. Funcionalidades Testadas e Entregues (*Running Tested Features* ou RTF)	Extraída de Sato (2007)	NÃO
M8. Tempo de Execução de Testes	Proposta neste trabalho	SIM
M9. Quantidade de Defeitos Encontrados	XP-EF (Williams et al., 2004a)	NÃO

4.4.1 Métricas de apoio a Testes de Unidade

As métricas de apoio a testes de unidade auxiliam diretamente os desenvolvedores que utilizam testes de unidade e a estratégia TDD. As métricas de *"cobertura de código"* e *"quantidade de casos de teste e assertivas"* já são empregadas em projetos ágeis e tradicionais. Essas duas métricas foram utilizadas no XP-EF (Williams et al., 2004a) e também nas métricas propostas por Nagappan et al. (2005). O *"fator de teste"* foi uma métrica apresentada em (Sato, 2007) e (Nagappan et al., 2005). A *"cobertura de código"*, pode apoiar a estratégia TDD, guiando a construção e execução de casos de teste conforme a cobertura atingida. A *"quantidade de casos de teste e assertivas"* fornece instrumentos para que a equipe de desenvolvimento verifique a evolução dos testes de unidade automatizados. Por fim, o *"fator de teste"* evidencia o esforço de uma equipe na criação de testes de unidade, comparando-se com a quantidade de código produzido. Essas três métricas são descritas a seguir.

M1. Cobertura de Código

- **Classificação:** Objetiva, Quantitativa e Acompanhamento
- **Objetivo:** Atingir a cobertura de código desejada, contribuindo com a qualidade do código que está sendo produzido e do código de testes.
- **Pergunta:** Qual a cobertura de código-fonte utilizando-se critérios estruturais (Fluxo de Controle e Fluxo de Dados).
- **Base da medição:** No caso de critérios estruturais, para cada critério será dado por iteração: (i) o número de requisitos de teste, (ii) a quantidade de elementos cobertos e (iii) a porcentagem de cobertura.
- **Suposições:** A equipe está desenvolvendo código de produção e código de testes para verificar a possível existência de defeitos no código de produção. Além disso, procura-se determinar o percentual de elementos requeridos estabelecidos por um critério de teste que foram executados pelo conjunto de casos de teste.
- **Tendência esperada:** Se a equipe está utilizando TDD com passos pequenos, a tendência é que cada trecho de código possua um caso de teste associado, logo a cobertura deve ser alta. Caso a equipe não esteja se preocupando com a criação de casos de teste associada à criação de código de produção a cobertura será baixa.
- **Quando utilizar:** Quando a equipe estiver preocupada com a qualidade do código produzido (ex.: encontrar defeitos no código por meio da criação de casos de teste em trechos de código ainda não testados) e o código de testes (ex.: identificar casos de teste redundantes que não aumentam a cobertura ou a falta de casos de teste). Também

podem ser utilizados para remover código não utilizado pela aplicação. A cobertura pode ser utilizada paralelamente ao ciclo de desenvolvimento utilizando o TDD e a refatoração, auxiliando na criação de novos casos de teste ou verificando trechos de código refatorados que devem continuar sendo cobertos por meio de testes.

- **Quando parar de utilizar:** É sempre importante que a equipe verifique a qualidade do código de produção e de testes desenvolvido. Caso decidam parar de utilizar esta métrica deverá ser substituída por outra métrica que também avalie a qualidade dos casos de teste produzidos.

- **Formas de manipulação:** Uma das maneiras de manipular a métrica é adicionar trechos código de produção não utilizados pela aplicação com o objetivo de criar requisitos de teste facilmente cobertos por um caso de teste. Se a ferramenta utilizada para medir a cobertura possibilitar a seleção de requisitos como não alcançáveis, a cobertura poderá ser manipulada.

- **Cuidados e observações:** Trechos de código não exercitados indicam pontos não seguros (não testados), no entanto trechos de código exercitados não garantem a ausência de defeitos. Assim, a utilização de diversas técnicas é recomendada. A cobertura de código legado muitas vezes pode ser custosa, se tornando inviável a equipe atingir uma alta taxa de cobertura neste código. Além disso, mesmo com uma alta cobertura o código de produção pode não fazer exatamente o que o cliente requisitou.

M2. Fator de Teste

- **Classificação:** Objetiva, Quantitativa e Acompanhamento

- **Objetivo:** Verificar a evolução do código de teste e o esforço para criação de testes.

- **Pergunta:** Qual a relação entre tamanho do código de teste e tamanho do código de produção?

- **Base da medição:** O fator de teste T_i para a iteração i é calculado como a razão entre o número de linhas de código de teste e o número de linhas de código de produção:

$$T_i = \frac{TLOT_i}{TLOC_i}$$

  onde:
  $TLOT_i$ = número total de linhas de código de teste na iteração i
  $TLOC_i$ = número total de linhas de código de produção na iteração i

- **Suposições:** A equipe está desenvolvendo código de produção e código de testes automatizados para verificar a qualidade do software produzido.

- **Tendência esperada:** Se uma equipe utiliza técnicas como TDD desde o início do projeto, o fator de testes será alto (em muitos casos inclusive com valor acima de 1). Para equipes onde os testes não são desenvolvidos junto com o código de produção, o fator de testes será provavelmente baixo e a tendência é que utilizando essa métrica, a equipe passe a se preocupar em criar mais código de testes.

- **Quando utilizar:** Quando a equipe estiver preocupada com a qualidade do software produzido que envolve a preocupação com a criação de casos de teste. Além disso, esta métrica ajuda a monitorar a melhoria de práticas como o TDD, a integração contínua, o *design* incremental ou a refatoração.

- **Quando parar de utilizar:** Caso decidam parar de utilizar o Fator de Teste, é importante substituí-lo por outra métrica para avaliar evolução do código de teste e o esforço para criação de testes e mantê-la alta.

- **Formas de manipular:** Essa métrica pode ser manipulada por meio da produção de muito código de teste que não verifica o comportamento esperado do sistema (testes sem assertivas, por exemplo). Da mesma forma, medir linhas de código pode gerar um incentivo para tornar o código menos legível, evitando quebras de linha, o que afetaria essa métrica.

- **Cuidados e observações:** Um fator de teste maior que um não representa necessariamente o cenário ideal, assim como não garante que o código está totalmente testado ou com boa qualidade. Apesar de influenciar a qualidade do software produzido, essa métrica não serve como medida objetiva de qualidade.

M3. Quantidade de Casos de Teste e Assertivas

- **Classificação:** Objetiva, Quantitativa e Acompanhamento

- **Objetivo:** Acompanhar a evolução da quantidade de casos de teste e assertivas produzidas pelos desenvolvedores.

- **Pergunta:** Qual a quantidade de casos de teste e assertivas criadas em uma determinada iteração em relação à quantidade de código produzido

- **Base da medição:**

$$\frac{TCT_i}{TLOC_i} \quad e \quad \frac{TA_i}{TLOC_i}$$

onde:

TCT_i = número total de casos de teste na iteração i

TA_i = número total de assertivas na iteração i

$TLOC_i$ = número total de linhas de código de produção na iteração i

- **Suposições:** O desenvolvedor sempre deverá criar novos testes associados ao código de produção desenvolvido. A funcionalidade estará pronta quando a equipe integrar todo o código ao repositório, sendo que o código integrado deve passar por todo conjunto de casos de testes desenvolvidos.

- **Tendência esperada:** A quantidade de casos de teste e assertivas deve crescer proporcionalmente à quantidade de código produzido na iteração. Essa quantidade também pode ser influenciada pela utilização de métricas como a cobertura de código, que pode aumentar ou diminuir a quantidade de casos de teste e assertivas conforme a análise de cobertura do código.

- **Quando utilizar:** Quando a equipe desejar acompanhar a automação e evolução dos testes de unidade ou o TDD.

- **Quando parar de utilizar:** Apenas quando o projeto terminar ou quando a equipe acreditar que a prática do TDD, principalmente em relação à criação de casos de teste já está sendo utilizada de forma madura pela equipe de desenvolvimento.

- **Formas de manipular:** Podem ser criados diversos casos de teste ou assertivas que não são úteis para verificar a presença de defeitos no código de produção.

- **Cuidados e observações:** A quantidade de testes ou assertivas pode diminuir em iterações de manutenção/refatoração de código. Essa métrica deve ser utilizada juntamente com a métrica de cobertura, para não apenas garantir um grande conjunto de casos de teste, mas também um conjunto de teste de qualidade. Esses casos de teste devem ser complementados pelo teste de aceitação que irá verificar em mais alto-nível se as funcionalidades implementadas atendem às necessidades do cliente.

M4. Porcentagem de Assertivas de Teste de Unidade Passando e Falhando

- **Classificação:** Objetiva, Quantitativa e Acompanhamento

- **Objetivo:** Verificar qual a porcentagem de assertivas dos casos de teste de unidade/integração que estão passando ou falhando.

- **Pergunta:** Qual a porcentagem de assertivas de testes de unidade/integração que estão passando/falhando?

- **Base da medição:** A porcentagem de assertivas passando ou falhando por iteração (i) será dada por:

$$\text{Assertivas Passando}_i\ \% = (\text{N}^o \text{ de assertivas passando}_i\ /\ \text{N}^o \text{ total de assertivas}_i)$$

ou

$$\text{Assertivas Falhando}_i\ \% = (\text{N}^o \text{ de assertivas falhando}_i\ /\ \text{N}^o \text{ total de assertiva}_i)$$

- **Suposições:** A equipe está desenvolvendo código-fonte para determinadas funcionalidades do sistema que possuem testes de unidade/integração associados. Para cada caso de teste de unidade devem ser criadas assertivas suficientes para verificar o comportamento (entrada/saída esperada) de uma unidade ou a integração dela com outras unidades.

- **Tendência esperada:** Todo o código de produção deve passar pelos casos de teste e consequentemente suas assertivas.

- **Quando utilizar:** Essa porcentagem deve ser utilizada para acompanhamento dos testes de unidade, podendo verificar a existência de defeitos em novas funcionalidades ou em funcionalidades alteradas durante a iteração.

- **Quando parar de utilizar:** Apenas quando o projeto terminar, pois sempre a equipe de desenvolvimento deve acompanhar os resultados da execução dos testes de unidade/integração para verificar a existência de defeitos no código de produção.

- **Formas de manipular:** Podem ser criadas diversas assertivas que não melhorem a qualidade dos testes simplesmente para aumentar a porcentagem de assertivas corretas.

- **Cuidados e observações:** Mesmo se todas as assertivas estão passando, isso não garante que o teste de unidade e consequentemente a funcionalidade funcionam de acordo com o desejo do cliente. Assertivas falhando podem ser consequência de um defeito no código desenvolvido como também defeito na assertiva.

4.4.2 Métricas de Teste de Aceitação (*Business Testing*)

Para acompanhamento do teste de aceitação foram propostas três métricas. A primeira métrica é a *"quantidade de testes de aceitação por funcionalidades"*. Essa métrica foi proposta por (Nagappan, 2004), e cada funcionalidade representa uma história do cliente no método XP. A segunda métrica é a *"porcentagem de assertivas de teste de aceitação passando e falhando"*, essa métrica foi proposta por esse trabalho para que a equipe possa guiar a evolução do projeto por meio dos testes de aceitação que já passaram. Por fim, a última métrica para testes de aceitação é a métrica de *"funcionalidades testadas e entregues - RTF"*, essa métrica foi proposta por Jeffries (2004) e também utilizada por Sato (2007). A RTF procura acompanhar as funcionalidades que passaram pelo teste de aceitação e que podem ser entregues ao cliente.

M5. Quantidade de Testes de Aceitação por Funcionalidades

- **Classificação:** Objetiva, Quantitativa e Acompanhamento
- **Objetivo:** Acompanhar a evolução da quantidade de testes de aceitação produzidos.
- **Pergunta:** Qual a quantidade de casos de teste de aceitação por funcionalidades criados em uma determinada iteração.
- **Base da medição:** Cada requisito representado por meio de histórias no XP e por meio do *backlog* do produto no Scrum são quebrados em tarefas que devem ser realizadas durante a iteração. Cada tarefa pode ter um ou mais casos de teste de aceitação que devem avaliar quando aquela tarefa está pronta.

$$\text{TTA}_i = \text{TCT}_i \text{ - } \text{TCT}_{i-1}$$

  onde:
  TTA_i = número de casos de teste de aceitação na iteração i
  TCT_i = total de casos de teste de aceitação na iteração i
  TCT_i = total de casos de teste de aceitação na iteração anterior (i-1)

- **Suposições:** O cliente deve trabalhar colaborativamente com a equipe de teste e desenvolvimento, definindo histórias ou itens do *backlog* e criando novos casos de teste de aceitação. A funcionalidade deve ser colocada em produção apenas quando passar pelo teste de aceitação.
- **Tendência Esperada:** A quantidade de casos de teste de aceitação deve crescer de forma semelhante à quantidade de novas histórias da iteração.
- **Quando Utilizar:** Para a equipe atingir uma melhor qualidade no entendimento dos requisitos, identificando possíveis defeitos em funcionalidades requisitadas pelo cliente. Além disso, serve como indicador do progresso do desenvolvimento, mostrando quais das funcionalidades já podem ser colocadas em produção.
- **Quando parar de Utilizar:** Apenas quando o projeto terminar ou quando a equipe utilizar outras técnicas para apoio à validação das funcionalidades implementadas de acordo com as necessidades do cliente.
- **Formas de Manipular:** Podem ser criados diversos casos de teste de aceitação não desnecessários para a validação de determinadas funcionalidades.

- **Cuidados e Observações:** A quantidade de novos testes de aceitação pode ser pequena em iterações de manutenção de código. Os testes de aceitação apesar de algumas vezes serem criados pelos desenvolvedores, devem sempre passar pela validação do cliente após a funcionalidade ter sido desenvolvida.

M6. Porcentagem de assertivas de teste de aceitação passando e falhando

- **Classificação:** Objetiva, Quantitativa e Acompanhamento
- **Objetivo:** Verificar qual a porcentagem de assertivas dos testes de aceitação que estão passando ou falhando.
- **Pergunta:** Qual a porcentagem de assertivas de testes de aceitação que estão passando e falhando?
- **Base da medição:** A porcentagem de assertivas passando ou falhando por iteração (i) será dada por:

$$\text{Assertivas Passando}_i\ \% = (\text{N}^o \text{ de assertivas passando}_i\ /\ \text{N}^o \text{ de assertivas total}_i)$$
$$\text{ou}$$
$$\text{Assertivas Falhando}_i\ \% = (\text{N}^o \text{ de assertivas falhando}_i /\ \text{N}^o \text{ de assertivas total}_i)$$

- **Suposições:** A equipe está desenvolvendo código-fonte para determinadas funcionalidades do sistema que possuem testes de aceitação associados. Para cada teste de aceitação devem ser criadas assertivas (exemplos) suficientes para validar uma funcionalidade.
- **Tendência esperada:** Todas as funcionalidades do sistema devem passar pelas assertivas criadas em cada teste de aceitação.
- **Quando utilizar:** Essa porcentagem deve ser utilizada para acompanhamento dos testes de aceitação, podendo verificar a existência de defeitos em novas funcionalidades ou em funcionalidades alteradas durante a iteração.
- **Quando parar de Utilizar:** Apenas quando o projeto terminar ou quando a equipe não achar necessário a utilização de testes de aceitação.
- **Formas de manipular:** Podem ser criadas diversas assertivas que não melhorem a qualidade dos testes simplesmente para aumentar a porcentagem de assertivas corretas.
- **Cuidados e observações:** Todas as assertivas passando não garante que o teste de aceitação e consequentemente a funcionalidade funcionam de acordo com o desejo do cliente.

M7. Funcionalidades Testadas e Entregues (*Running Tested Features* ou RTF) - Extraída de Sato (2007)

- **Classificação:** Subjetiva, Quantitativa e Organizacional. Subjetiva pois o cliente define quando uma funcionalidade está pronta por meio do teste de aceitação.

- **Objetivo:** Medir a quantidade de valor de negócio entregue em cada funcionalidade do sistema, sob o ponto de vista do cliente.

- **Pergunta:** Qual é a taxa de valor de negócio entregue por funcionalidade testada e implantada? Quando o software deixa de ser um requisito e passa a agregar valor?

- **Base da medição:** Requisitos são quebrados em histórias (funcionalidades). Casos de teste de aceitação são definidos pelo cliente e servem como critério para avaliar quando a história está pronta (testada). Toda funcionalidade pronta deve estar integrada, com a possibilidade de ser implantada e prontamente utilizada (entregue). Serão contabilizadas então a quantidade de funcionalidades testadas e entregues ao cliente por iteração:

  $$\mathrm{RTF}_i$$

  onde:
  RTF_i = Funcionalidades Testadas e Entregues por iteração

- **Suposições:** O cliente deve trabalhar em colaboração com a equipe, definindo histórias e cenários de aceitação. Uma funcionalidade testada e pronta só poderá trazer valor efetivo de negócio quando entrar em produção.

- **Tendência esperada:** RTF deve crescer linearmente logo após o início do projeto e até o seu término. As funcionalidades que trazem mais valor devem ser implantadas primeiro.

- **Quando utilizar:** Para avaliar a execução de projetos ágeis. Para entregar funcionalidades testadas a partir do início do projeto, a equipe vai precisar de práticas ágeis. Ela não terá tempo de projetar o software de forma ampla (*big design up-front*), assim como não poderá esquecer do teste automatizado.

- **Quando parar de Utilizar:** Quando o projeto terminar ou o retorno financeiro não justificar seu prolongamento.

- **Formas de manipular:** Uma forma de manipular essa métrica é por meio da entrega das funcionalidades mais fáceis, ao invés das mais importantes. Definir poucos casos de teste também fará com que a funcionalidade esteja pronta mais rapidamente.

- **Cuidados e observações:** Medir funcionalidade entregue pode não refletir diretamente em valor de negócio. O excesso de funcionalidades é, na verdade, um dos grandes inimigos do desenvolvimento de software. É importante que as funcionalidades entregues no início do projeto sejam as mais importantes e com maior valor de negócio.

4.4.3 Métricas de Teste Gerais

Duas métricas foram propostas para medirem o tempo de execução dos casos de teste e a quantidade de defeitos encontrados. A métrica de *"tempo de execução de casos de testes"* proposta nesse trabalho é bastante importante, já que qualquer tipo de teste deve ser automatizado e executado regularmente. O teste de unidade é executados a constantemente em ciclos do TDD, além disso, a velocidade de outros teste automatizado como o teste de aceitação, apesar de não serem executados com tanta frequência, também devem ter um tempo aceitável. A *"quantidade de defeitos encontrados"* é utilizada tanto por projetos tradicionais, como em projetos ágeis e também foi utilizada no *framework* XP-EF. Essa métrica procura quantificar o número de defeitos encontrados no software em produção, por meio do teste exploratório ou de aceitação. Tanto a métrica de tempo de execução como a métrica de quantidade de defeitos encontrados podem ser relacionadas com a quantidade de funcionalidades implementadas em uma iteração ou também o número de linhas de código produzidos. No entanto, no contexto deste trabalho essas métricas serão descritas de forma isolada.

M8. Tempo de Execução de Casos de Teste

- **Classificação:** Objetiva, Quantitativa e Acompanhamento
- **Objetivo:** Medir o tempo de execução do conjunto de testes.
- **Pergunta:** Qual o tempo de execução de um determinado conjunto de teste (unidade, integração, aceitação) em uma determinada iteração.
- **Base da medição:**

$$\mathrm{TET}_i$$

  onde:
  TET_i = tempo de execução de casos de teste na iteração i (em segundos/minutos/horas)

- **Suposições:** O tempo tende a crescer conforme a quantidade de funcionalidades implementadas e consequentemente a quantidade de casos de teste. A equipe deve procurar não aumentar muito o tempo de execução dos casos de teste, que devem ser frequentemente executados em um tempo razoável.

- **Tendência esperada:** A tendência é que esse tempo de execução aumente conforme a complexidade dos módulos do sistema. Se houver a preocupação com a refatoração de código espera-se que esse tempo de execução diminua.
- **Quando utilizar:** Se a equipe desejar encontrar gargalos no sistema, melhorar o código e diminuir ou manter um tempo aceitável para execução dos testes.
- **Quando parar de utilizar:** Quando o projeto acabar ou quando a equipe reservar tempo para refatorar o código-fonte e de testes para otimizar a qualidade dos testes, o que inclui diminuir o tempo de execução.
- **Formas de manipular:** O desenvolvedor/testador deixar de criar mais testes para que o tempo de execução não aumente.
- **Cuidados e observações:** Se o tempo se estabilizar, ou não variar muito, não significa que a equipe chegou a um valor ótimo. A equipe sempre deve procurar desafiar os padrões atuais para encontrar formas de reduzir o tempo.

M9. Quantidade de Defeitos Encontrados

- **Classificação:** Objetiva, Quantitativa e Acompanhamento
- **Objetivo:** Uma equipe de desenvolvimento ágil deve se preocupar em encontrar defeitos do software antes que eles sejam entregues ao cliente. Quando encontrá-los, deve corrigir em um pequeno intervalo de tempo, relatando as causas do defeito e como ele foi solucionado para aprendizagem da equipe de desenvolvimento.
- **Pergunta:** Qual a quantidade de defeitos encontrados no software em produção, por meio de testes exploratórios ou de aceitação em uma determinada iteração?
- **Base da Medição:**

$$\text{Qtde. defeitos encontrados}_i$$

  onde:
  Qtde. defeitos encontrados$_i$ = Quantidade de defeitos encontrados em uma iteração i
  Os defeitos podem ser separados de acordo com sua severidade, de acordo com a fase/técnica de teste e também categorizados como resolvidos ou em aberto.

- **Suposições:** A equipe está desenvolvendo código-fonte que produza o menor número de defeitos, no entanto se forem detectados defeitos após a integração do código em seu repositório, por exemplo, por meio de testes exploratórios ou testes de aceitação, esses defeitos devem ser reportados e posteriormente corrigidos de acordo com a sua severidade.

- **Tendência esperada:** A quantidade de defeitos pode diminuir conforme a maturidade da equipe de desenvolvimento e da equipe de teste. No entanto, outros fatores podem interferir nessa métrica como, por exemplo, a complexidade das funcionalidades implementadas na iteração ou o tempo gasto pela equipe de teste em seções de teste exploratório. Idealmente, esse número deve ser baixo, uma vez que os métodos ágeis sugerem o uso de teste automatizado para prevenir defeitos e não para encontrá-los (Poppendieck e Poppendieck, 2003).

- **Quando utilizar:** Quando a equipe desejar medir e minimizar a quantidade de defeitos no software, antes de entregá-lo ao cliente.

- **Quando parar de utilizar:** É importante que a equipe sempre tenha o gerenciamento de defeitos encontrados, para que esses problemas sejam resolvidos. Além disso deve haver uma preocupação com a melhoria contínua do software desenvolvido, diminuindo a quantidade de defeitos.

- **Formas de manipular:** Corrigir o defeito sem reportá-lo, diminuir a severidade do defeito ou até mesmo ignorá-lo.

- **Cuidados e observações:** Defeitos devem ser corrigidos o quanto antes e a sua correção não deve afetar funcionalidades desenvolvidas.

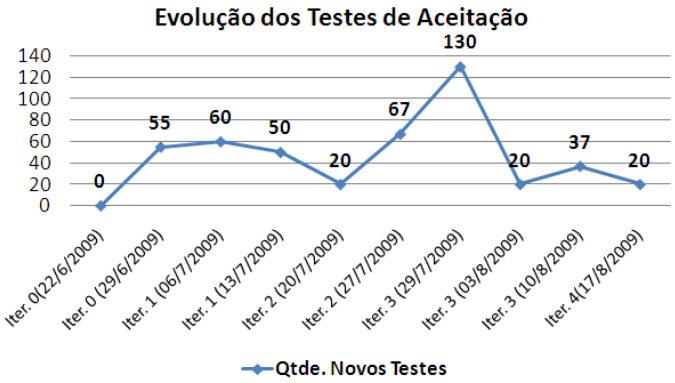
4.4.4 Área de Trabalho Informativa e Retrospectivas

Retrospectivas encorajam a discussão constante sobre o processo e forma de trabalho da equipe, devendo fazer parte do ciclo de inspeção e adaptação proposto pelos métodos ágeis. Todos os métodos ágeis possuem alguma prática ligada a retrospectivas: no XP ela se dá dentro dos ciclos semanais e trimestrais, no Scrum elas ocorrem em reuniões de revisão e retrospectiva, no FDD por meio de relatos e visibilidade de resultados, na família Crystal por meio de *workshops* de reflexão, no ASD revisões de qualidade, no DSDM por *workshops* com o cliente e no método Lean por meio do princípio de criação de conhecimento. Além disso retrospectivas rápidas podem ser feitas em reuniões diárias, presente no método Scrum.

Uma das práticas primárias do XP que se relaciona com a retrospectiva é a área de trabalho informativa. Essa prática sugere que a equipe forneça instrumentos informativos a respeito do andamento do projeto, de tarefas a serem feitas, informações técnicas sobre o desenvolvimento, além de apoiar a melhoria do processo e do produto. Isto pode incluir cartões de histórias da iteração em um mural ou quadros brancos, além de informações a respeito do projeto na forma de gráficos ou tabelas. Esses instrumentos informativos podem ser utilizados durante reuniões de planejamento ou reuniões diárias e também durante o andamento do ciclo iterativo de desenvolvimento.

Métricas para acompanhamento da atividade de teste podem ser exibidas utilizando-se gráficos de melhoria do processo e produto ou por meio de quadros informativos. De acordo com os objetivos do projeto, a equipe deve decidir quais métricas utilizar, como medir, interpretar e reagir perante os resultados obtidos e, por fim, até quando utilizar tais métricas. O objetivo de utilizar gráficos de melhoria do processo e produto é mostrar informações sobre o projeto de forma simples e não ambígua pelo ambiente de desenvolvimento. Esses gráficos podem exibir o quadro de planejamento da iteração e de versões, o calendário da equipe de desenvolvimento, mostrando datas importantes, número das iterações, histórias que estão sendo desenvolvidas e seus respectivos pontos etc.

Gráficos também podem medir áreas específicas que a equipe de desenvolvimento deseja melhorar. Como, por exemplo, a evolução de testes de aceitação e informações de cobertura de código (Figura 4.1). Essas áreas são selecionadas durante reuniões de retrospectiva e ao contrário de quadros de planejamento e calendário do time de desenvolvimento, esses gráficos são mantidos durante o tempo que a equipe achar necessário e devem ser utilizados de forma efetiva. A criação de gráficos de melhoria é uma decisão do time de desenvolvimento e deve ser mantida sempre atualizada. A atualização desses gráficos deve ocorrer conforme a frequência com que a equipe concordar e, geralmente, é feita pelo *tracker* do projeto.



Coverage Report - wire.jspbeans

Package	# Classes	Line Coverage	Branch Coverage	Complexity
wire.jspbeans	46	73%	86%	4.166

Classes in this Package	Line Coverage	Branch Coverage	Complexity
MacroSwitch	97%	78%	1.15
data_Bean	74%	98%	22.4
data_BeanHelper	94%	99%	4.162
data_BeanRetrieve	86%	72%	9
data_ImportBean	88%	100%	1.75
data_Loger	84%	100%	2.5
data_PermissionBean	100%	100%	2.5
data_RSSFilter	100%	100%	4
data_SendItAway	85%	94%	4.333
db_AdminBean	80%	99%	11.438
db_ClipBean	82%	100%	11.6
db_ConsoleBean	76%	93%	14.353

Figura 4.1: Gráfico de evolução do número de testes de aceitação criados por iteração e informações sobre cobertura de código.

4.5 Ferramenta ATMM (*Agile Testing Metrics Management Tool*)

A ferramenta *Agile Testing Metrics Management* (ATMM) complementa a proposta deste trabalho, tendo sido desenvolvida como parte de suas contribuições. Tem como objetivo apoiar a atividade de teste de software no contexto ágil, especificamente na fase de teste de unidade, em que os testes são gerenciados em cada iteração de desenvolvimento. O objetivo principal da ferramenta, além de gerenciar essas iterações, é exibir métricas relacionadas ao código que está sendo testado e aos casos de teste desenvolvidos utilizando a ferramenta JUnit (bastante utilizada em projetos ágeis). Além disso, a ferramenta exibe informações de cobertura a partir de informações extraídas da ferramenta *JaBUTi* (*Java Bytecode Understanding and Testing*) que utiliza critérios estruturais de fluxo de controle e fluxo de dados (Vincenzi et al., 2006; Vincenzi, 2004).

Com auxílio do módulo de inserção de casos de teste da ferramenta *JaBUTi* (que por sua vez utiliza a ferramenta JUNIT) e também da API Java Parser [1] são coletadas as métricas de teste descritas na Tabela 4.3. A maioria delas é coletada de forma automática (automatizada) e outras dependem da informação do usuário que está utilizando a ferramenta. Algumas das métricas coletadas pela ferramenta foram descritas na Seção 4.4, são elas: fator de teste, qtde de casos de teste, qtde de assertivas, casos de teste com sucesso/falha/ignorados e tempo total de execução dos casos de teste. O restante das métricas foram implementadas por serem facilmente coletadas e também contribuírem para a avaliação das iterações de desenvolvimento.

4.5.1 Arquitetura Geral e Funcionamento da Ferramenta

A ferramenta ATMM faz interface com a API Java Parser e a ferramenta *JaBUTi* (Vincenzi et al., 2006). A arquitetura geral da ferramenta é descrita a seguir e ilustrada pela Figura 4.2:

- **Coleta de informações do código fonte e testes:** a ferramenta utiliza a API *Java Parser* para coletar informações sobre o código-fonte e os casos de teste no formato da ferramenta JUnit, desenvolvidos em Java. Esta API utiliza um *parser* para Java 1.5 com geração da árvore sintática (*Abstract Syntax Tree* - AST) e suporte a visitação da AST. A AST grava a estrutura do código-fonte (Java), anotações utilizando *javadocs* e comentários. É possível também modificar os nodos da AST ou criar novos nodos para modificar o código-fonte. O *parser* foi criado utilizando o compilador java javacc.

[1] AST Java Parser (API) - Disponível em: `http://code.google.com/p/javaparser`

Tabela 4.3: Métricas de Acompanhamento de Testes implementada pela ATMM

Métrica	Descrição	Automatizada
# de Classes SUT	Quantidade de classes do sistema que está sendo testado (SUT)	SIM
# de Classes T	Quantidade de classes de teste	SIM
Métodos / Classes (SUT)	Quantidade de métodos por classe SUT	SIM
LOC / Classes (SUT)	Quantidade de linhas de código por classe SUT	SIM
LOC SUT e LOCT T	Quantidade total de linhas de código SUT e total de linhas de código de teste	SIM
Fator de Teste LOC SUT / LOC T	Quantidade de linhas de código SUT por linhas de código de teste	SIM
Qtde de casos de teste	Quantidade de casos de teste de unidade	SIM
Qtde de assertivas	Quantidade de assertivas de testes de unidade	SIM
Casos de teste com Sucesso/Falha/Ignorados	Quantidade de casos de teste JUnit que passaram, falharam ou que foram ignorados	SIM
Tempo Total	Tempo total de execução dos casos de teste JUnit	SIM
Tempo Médio de execução	Tempo médio de execução de casos de teste por classe de teste	SIM
Histórias ou itens do *Backlog* implementados	Quantidade de histórias ou itens do backlog que foram implementados	–
Horas ou Pontos implementados	Quantidade de horas ou pontos implementados conforme o valor de cada história ou item do *backlog*	–
Duração da Iteração	Duração da Iteração em dias ou semanas	–

- **Execução dos casos de teste e coleta de informações de cobertura:** utilizando a ferramenta *JaBUTi*, a ferramenta ATMM executa os casos de teste, coleta informações sobre essa execução dos testes JUnit e também informações de cobertura de código. As informações de cobertura são coletadas pelos relatórios XML gerados pela ferramenta, sendo que a geração desses relatórios é feita pelo testador, que pode executar a ferramenta *JaBUTi* pelo modo gráfico ou por linha de comando. Para facilitar a geração desses relatórios, podem ser utilizados *shell scripts*, que a partir de um arquivo de configuração, instrumentam, executam o código e consolidam as informações de cobertura. Esses *scripts* foram criados no contexto do projeto Qualipso (*Quality Plataform for Open Source*) (Qualipso Project, 2006) para facilitar a geração de informações de cobertura de projetos que tenham uma grande quantidade de classes Java.

- **Gerenciamento das iterações:** as métricas coletadas são armazenadas a cada iteração de desenvolvimento.

O primeiro passo da ferramenta ATMM (ilustrado na Figura 4.3) é a criação do projeto e a seleção de informações gerais do projeto. Nesse passo o testador no início do projeto deverá inserir informações que envolvem características gerais do projeto, a sua descrição, as estratégias ou técnicas de teste que serão utilizadas no projeto, além de outras práticas ágeis que o projeto utiliza, como programação em pares ou refatoração.

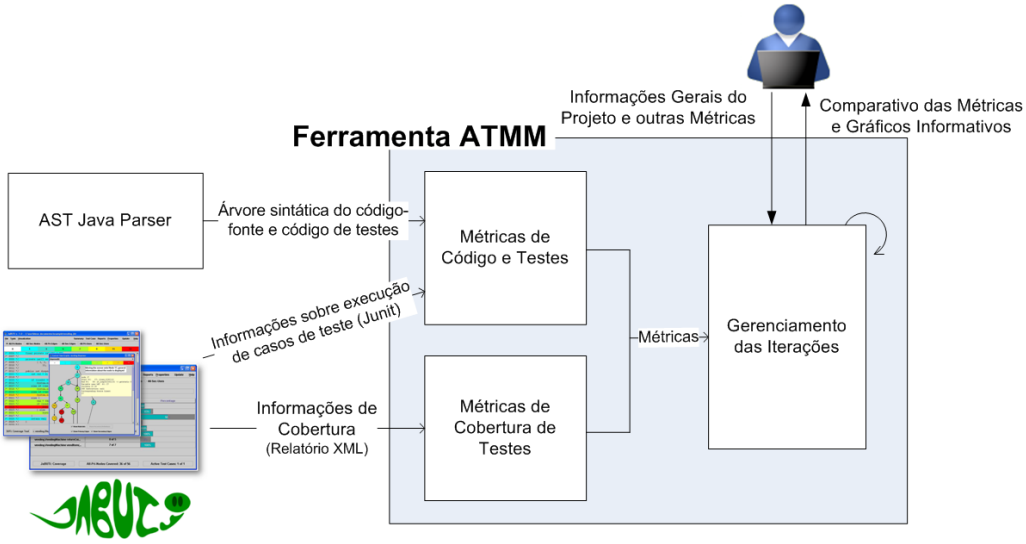


Figura 4.2: Arquitetura Geral da Ferramenta ATMM

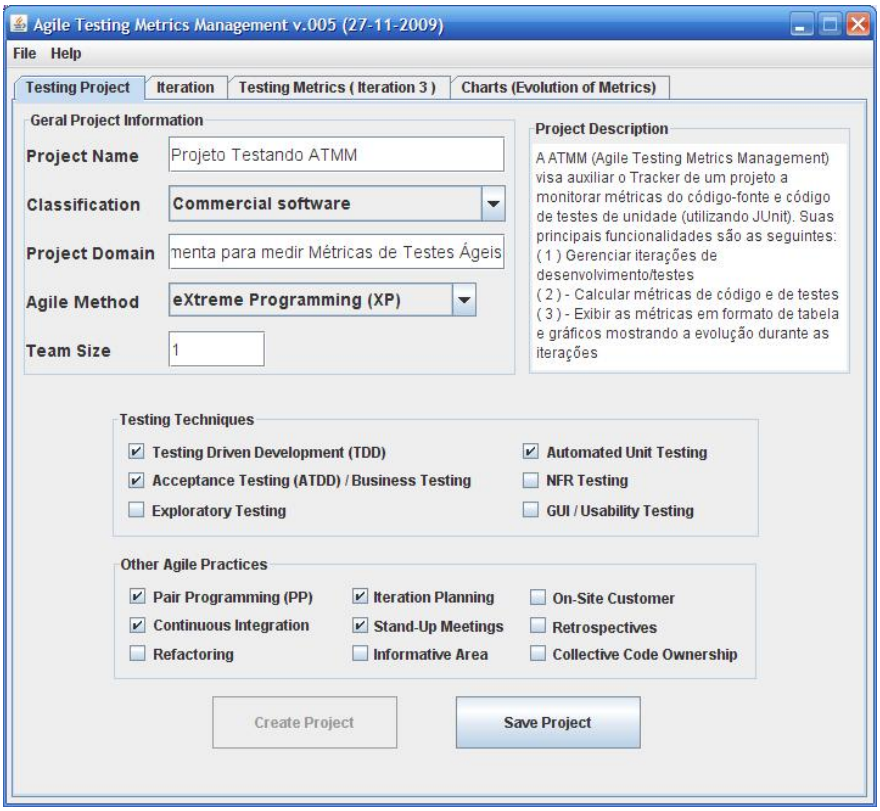


Figura 4.3: Ferramenta ATMM - Criação e Informações gerais do Projeto (Tela Principal)

Após a criação do projeto, o segundo passo (Figura 4.4) é inserir informações sobre cada iteração de desenvolvimento e testes. Ao fim de cada iteração o testador deverá empacotar o código-fonte e código de teste em diretórios específicos e esses caminhos deverão ser informados a ferramenta. Cada iteração terá sua data de início e fim, histórias ou itens de *backlog* implementados, horas ou pontos implementados, os caminhos de diretórios para as classes que estão sendo testadas e os conjuntos de casos de teste. Além disso podem ser fornecidos os caminhos de bibliotecas necessárias para executar os casos de teste. Nesse passo o desenvolvedor deverá executar os *scripts* da *JaBUTi* para gerar a cobertura de código-fonte produzido na iteração atual. O caminho do diretório em que se encontra o relatório XML gerado pela ferramenta *JaBUTi* deverá ser informado a ferramenta ATMM.

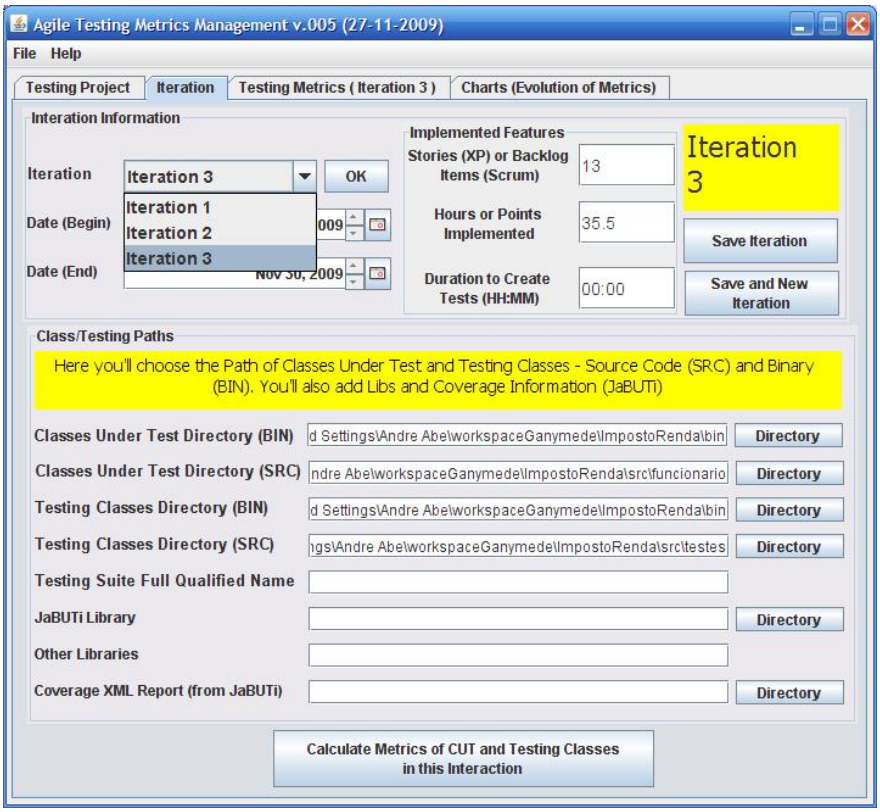


Figura 4.4: Ferramenta ATMM - Informações sobre cada Iteração de desenvolvimento e testes

O terceiro passo (Figura 4.5) irá exibir as métricas relacionadas ao código que está sendo testado, o código de testes e também uma tabela comparativa que mostra os valores das métricas em cada iteração. Esses dados podem ser exibidos na forma de gráficos comparativos conforme ilustrado na Figura 4.6.

Ao final de cada iteração conforme os resultados das métricas apresentadas pela ferramenta, o testador poderá desenvolver mais casos de teste ou refatorar os casos de teste existentes. A equipe pode decidir se essa tarefa será realizada antes da retrospectiva da iteração ou será realizada na próxima iteração. Por fim, o resultado das métricas deverá ser discutido durante a reunião de retrospectiva da iteração, e a equipe de desenvolvimento

e teste poderá estabelecer novas metas de qualidade do código-fonte e dos casos de teste para a próxima iteração baseando-se nas métricas obtidas nas iterações anteriores. O *tracker* deverá atualizar as informações a respeito das métricas na área de trabalho informativa com gráficos ou tabelas que mostrem a evolução das métricas durante as iterações . Caso a equipe de desenvolvimento, teste e os gerentes do projeto acharem necessário, as métricas poderão ser avaliadas diariamente e não somente no final de cada iteração.

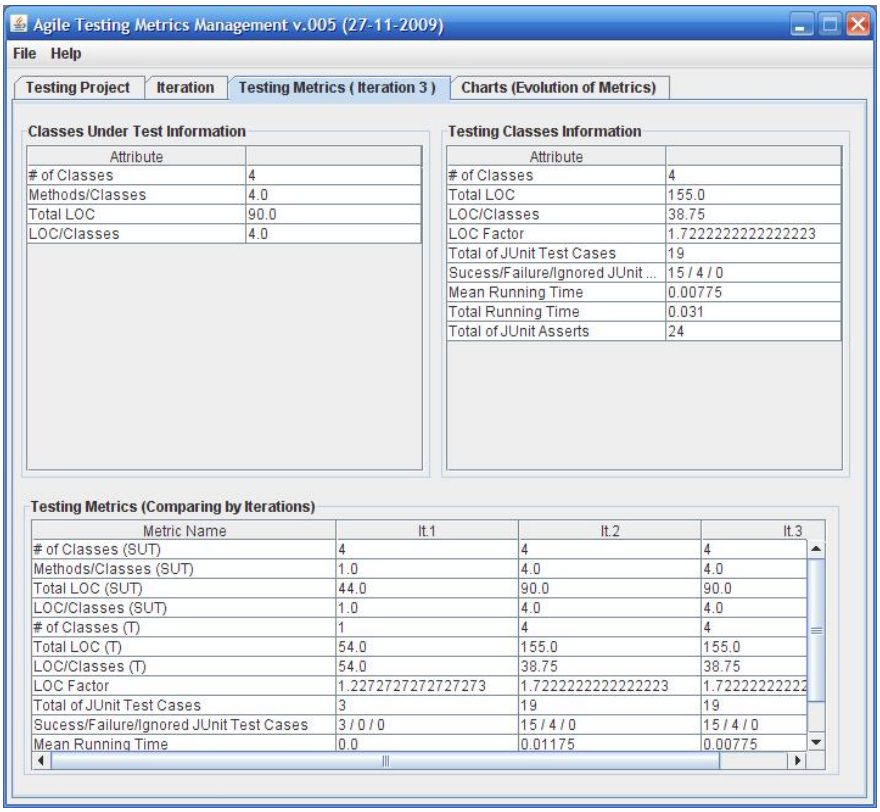


Figura 4.5: Ferramenta ATMM - Exibição comparativa das métricas

4.6 Considerações Finais

Nesse capítulo foram apresentados conceitos gerais ligados a métricas de software e abordagens para criação e seleção de métricas. Além disso, foram descritos o funcionamento e os benefícios decorrentes da utilização de métricas de acompanhamento em métodos ágeis, principalmente em relação a atividade de teste de software. Por fim, foi apresentada a ferramenta ATMM que implementa algumas das métricas propostas neste trabalho.

No próximo capítulo serão apresentados os estudos de casos deste trabalho que analisaram dois projetos que foram desenvolvidos utilizando métodos ágeis de desenvolvimento. Esses trabalhos são descritos a partir de um questionário, que foi aplicado e teve como principal objetivo coletar informações sobre como as práticas de teste de software e outras práticas ágeis que foram utilizadas nesses projetos. Além disso, foram coletadas e analisadas as métricas de código-fonte e código de testes desses projetos.

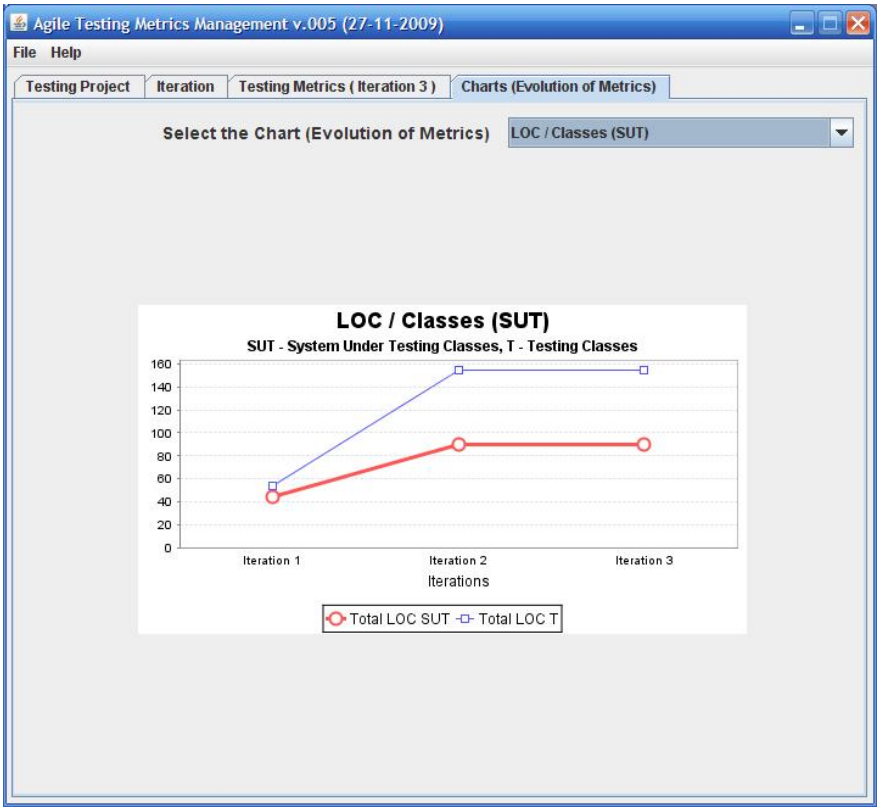


Figura 4.6: Ferramenta ATMM - Exibição comparativa das métricas

Capítulo

5

Estudo de Caso

Métodos de desenvolvimento ágil, segundo alguns de seus princípios descritos no manifesto ágil, devem promover o desenvolvimento sustentável, cuidar continuamente da excelência técnica e produzir um bom *design* do projeto. Além disso, em intervalos regulares, o time deve refletir sobre como se tornar mais eficiente, refinando e ajustando seu comportamento apropriadamente. Nesse sentido, Sato (2007) afirma que métodos ágeis devem gerar um *feedback* frequente para que os membros da equipe entendam o andamento atual do projeto e para guiá-los em direção a um ambiente de melhoria contínua. Para atingir esse ambiente, uma das práticas sugeridas é a utilização de um ambiente informativo que deve incluir métricas de software. Essas métricas são coletadas durante todo o andamento do projeto e podem ser discutidas frequentemente em reuniões de projeto.

Este capítulo descreve dois estudos de casos aplicado em projetos de software que utilizam o método de desenvolvimento XP. O objetivo dos estudos de casos são avaliar a aplicação das métricas de teste de software propostas por esse trabalho. Essas métricas, como descrito no capítulo anterior, devem facilitar o acompanhamento da atividade de teste, visando à melhoria contínua dos testes e, consequentemente, a qualidade do código-fonte, com menos defeitos, com um bom *design* e mais fácil de ser mantido. Na Seção 5.1 é apresentado o método e quais as métricas avaliadas nos estudos de casos. Os projetos avaliados no estudo de caso serão descritos na Seção 5.2. Na Seções 5.3 e 5.4, são sintetizados, analisados e discutidos os resultados coletados desses projetos.

5.1 Método e Métricas Avaliadas

As métricas de acompanhamento de testes foram analisadas após as iterações terem sido desenvolvidas. Para coletar as métricas de forma automatizada, foram utilizadas as seguintes fontes de informação:

- **Ferramenta ATMM:** a partir da ferramenta ATMM foram coletadas algumas das métricas de teste propostas nesse trabalho, além de métricas relacionadas ao código-fonte de cada projeto. Não foram coletadas duas métricas: *"Histórias ou itens do Backlog implementados"* e *"Horas ou pontos implementados"*, pois essas métricas deveriam ser inseridas manualmente na ferramenta e essa informação não havia sido armazenada no decorrer do andamento dos dois projetos.

- ***Scripts* da ferramenta *JaBUTi*:** foram obtidas informações a respeito da cobertura de código conforme critérios estruturais de fluxo de controle e fluxo de dados a partir de um conjunto de *scripts*. Esses *scripts* foram criados no contexto do projeto Qualipso (*Quality Plataform for Open Source*) (Qualipso Project, 2006) para facilitar a geração de informações de cobertura de projetos de software a partir da ferramenta *JaBUTi*. Um guia de execução para os *scripts* da *JaBUTi* podem ser consultados no Apêndice B.

Também foram coletadas informações gerais sobre os dois projetos avaliados no estudo de caso, além de informações relacionadas a métricas de qualidade do código. O objetivo da coleta dessas informações foi contextualizar os resultados obtidos com as métricas de acompanhamento de teste proposta neste trabalho. Utilizou-se as seguintes fontes de informação:

- **Questionário:** outras métricas quantitativas e subjetivas foram coletadas a partir de um questionário apresentado no Apêndice A. Esse questionário foi baseado no *Extreme Programming Framework* (XP-EF) (Williams et al., 2004b) e será discutido na Seção 5.2. O questionário coletou desde informações gerais do projeto, passando pelo fator ergonômico, fatores tecnológicos, geográficos, sociológicos e específicos de cada projeto, além da aderência e maturidade da atividade de teste de software. Na próxima Seção serão caracterizados os dois projetos ágeis utilizados nesse estudo de caso a partir de informações obtidas pelo questionário.

- **Ferramenta Kalibro:** foram coletadas algumas métricas da ferramenta Kalibro para verificar a qualidade do código produzido no decorrer das iterações. Essas métricas são relacionadas e a qualidade dos métodos produzidos, e também métricas de coesão e acoplamento.

As métricas de acompanhamento de testes foram coletadas após a implementação, no entanto seria ideal que essas métricas fossem coletadas durante o desenvolvimento do projeto. Nesse sentido, a equipe de desenvolvimento poderia utilizar as métricas como apoio ao

feedback das iterações e melhoria contínua do projeto. As métricas poderiam ser utilizadas na área de trabalho informativa, durante as reuniões diárias, retrospectivas, planejamento e desenvolvimento das iterações.

5.2 Caracterização dos Projetos Ágeis

Foram selecionados dois projetos para o estudo de caso, sendo que eles utilizam licenças livres e o método de desenvolvimento XP desde o início do projeto. O primeiro projeto é a ferramenta Kalibro, que foi originada a partir do protótipo da ferramenta Crab (Meirelles et al., 2009). A ferramenta Kalibro [1] é uma ferramenta para configuração e interpretação de métricas de código-fonte, e vem sendo desenvolvida por alunos do IME-USP desde 2009 (6 meses). O outro projeto é a ferramenta Archimedes, que é um sistema CAD (*Computer Aided Design*) [2] que inclui atualmente as principais funcionalidades de desenho em 2D assim como manipulação vetorial dos elementos criados. O projeto Archimedes também vêm sendo desenvolvido por alunos do IME-USP e foi iniciado em 2006 (4 anos).

Nesta Seção serão apresentadas informações gerais de cada projeto, que foram coletadas a partir de um questionário subjetivo, apresentado no Apêndice A. O questionário teve como objetivo coletar dados gerais sobre projetos que utilizam métodos ágeis e relacionar essas informações com as métricas de acompanhamento de teste e métricas de qualidade. Esse questionário coletou informações gerais do projeto, informações referentes a práticas ágeis e diversos aspectos da atividade de teste de software. Essas informações coletadas pelo questionário foram baseados no *Extreme Programming Framework* (XP-EF) que é um arcabouço para avaliação de projetos desenvolvidos que utilizam práticas do XP. Foram propostos nesse trabalho duas novas categorias: a categoria de *"aderência à prática de teste de software"* e a categoria de *"maturidade da atividade de teste de software"*. Essas duas novas categorias foram baseadas no trabalho de Krebs (2002) e parte do questionário de métricas de aderência proposto por Sato (2007).

A aderência à prática de teste é calculada de 0 a 4, conforme as práticas das seguintes categorias:

- **Atividades de teste de unidade automatizados [0- Nenhuma das atividades e 4- Segue todas as atividades]**: (1) testes de unidade automatizados existem para o código de produção, (2) uma ferramenta é utilizada para medir a cobertura de código, (3) há uma forma automatizada de executar todo o conjunto de casos de teste para todo o programa, (4) todos os casos de teste de unidade são executados e passam quando uma tarefa é finalizada e antes de integrarem o código, (5) quando estão sendo consertados os defeitos do software, testes de unidade são utilizados para

[1] Ferramenta Kalibro (Projeto Mezuro) - Disponível em: `http://softwarelivre.org/mezuro/kalibro/`
[2] Projeto Archimedes - Disponível em: `http://www.archimedes.org.br`

capturar o defeito antes de ser reparado, (6) testes de unidade são refatorados, (7) testes de unidade são rápidos o bastante para serem executados com frequência.

- **Atividades de teste de aceitação (*Business Testing*) [0- Nenhuma importância e 4- Total importância]**: (1) testes de aceitação são utilizados para verificar uma funcionalidade do sistema e requisitos do cliente, (2) o cliente fornece o critério de aceitação, (3) o cliente usa os testes de aceitação para determinar o que foi terminado no fim de uma iteração, (4) o teste de aceitação é automatizado, (5) uma história não é considerada finalizada até que os testes de aceitação passem, (6) testes de aceitação são executados automaticamente toda noite, (7) um ambiente compatível com o ambiente do usuário final é utilizado para o teste.

- **Atividades do desenvolvimento dirigido a testes (TDD) [0- Não aplicado e 4- Totalmente aplicado]**: (1) código é desenvolvido somente após um teste de unidade (que falha) tenha sido escrito, (2) melhoria do código por meio de refatorações, (3) uso de padrões para criação de testes, buscando a testabilidade e qualidade dos testes, (4) os testes guiam o *design* do código-fonte, (5) todo código de produção é desenvolvido utilizando TDD.

A maturidade da atividade de teste de software é dividida em quatro categorias e elas são avaliadas da seguinte maneira:

- **Teste de unidade e TDD:** possui seis níveis, que vão desde o projeto que não possui nenhum teste formal (0) até o nível em que a equipe se preocupa com padrões para os testes, que incluem a preocupação com a testabilidade (5).

- **Teste de Aceitação e Teste de Sistema:** possue apenas dois níveis, o de teste de aceitação e o de teste de sistema. O nível é calculado conforme a porcentagem de opções selecionadas (de 0 a 5). Se forem selecionadas 50% das opções por exemplo, será atribuído um valor 2,5 e se não forem utilizados testes de aceitação e testes de sistema será atribuído o nível 0.

- **Aspectos de automatização do teste:** possui seis níveis que também serão calculados conforme a porcentagem de opções selecionadas (de 0 a 5). Se forem utilizados apenas testes manuais será atribuído o nível 0.

- **Processo de teste e melhoria contínua:** possui quatro níveis e o nível é calculado conforme a porcentagem de opções selecionadas (de 0 a 5). Se não forem utilizadas nenhuma das práticas sugeridas, será atribuído o nível 0.

5.2.1 Kalibro

A ferramenta Kalibro foi projetada para ser incorporada a qualquer ferramenta de métricas de código-fonte, estendendo essas ferramentas para fornecer um fácil entendimento na avaliação de qualidade do software analisado. A Kalibro permite que um usuário experiente em métricas de software especifique intervalos de aceitação para cada métrica fornecida pela ferramenta base de métricas e permite a criação de métricas customizadas a partir das métricas nativas da ferramenta base (Figura 5.2(a)). Além disso, a ferramenta permite a configuração de categorias e pesos de cada métrica (Figura 5.2(b)), e resultados agregados de todo o código-fonte ou resultados detalhados por classe (Figura 5.2(c)).

Para coleta das métricas a partir de códigos-fontes de múltiplas linguagens (atualmente testada com C, C++ e Java) a ferramenta Kalibro, integra a ferramenta Analizo. A ferramenta Analizo faz a análise do código fonte e relata informações úteis sobre ele, como a dependência entre módulos e métricas de código (incluindo métricas de código orientado a objetos). Futuramente, por meio do projeto Mezuro, a ferramenta Kalibro integrada com a Analizo será implementada em uma plataforma Web, incorporando outras funcionalidades e permitindo que usuários submetam e avaliem o seu código-fonte. A ideia da arquitetura geral do projeto Mezuro é ilustrada na Figura 5.1.

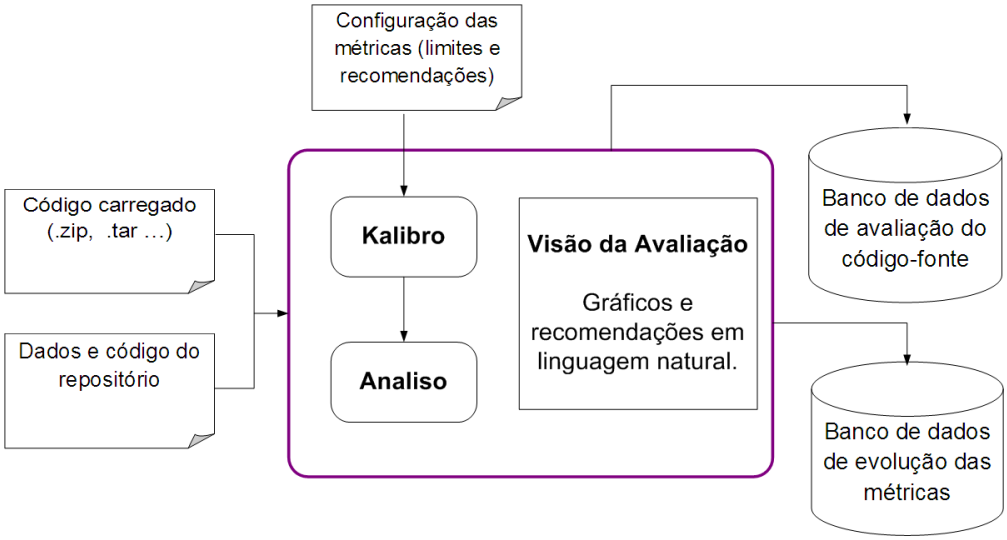
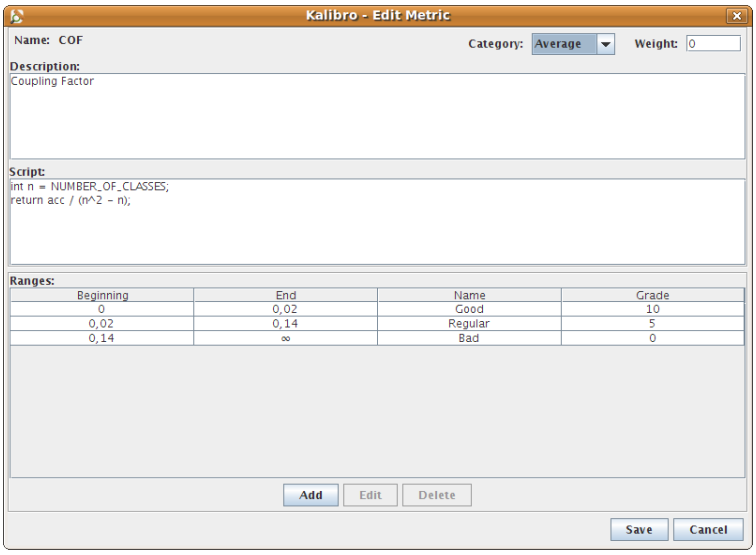
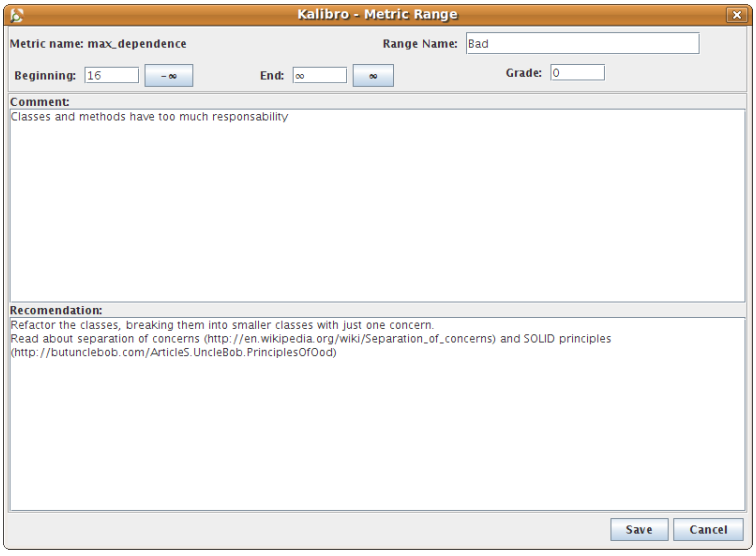


Figura 5.1: Arquitetura geral do projeto Mezuro

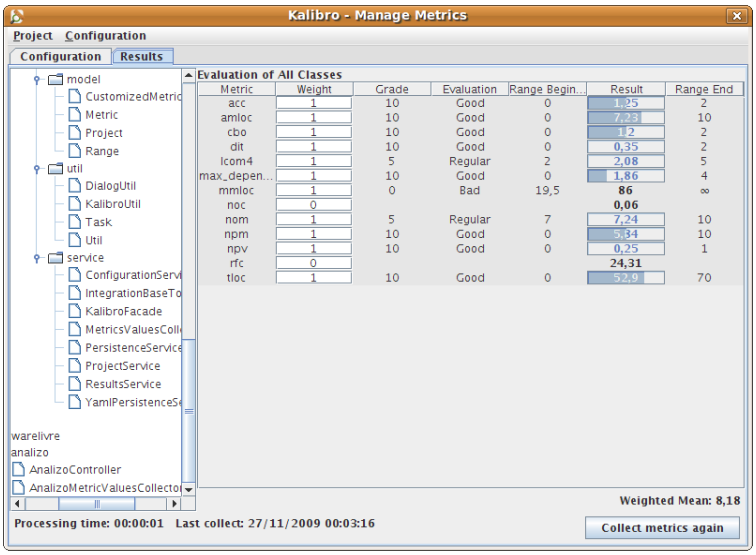
Na Tabela 5.1 são apresentadas informações gerais do projeto Kalibro e informações sobre fatores tecnológicos utilizados no projeto. O fator ergonômico, que trata a respeito da comunicação com o cliente, fatores geográficos (da equipe e dos clientes), fatores específicos e fatores sociológicos são apresentados na Tabela 5.2. Também foram coletadas informações sobre as práticas ágeis e práticas de teste utilizadas no projeto, além do nível de aderência à atividade de teste (Tabela 5.3). Por fim, são apresentadas informações a respeito da maturidade da atividade de teste na Tabela 5.4.



(a) Kalibro - Criação de métricas compostas



(b) Kalibro - Configuração de intervalos e recomendações



(c) Kalibro - Visualização dos resultados

Figura 5.2: Kalibro - Ferramenta de configuração e interpretação de métricas de código-fonte

Tabela 5.1: Kalibro - Informações gerais e fatores tecnológicos do projeto

Informações gerais do projeto	
Endereço Web do projeto	`http://softwarelivre.org/mezuro/kalibro`
Endereço do respositório de código	`svn://ccsl.ime.usp.br/kalibro/trunk/Kalibro`
Licença do software	Software livre
Classificação do Software	Software para usuário final
Fatores tecnológicos	
Método de desenvolvimento	Programação Extrema (XP)
Linguagens	Java (J2SE)
Tecnologias e/ou *frameworks* utilizados	-
Ferramentas (ambientes de desenvolvimento)	Eclipse
Ferramentas (gerenciamento de projeto)	Xplanner/Noosfero
FERRAMENTAS de teste de software	Ferramentas de Teste de Unidade (Família XUnit ou similares), Cobertura de Testes, Teste de GUI
Ferramentas/bibliotecas utilizadas (teste de software)	JUnit, Coverage, Fest
Outras ferramentas	Metrics (métricas)

Tabela 5.2: Kalibro - Fator ergonômico, fatores geográficos, específicos e sociológicos do projeto

(a) Fator ergonômico, fatores geográficos e específicos

Fator ergonômico	
Comunicação com o cliente (meios utilizados)	E-mail e pessoalmente
Comunicação com o cliente (frequência)	Alta
Fatores geográficos	
Localização do time	Time Local
Clientes (quantidade)	2
Clientes (localização)	Próprio ambiente
Fatores específicos	
Domínio	Apoio ao desenvolvimento de software
Duração do projeto e iterações	6 meses e 6 iterações
Total de histórias ou itens de *Backlog* entregues	40

(b) Fatores sociológicos

Fatores sociológicos	
Tamanho da equipe (desenvolvedores)	2
Tamanho da equipe (testadores)	2
Tamanho da equipe (*coach*)	1
Nível de educação da equipe[1]	Fase 1: 1 grad., 5 msc., 2 dr. Fase 2: 1 grad. 2 msc. 1 dr. Fase 3: 1 ba. 1 dr.
Nível de experiência da equipe	-
Conhecimento do domínio pela equipe	Alto
Conhecimento técnico da equipe	Médio

[1] Nível de Educação - grad: graduando, ba: bacharel, msc.: mestrando, dr.: doutorando.

Tabela 5.3: Kalibro - Práticas ágeis, práticas de teste do projeto e aderência a atividade de testes

Fatores específicos	
Práticas ágeis utilizadas (principais)	Integração contínua, área de trabalho informativa, cliente presente, reuniões em pé, planejamento da iteração, código compartilhado, programação em pares, refatoração
Práticas ágeis de teste utilizadas (principais)	Testes de unidade, testes de GUI / usabilidade, métricas de código e testes
Aderência à prática de teste de software	
Testes de unidade automatizados	3
Testes de aceitação com o cliente (*Business Testing*)	1
Desenvolvimento dirigido a testes (TDD)	2

Tabela 5.4: Kalibro - Maturidade da atividade de teste de software

Maturidade da atividade de teste de software	
Testes de unidade e TDD	- Após pensar no *design* e escrever um pouco de código, é escrito um teste automatizado. - O TDD é utilizado, contribuindo para o *design* e qualidade do código. - Preocupa-se com a utilização de padrões para criação de testes que incluem preocupação com a testabilidade do código.
Testes de aceitação e testes de sistema	- Os clientes ou desenvolvedores rodam testes de aceitação para validar o software produzido.
Aspectos de automatização dos testes	- Utilizam-se ferramentas para facilitar o desenvolvimento e execução dos testes. - Utilizam-se testes manuais.
Processo de testes e melhoria contínua	- É estabelecida uma organização das atividades de teste. - Utilização de métricas que controlam e monitoram os testes. Essas métricas são expostas na área Informativa e discutidas em *stand-up meetings*, retrospectivas e reuniões de planejamento. - Preocupação com a melhoria do código de testes e com a otimização do processo.
Importância, benefícios e dificuldades da atividade de teste de software	*"Segurança para refatoração e integração contínua"*
Melhoria da condução da atividade de Teste de Software	*"TDD e planejamento dos testes desde o início da iteração"*

5.2.2 Archimedes

O Archimedes é um sistema CAD (*Computer Aided Design*) livre e gratuito para o desenvolvimento de sistemas de desenho assistidos por computador. Ele foi inicialmente desenvolvido por estudantes de ciência da computação em colaboração com arquitetos profissionais. O projeto foi primeiramente concebido em junho de 2005, mas o desenvolvimento atual iniciou-se em março de 2006. O objetivo do projeto é procurar estabelecer uma base de desenvolvimento confiável e simples com o objetivo de fornecer as ferramentas básicas para elaborar um projeto técnico arquitetural (Figura 5.3). O sistema foi desenvolvido sobre a plataforma da IDE Eclipse, baseando-se em *plug-ins*. Isso significa que o sistema é facilmente extensível, permitindo que a partir de um pequeno núcleo sejam selecionadas as funcionalidades que o usuário considerar interessante e ignore as outras. Além disso, a partir das ferramentas básicas de projeto, deve ser possível incrementar o programa para obter funcionalidades mais complexas. Com uma interface semelhante à da ferramenta AutoCAD (atual líder de mercado), o Archimedes funciona em Windows, Linux e Mac OS X mantendo a aparência nativa do sistema para que os usuários não tenham dificuldades de se adaptar.

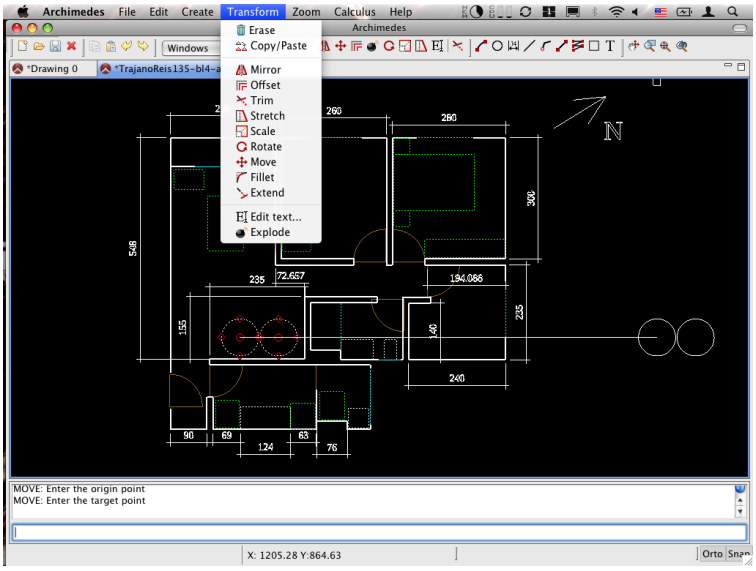


Figura 5.3: Archimedes - Ferramenta CAD (*Computer Aided Design*)

Na Tabela 5.5 são apresentadas informações gerais do projeto Archimedes e informações sobre fatores tecnológicos utilizados no projeto. O fator ergonômico, que trata a respeito da comunicação com o cliente, fatores geográficos (da equipe e dos clientes), fatores específicos e fatores sociológicos são apresentados na Tabela 5.6. Outros fatores específicos referentes ao nível de educação, experiência e conhecimento da equipe, às práticas ágeis e práticas de teste utilizadas no projeto são apresentados na Tabela 5.7. Também foram coletadas informações sobre o nível de aderência e maturidade da atividade de teste (Tabela 5.8).

Tabela 5.5: Archimedes - Informações gerais e fatores tecnológicos do projeto

Informações gerais do projeto	
Endereço Web do projeto	`http://www.archimedes.org.br`
Endereço do respositório de código	`http://svn.archimedes.org.br/public`
Licença do software	Software livre
Classificação do software	Software para usuário final
Fatores tecnológicos	
Método de desenvolvimento	Programação Extrema (XP)
Linguagens	Java (J2SE)
Tecnologias e/ou *frameworks* utilizados	Eclipse RCP
Ferramentas (ambientes de desenvolvimento)	Eclipse
Ferramentas (gerenciamento de projeto)	Xplanner
FERRAMENTAS de teste de software	Ferramentas de Teste de Unidade (Família XUnit ou similares), Cobertura de Testes
Ferramentas/bibliotecas utilizadas (teste de software)	JUnit, JUnit plug-in Coverage
Outras Ferramentas	-

Tabela 5.6: Archimedes - Fator ergonômico, fatores geográficos, específicos e sociológicos do projeto

(a) Fator ergonômico, fatores geográficos e específicos

Fator ergonômico	
Comunicação com o cliente (meios utilizados)	Pessoalmente
Comunicação com o cliente (frequência)	Raramente
Fatores geográficos	
Localização do time	Time Local
Clientes (quantidade)	2
Clientes (localização)	Locais diferentes
Fatores específicos	
Domínio	Software de desenho técnico
Duração do projeto e iterações	4 anos, 26 iterações e 30 versões
Total de histórias ou Itens de *Backlog* entregues	80

(b) Fatores sociológicos

Fatores sociológicos	
Tamanho da equipe (desenvolvedores)	>5
Tamanho da equipe (testadores)	0
Tamanho da equipe (*coach*)	3
Tamanho da equipe (observações)	Todos os desenvolvedores cumpriram todos os papéis e foram 3 equipes diferentes trabalhando no primeiro semestre de cada ano. Cada equipe tinha em torno de 6 a 8 desenvolvedores.

Tabela 5.7: Archimedes - Fatores específicos, práticas ágeis e práticas de teste do projeto

Fatores específicos	
Nível de educação da equipe	Superior - Todos os desenvolvedores eram aluno do curso de ciências da computação (alguns poucos alunos de mestrado).
Nível de experiência da equipe	A maioria tinha experiência de, no máximo, 3 anos de faculdade. Poucos tinham até 2 anos de experiência profissional.
Conhecimento do domínio pela equipe	Médio
Conhecimento técnico da equipe	Médio
Práticas ágeis utilizadas (principais)	Área de trabalho informativa, reuniões em pé, planejamento da iteração, código compartilhado, programação em pares, Refatoração.
Práticas ágeis de teste utilizadas (principais)	Testes de unidade, desenvolvimento dirigido a testes (TDD), métricas de código e testes, testes de integração.

Tabela 5.8: Archimedes - Aderência e Maturidade da atividade de Teste de Software

Aderência à prática de teste de software	
Testes de unidade automatizados	3
Testes de aceitação com o cliente (*Business Testing*)	0
Desenvolvimento dirigido a testes (TDD)	2
Maturidade da atividade de teste de software	
Testes de unidade e TDD	O TDD é utilizado, contribuindo para o *design* e qualidade do código.
Testes de aceitação e Testes de sistema	-
Aspectos de automatização dos testes	- Utilizam-se ferramentas para facilitar o desenvolvimento e execução dos testes. - Utilizam-se ferramentas para gerenciar os testes. - Utilizam-se ferramentas para apoiar a melhoria dos testes. - Utilizam-se testes manuais.
Processo de testes e melhoria contínua	- A atividade de teste é planejada e gerenciada desde o início do projeto. - Utilização de métricas que controlam e monitoram os testes. Essas métricas são expostas na área Informativa e discutidas em *stand-up meetings*, retrospectivas e reuniões de planejamento. - Preocupação com a melhoria do código de testes e com a otimização do processo.
Importância, benefícios e dificuldades da atividade de teste de software	*"Os testes são excelentes indicadores da qualidade do código gerado assim como do design geral. Existe um problema de performance para executar os testes de integração já que levantar a plataforma toda do eclipse é custoso. É sempre exigido que se escrevam testes, mas nem sempre a equipe tem cuidado para não esquecê-los."*
Melhoria da condução da atividade de Teste de Software	*"O projeto como um todo tem um legado de testes mal escritos que precisam melhorar."*

5.3 Análise dos Resultados

Essa seção analisa os resultados obtidos a partir coleta de informações dos projetos Kalibro e Archimedes. Primeiramente serão discutidos os resultados do questionário apresentado no Apêndice A, dando ênfase principalmente à condução da atividade de teste de software nesses dois projetos. Além disso, serão analisadas as métricas de acompanhamento de teste de software coletadas pela ferramenta ATMM e pelos *scripts* da ferramenta *JaBUTi*.

5.3.1 Resultados do questionário de informações gerais do projeto

Os dois projetos tiveram seu início em ambiente acadêmico e possuem licença de software livre. Em relação aos fatores tecnológicos os dois projetos utilizaram o método de desenvolvimento XP, a linguagem Java (J2SE), o ambiente de desenvolvimento Eclipse (`http://www,eclipse.org`) e a ferramenta para gerenciamento de projeto XPlanner (`http://www.xplanner.org`). O projeto Kalibro utiliza a plataforma Noosfero (`http://noosfero.org`) para apoiar o seu gerenciamento. Para auxiliar na condução da atividade de testes, os dois projetos utilizam a ferramenta JUnit para testes de unidade (`http://www.junit.org`) e ferramentas para cobertura de testes. O projeto Kalibro também utiliza o *framework* Fest (`http://code.google.com/p/fest`) para testes da interface gráfica, além da ferramenta Analizo que faz parte da Kalibro utilizar testes de aceitação por meio da ferramenta Cucumber (`http://cukes.info`).

Os dois projetos analisados possuem dois clientes, sendo que no projeto Kalibro eles são localizados no próprio ambiente, e no projeto Archimedes em locais diferentes, por se tratar de um projeto na área de arquitetura. A comunicação com o cliente no projeto Kalibro é alta, sendo utilizados o e-mail e conversas pessoalmente. No projeto Archimedes as conversas são feitas apenas pessoalmente e ocorrem com menos frequência (raramente).

O projeto Kalibro e Archimedes têm o seu time localizado localmente, sendo que o projeto Kalibro conta atualmente com dois desenvolvedores que também assumem o papel de testadores e um *coach*. No início do projeto Kalibro, a equipe era formada por 8 integrantes (sendo 1 aluno de graduação, 5 de mestrado e 2 de doutorado). No projeto Archimedes os desenvolvedores cumprem todos os papéis, sendo que já houveram 3 equipes diferentes, que variaram entre 6 a 8 desenvolvedores e 3 *coaches*. Em relação ao tempo de projeto, o Archimedes já vêm sendo desenvolvido a 4 anos e o Kalibro a apenas 6 meses. O projeto Kalibro em suas três fases contou com alunos de graduação, mestrado e doutorado, já o projeto Archimedes contou em sua maioria, com alunos de graduação. As duas equipes possuem um conhecimento técnico médio e o conhecimento do domínio varia de médio no projeto Archimedes, a alto no projeto Kalibro.

Algumas práticas ágeis são utilizadas pelos dois projetos, como a área de trabalho informativa, reuniões em pé, planejamento da iteração, código compartilhado, programação em

pares e refatoração. O projeto Kalibro também utiliza a prática de integração contínua e cliente presente. As práticas de teste utilizada pelos dois projetos são os testes de unidade e métricas de código e testes, sendo que no projeto Kalibro são conduzidos testes com a interface gráfica e no projeto Archimedes utiliza-se TDD e testes de integração.

Foram medidos o nível de aderência à prática de teste de software e também a maturidade do projeto em relação à atividade de teste. O nível de aderência (de 0 à 4) foi coletado por valores objetivos do questionário. O nível de maturidade de cada prática de testes (de 0 à 5) também foi coletado conforme as opções selecionadas no questionário. Na categoria de "Testes de unidade e TDD" o nível vai de 0 (quando não há nenhum teste formal) até o nível 5 (quando o TDD é aplicado e ainda há uma preocupação com a utilização de padrões para os testes). O nível do restante das categorias foi medido conforme a porcentagem de opções selecionadas. Esses dados são apresentados na Tabela 5.9 e na Tabela 5.10.

Tabela 5.9: Kalibro - Níveis de aderência e maturidade da atividade de testes

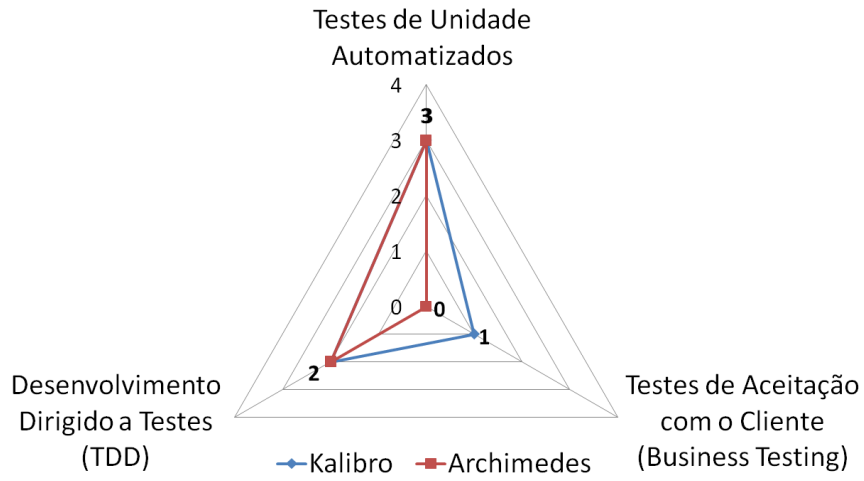
Aderência à atividade de teste (0 - 4)		
Testes de Unidade Automatizados	3	
Testes de Aceitação com o Cliente (*Business Testing*)	1	
Desenvolvimento Dirigido a Testes (TDD)	2	
Maturidade da atividade de teste (0 - 5)		
Testes de Unidade e TDD	5	Máx (TDD e Padrões)
Testes de Aceitação e Testes de Sistema	2,5	1 de 2 (50%)
Aspectos de Automatização dos Testes	2	2 de 5 (40%)
Processo de Testes e Melhoria Contínua	3,75	3 de 4 (75%)
Total de Práticas de Teste	2,1	3 de 7 (42%)

Tabela 5.10: Archimedes - Níveis de aderência e maturidade da atividade de testes

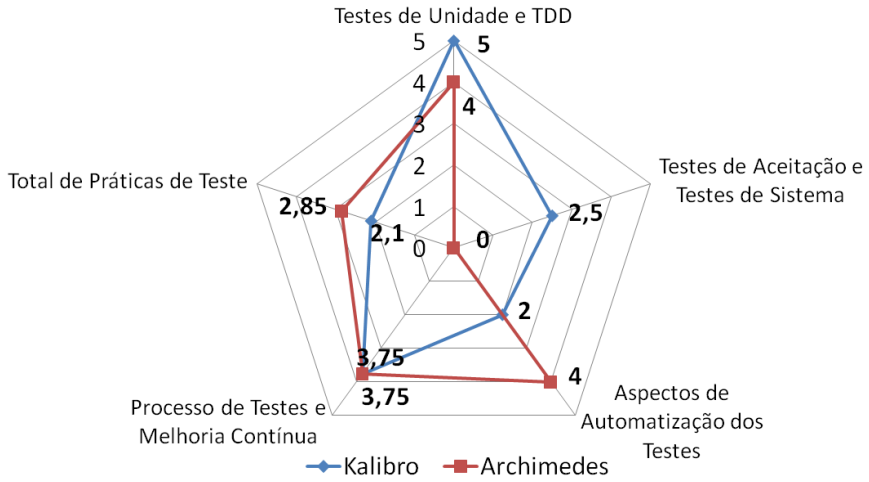
Aderência à atividade de teste (0 - 4)		
Testes de Unidade Automatizados	3	
Testes de Aceitação com o Cliente (*Business Testing*)	0	
Desenvolvimento Dirigido a Testes (TDD)	2	
Maturidade da atividade de teste (0 - 5)		
Testes de Unidade e TDD	4	TDD
Testes de Aceitação e Testes de Sistema	0	0 de 2 (0%)
Aspectos de Automatização dos Testes	4	4 de 5 (80%)
Processo de Testes e Melhoria Contínua	3,75	3 de 4 (75%)
Total de Práticas de Teste	2,85	4 de 7 (42%)

Os níveis de aderência e maturidade também foram descritos por meio dois gráficos do tipo radar, ilustrados na Figura 5.4. Esse gráfico fornece uma simples representação de múltiplos indicadores de avaliação de forma intuitiva, até mesmo para não especialistas. O gráfico resultante mostra áreas de força ou fraquezas relativas, bem como descreve a performance geral do conjunto de indicadores avaliados (Mosley e Mayer, 1998). As categorias com valores mais altos, marcados mais próximos da extremidade dos gráficos possuem um valor mais alto, descrevendo um maior nível de aderência ou de maturidade.

Em termos da aderência à atividade de teste, foi constatado que os dois projetos possuem um equilíbrio em relação aos testes de unidade automatizados e ao TDD. No projeto Archimedes não são realizados testes de aceitação automatizados. Por outro lado, no projeto Kalibro foram considerados testes de aceitação, testes com o *framework* Fest e testes com a ferramenta Cucumber no projeto Analizo (`http://softwarelivre.org/mezuro/analizo`) que é utilizado para coletar as métricas da ferramenta Kalibro. A maturidade da atividade de teste nos dois projetos também apresentou um certo equilíbrio, sendo que os testes de unidade já possuem uma grande maturidade nas duas equipes e também há uma grande preocupação com o processo de testes e melhoria contínua. Nenhum dos projetos aplicam testes de sistema, e apenas o projeto Kalibro aplica testes de aceitação. Em relação a aspectos de automatização, o projeto Archimedes apresenta uma maior maturidade, apesar de os resultados do questionário afirmarem que o projeto utiliza ferramentas para gerenciar os testes, não são citadas quais ferramentas são utilizadas para esse fim.



(a) Aderência a atividade de teste (0 à 4)



(b) Maturidade da atividade de teste (0 à 5)

Figura 5.4: Gráfico de valores - Aderência e Maturidade da atividade de teste no projeto Kalibro e Archimedes

5.3.2 Análise das métricas de acompanhamento de teste de software

Nessa seção será apresentada uma análise dos resultados das métricas de acompanhamento de testes, que foram coletados pelos *scripts* da ferramenta *JaBUTi* (cobertura de código), pela ferramenta ATMM (métricas do código-fonte e acompanhamento de testes) e também pela ferramenta Kalibro (métricas de qualidade do código produzido). As métricas foram coletadas a partir do código-fonte e os testes armazenados nos repositórios de código dos projetos, sendo que no projeto Kalibro foram analisadas quatro iterações (4 meses de projeto) e no projeto Archimedes foram analisadas 3 iterações (6 meses de projeto).

Por limitações nos *scripts* da *JaBUTi* e na ferramenta ATMM não foi possível avaliar as iterações mais recentes da ferramenta Archimedes. A estrutura de código e testes das versões mais recentes do projeto Archimedes foi refeita, o que impossibilitou a avaliação dessas versões.

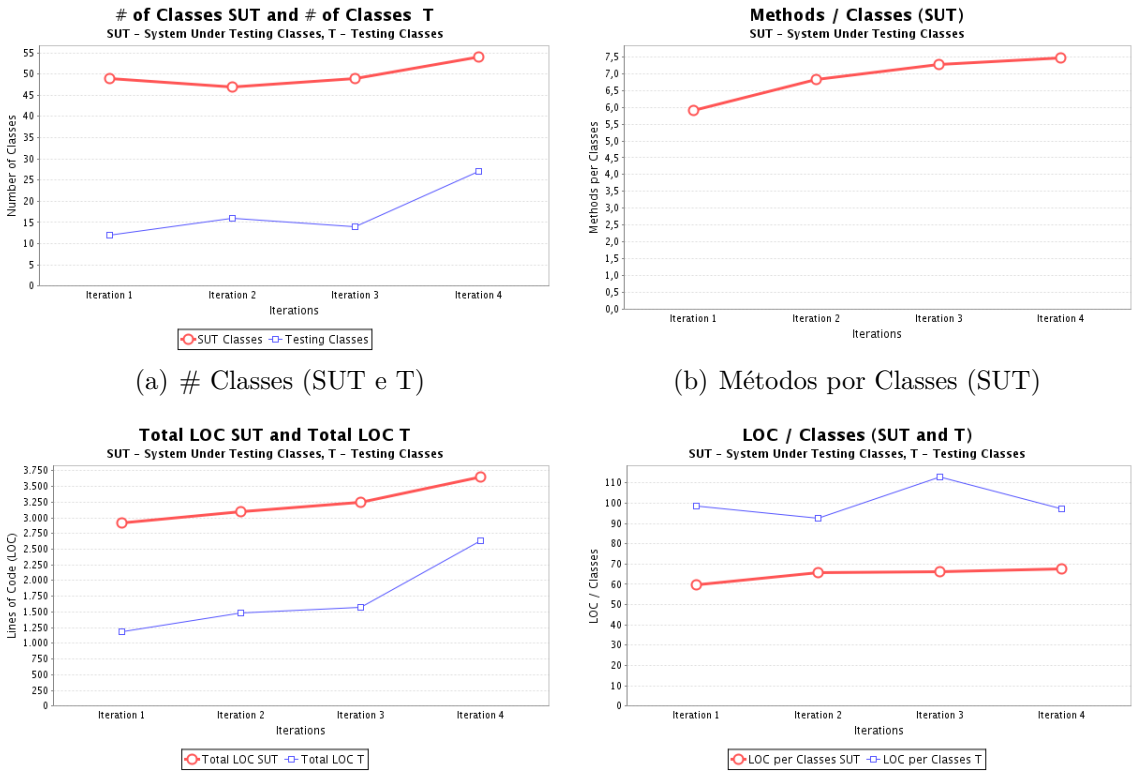
5.3.2.1 Evolução da quantidade de código e testes produzidos

A evolução da quantidade de classes, e código-fonte e código de testes produzidos no projeto Kalibro é descrita por meio da Tabela 5.11 e das Figuras 5.5 e 5.6. Constata-se que não houveram grandes alterações da iteração 1 à iteração 3, fazendo com que o fator de teste se mantivesse constante nessas iterações. No entanto, na iteração 4 foi produzido uma grande quantidade de código de testes (aumento de 66,26%), aumentando o fator de teste para 0,71 (Figura 5.6).

Também foram analisadas métricas relacionadas a quantidade de código e de testes produzidos no projeto Archimedes. Comparando a iteração 4 do projeto Kalibro com a iteração 3 do projeto Archimedes (Tabela 5.12 e Figura 5.8), é possível constatar que o projeto Archimedes possui uma grande diferença em relação à quantidade de código-fonte e código de teste. Apesar de terem sido analisadas três iterações do projeto Archimedes, não foi possível constatar evoluções nas métricas de acompanhamento de testes. A diferença de tamanho do projeto Kalibro e projeto Archimedes pode ser justificada tanto pelo tempo de existência dos dois projetos, como também pelos diferentes domínios de aplicação. A métrica de fator de teste do projeto Archimedes (ilustrada na Figura 5.7) é 33% mais baixa em relação ao projeto Kalibro, no entanto, comparativamente com outros projetos ágeis apresentados em Sato (2007) o fator de teste pode ser considerado bom.

Tabela 5.11: Kalibro - Métricas de Acompanhamento de Testes (Quantidade de Código e Testes)

Métrica	Iteração 1	Iteração 2	Iteração 3	Iteração 4
# de Classes SUT	49	47	49	54
# de Classes T	12	16	14	27
Métodos / Classes (SUT)	5.89	6.82	7.26	7.46
LOC SUT	2924	3098	3249	3649
LOC T	1184	1484	1580	2627
Fator de Teste (LOC SUT / LOC T)	0.40	0.47	0.48	0.71
LOC / Classes (SUT)	59.67	65.91	66.30	67.57
LOC / Classes (T)	98.66	92.75	112.85	97.29



(a) # Classes (SUT e T)

(b) Métodos por Classes (SUT)

(c) Total LOC (SUT e T)

(d) LOC / Classes

Figura 5.5: Kalibro - Quantidade de linhas de código (LOC) e LOC / Classes

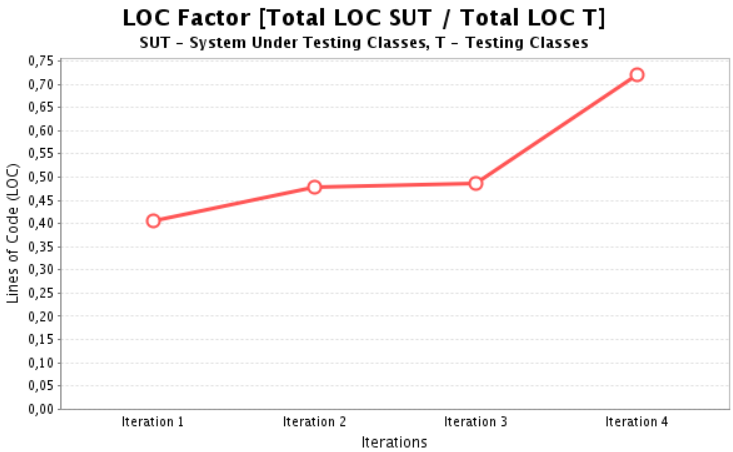


Figura 5.6: Kalibro - Fator de Teste (LOC SUT / LOC T)

Tabela 5.12: Archimedes - Métricas de Acompanhamento de Testes (Quantidade de Código e Testes)

Métrica	Iteração 1	Iteração 2	Iteração 3
# de Classes SUT	172	172	188
# de Classes T	70	70	70
Métodos / Classes (SUT)	8.45	8.48	7.87
LOC SUT	18525	18839	18611
LOC T	9958	9978	9987
Fator de Teste (LOC SUT / LOC T)	0.53	0.52	0.53
LOC / Classes (SUT)	107.70	109.52	98.99
LOC / Classes (T)	142.25	142.54	142.67

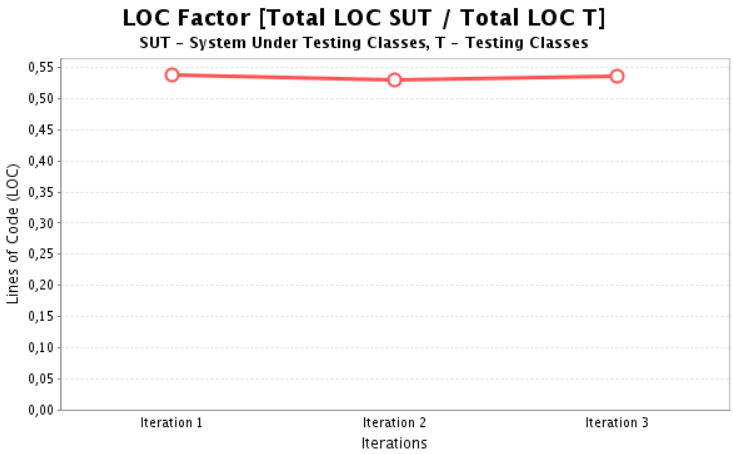
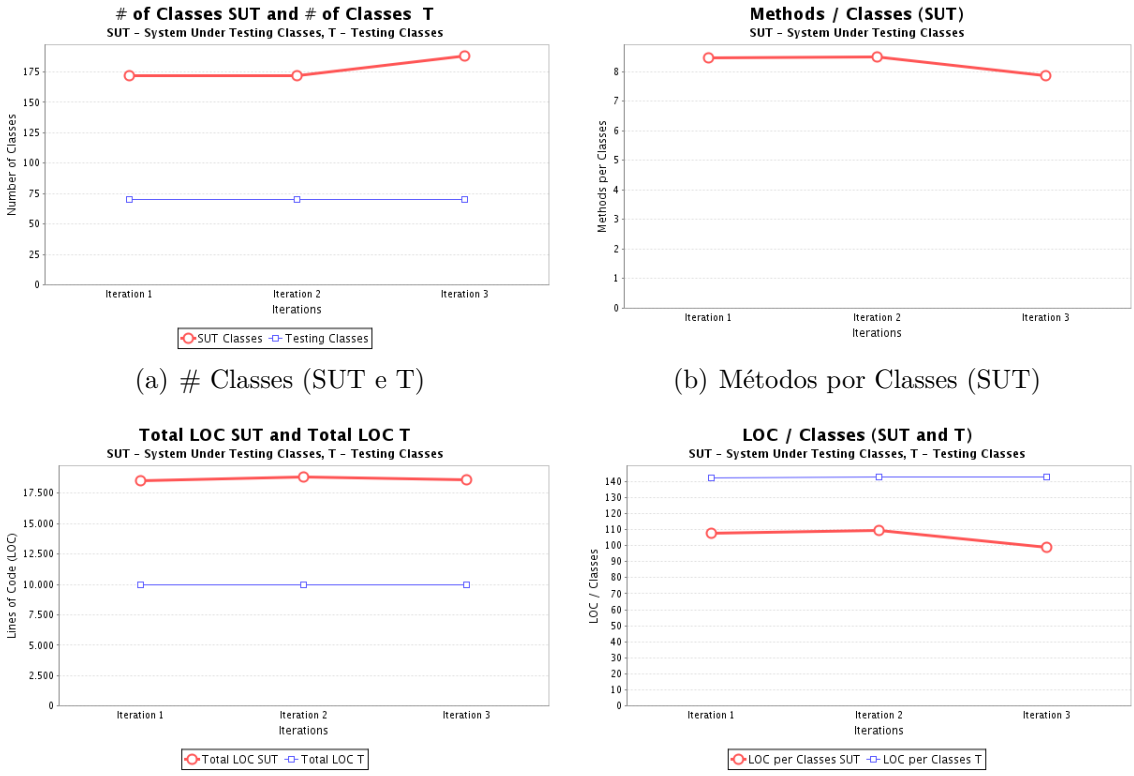


Figura 5.7: Archimedes - Fator de Teste (LOC SUT / LOC T)



(a) # Classes (SUT e T)

(b) Métodos por Classes (SUT)

(c) Total LOC (SUT e T)

(d) LOC / Classes

Figura 5.8: Archimedes - Quantidade de linhas de código (LOC) e LOC / Classes

Adicionalmente também foi avaliada a qualidade do código produzido, utilizando algumas das métricas fornecidas pela ferramenta Kalibro. A própria ferramenta sugere os intervalos para que o resultados das métricas sejam considerados bons, ruins ou regulares Oliveira Filho (2009)[3]:

- **AMLOC (*Average Lines per Method* - Número médio de linhas por método):** quanto maior a quantidade de linhas por método, fica mais difícil entender e manter esses métodos. A equipe deve optar por produzir operações pequenas e de fácil entendimento do que operações grandes e complexas. Os intervalos sugeridos são: até 10 (bom); entre 10 e 13 (regular); de 13 em diante (ruim).

- **MMLOC (*Max Method LOC* - Número máximo de linhas em um método):** semelhante a métrica AMLOC, no entanto ela mede o número máximo de linhas em um método. Os intervalos sugeridos são: até 13 (bom), entre 13 e 19.5 (regular) e 19.5 em diante (ruim).

- **LCOM4 (*Lack of Cohesion in Methods* - Ausência de coesão em métodos):** Dois métodos estão relacionados se ambos acessam pelo menos um mesmo atributo da classe ou se um método chama ou é chamado por outro . LCOM4 é a quantidade de

[3]Apesar de ser utilizada pela ferramenta Kalibro, a configuração dos intervalos está sendo avaliada e ainda não possui validação científica.

partições de formadas após separar os métodos em conjuntos de métodos relacionados. A coesão entre os métodos de uma classe é uma propriedade desejável, portanto o valor ideal dessa métrica é 1. Se uma classe tem diferentes conjuntos de métodos não relacionados entre si, é um indício de que a classe deveria ser quebrada em classes menores e mais coesas. Os intervalos sugeridos são: até 2 (bom); entre 2 e 5 (regular); de 5 em diante (ruim).

- **ACC (*Afferent Connections per Class* - Conexões aferentes de uma classe):** Mede a conectividade de uma classe (se uma classe acessa um método ou atributo de outra classe, a cada ligação o valor dessa métrica é incrementado em 1). Se o valor dessa métrica for grande, uma mudança na classe tem potencialmente mais efeitos colaterais, tornando mais difícil a manutenção. Os intervalos sugeridos são: até 2 (bom); entre 2 e 20 (regular); de 20 em diante (ruim).

- **CBO (*Coupling Between Objects* — Ligações entre objetos):** é a recíproca da métrica ACC. Mede quantas classes são utilizadas pela classe analisada. Os intervalos sugeridos são: até 2 (bom); entre 2 e 5 (regular); de 5 em diante (ruim).

Os valores médios das métricas AMLOC e MMLOC mostram que o projeto Kalibro possui métodos pequenos que facilitam o seu entendimento, no entanto ainda apresenta alguns métodos com uma quantidade de linhas maior do que a sugerida. Segundo a prática da refatoração, sempre que possível métodos com uma grande quantidade de linhas devem ser refatorados (Fowler, 1999). Em relação as métricas LCOM4, ACC e CBO que medem o grau de coesão e acoplamento, sugerem que a ferramenta Kalibro tem classes com um bom grau de coesão entre seus métodos segundo a métrica LCOM4 e um baixo acoplamento entre suas classes segundo as métricas ACC e CBO .

Tabela 5.13: Kalibro - Qualidade do código-fonte (AMLOC, MMLOC, LCOM4, ACC, COF)

Métrica	Iteração 1	Iteração 2	Iteração 3	Iteração 4
AMLOC	11.55 (bom)	7.69 (bom)	8.06 (bom)	7.20 (bom)
MMLOC	157.0 (ruim)	86.0 (ruim)	86.0 (ruim)	68.0 (ruim)
LCOM4	2.65 (regular)	1.48 (bom)	1.56 (bom)	1.72 (bom)
ACC	3.16 (regular)	0.98 (bom)	1.10 (bom)	1.25 (bom)
CBO	2,18 (regular)	0.92 (bom)	1.04 (bom)	1.28 (bom)

No projeto Archimedes, a métrica AMLOC que mede a média de linhas de código por método é considerada regular e o número máximo de linhas em um método (MMLOC) é ruim. Um método com 356 linhas (encontrado no projeto Archimedes) certamente deve ser refatorado pela equipe de desenvolvimento. Em relação as métricas LCOM4, ACC e CBO que medem o grau de coesão e acoplamento, sugerem que a ferramenta Archimedes tem classes com um grau regular de coesão entre seus métodos segundo a métrica LCOM4 e um grau de acoplamento regular entre suas classes segundo as métricas ACC e CBO .

Tabela 5.14: Archimedes - Qualidade do código-fonte (AMLOC, MMLOC, LCOM4, ACC, COF)

Métrica	Iteração 1	Iteração 2	Iteração 3
AMLOC	11.53 (regular)	11.52 (regular)	11.55 (regular)
MMLOC	356 (ruim)	356 (ruim)	157 (ruim)
LCOM4	2.82 (regular)	2.79 (regular)	2.65 (regular)
ACC	3.30 (regular)	3.33 (regular)	3.16 (regular)
CBO	2.25 (regular)	2.27 (regular)	2.18 (regular)

5.3.2.2 Evolução do teste de unidade (Casos de Teste, Assertivas e Tempo de Execução)

No projeto Kalibro os testes de unidade foram criados utilizando a ferramenta JUnit e o *framework* Fest (para testes da GUI). Por meio da Tabela 5.15 e da Figura 5.9 é possível constatar uma evolução constante da quantidade de casos de teste e da quantidade de assertivas, sendo que da iteração 3 para iteração 4 houve um aumento de 70% na quantidade de casos de teste e 41,50% na quantidade de assertivas. A partir da relação entre esses dados de evolução dos casos de teste e assertivas e o aumento do valor do fator de teste, pode-se presumir que houve uma evolução da cobertura de código durante as iterações, principalmente da iteração 3 para iteração 4. O tempo total e o tempo médio de execução dos casos de teste de unidade mantiveram valores aceitáveis, permitindo que sejam executados com frequência, a cada alteração no código-fonte.

Tabela 5.15: Kalibro - Métricas de Acompanhamento de Testes (Testes de Unidade)

Métrica	Iteração 1	Iteração 2	Iteração 3	Iteração 4
Qtde casos de teste	60	87	103	175
Qtde assertivas	231	317	378	535
Casos de teste com Sucesso / Erro / Falha	19 / 40 / 1	85 / 1 / 1	102 / 1 / 0	173 / 1 / 1
Tempo Total	0,2s	19s	22s	287s
Tempo Médio de execução	0,016s	1,187s	1,571s	10,62s

A quantidade de casos de teste e assertivas no projeto Archimedes mantiveram-se praticamente constantes. Além disso, o projeto Archimedes possui uma quantidade de casos de teste e assertivas maior do que o projeto Kalibro (Tabela 5.16 e Figura 5.10). No entanto, comparando a quantidade de linhas de código (LOC SUT) e quantidade de assertivas do projeto Archimedes e o projeto Kalibro em suas últimas iterações, há uma diferença de 410% a mais de código-fonte e apenas 247% a mais de assertivas no projeto Archimedes. Esse fato provavelmente resultará em uma menor cobertura de código do projeto Archimedes em relação ao projeto Kalibro. O tempo total e o tempo médio de execução dos casos de teste de unidade mantiveram valores aceitáveis para que eles sejam executados com frequência, a cada alteração no código-fonte.

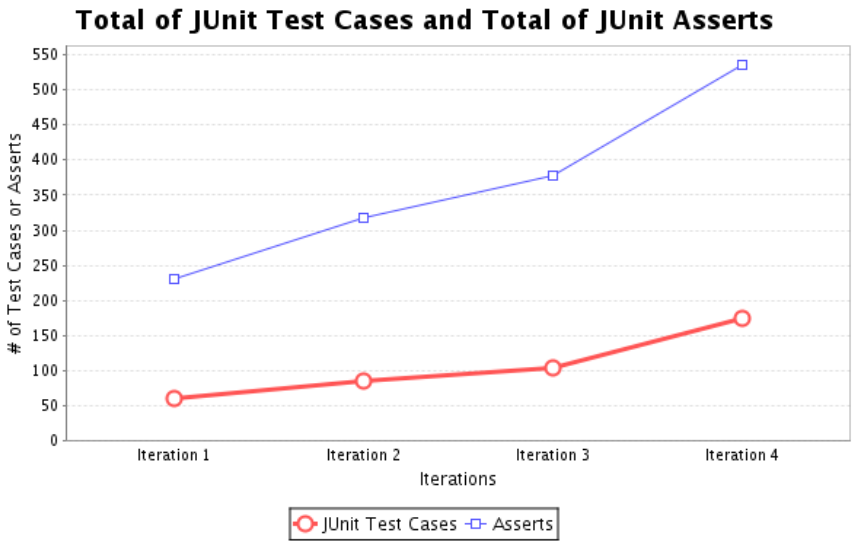


Figura 5.9: Kalibro - Total de Casos de Teste e Assertivas (JUnit)

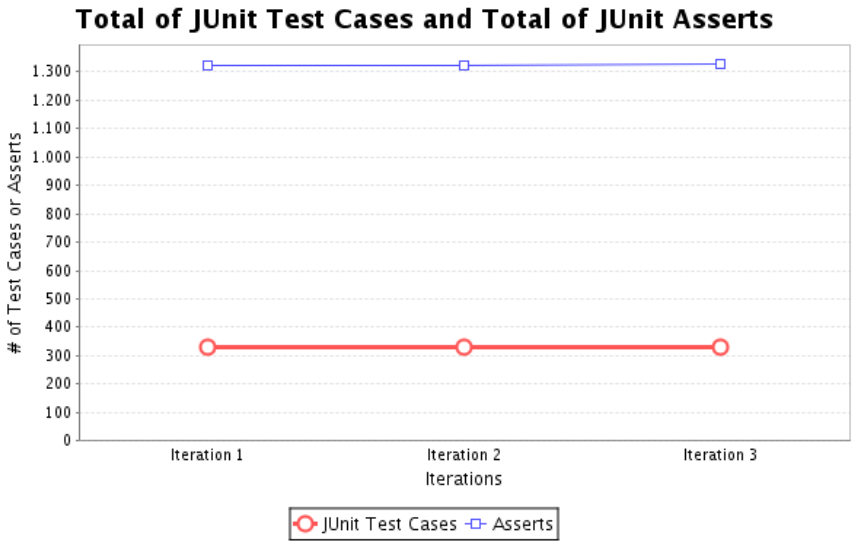


Figura 5.10: Archimedes - Total de Casos de Teste e Assertivas (JUnit)

Tabela 5.16: Archimedes - Métricas de Acompanhamento de Testes (Testes de Unidade)

Métrica	Iteração 1	Iteração 2	Iteração 3
Qtde casos de teste	331	331	332
Qtde assertivas	1320	1322	1326
Casos de teste com Sucesso / Erro / Falha	275/5/51	275/5/51	272/7/53
Tempo Total	1s	5.9s	8s
Tempo Médio de execução	0,014s	0,084s	0,11s

5.3.2.3 Evolução da Cobertura de Código

Uma das métricas de acompanhamento de teste é a análise de cobertura do código. Esse processo consiste em encontrar áreas do programa que não foram exercitados pelo conjunto de casos de teste, criar casos de teste adicionais para aumentar a sua cobertura, determinando uma medida quantitativa de cobertura de cobertura do código, que é indiretamente uma medida de qualidade. A cobertura também pode identificar casos de teste redundantes que não aumentam a cobertura. É responsabilidade da equipe ágil estabelecer quais serão as porcentagens mínimas de cobertura de código, para determinar quando parar a análise de cobertura e também selecionar quais técnicas de teste e ferramentas que serão utilizadas.

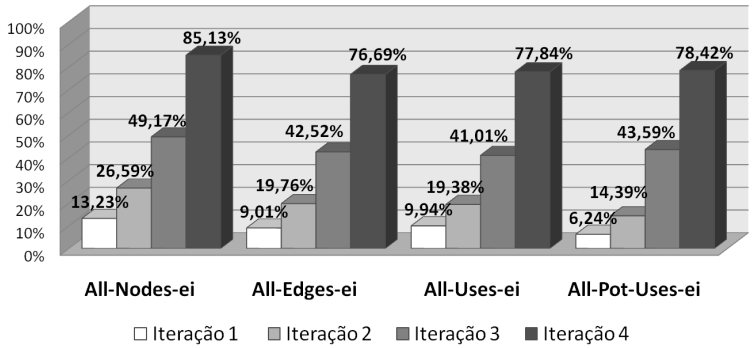
Um dos benefícios do TDD, segundo Beck (2002) é que a equipe consegue atingir 100% de cobertura dos testes. No entanto, alguns tipos de código são inerentemente difíceis de testar utilizando TDD (como por exemplo teste de GUI) (George e Williams, 2004). A equipe deve encontrar meios de testar esse tipo de código, utilizando um *framework*, ferramenta ou até mesmo executando testes manuais (Martin, 2007). Apesar de alguns estudos experimentais demonstrarem que a utilização do TDD fornece códigos próximos do 100% para cobertura de linhas e desvios do código (George e Williams, 2004), Madeyski (2010) afirma que o impacto da prática do TDD em termos de cobertura de código é ainda não conclusivo.

Programadores que utilizam o TDD para escrever teste de unidade, podem ser beneficiados por medidas que indiquem quando o software foi testado de forma eficiente (Madeyski, 2010). Nesse contexto são apresentados na Tabela 5.18 e Figura 5.11 a evolução da cobertura do código do projeto Kalibro utilizando critérios de teste estruturais da ferramenta *JaBUTi*.

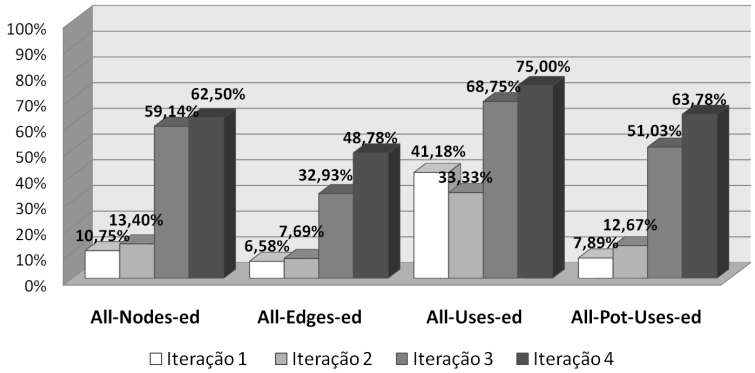
Observa-se que no projeto Kalibro houve uma grande evolução na cobertura do código, sendo que a equipe constantemente se dedica a desenvolver novos casos de teste para melhoria da cobertura. A preocupação com os testes pode ser constatada tanto pela evolução da cobertura dos testes, como nos diversos comentários nas revisões do repositório de código, que informam a adição de mais testes, além dos documentos das versões entregues que mostram a cobertura alcançada nos pacotes de classes do projeto. Outro aspecto interessante do projeto foi a utilização do *framework* Fest para testar as classes da interface gráfica da ferramenta. Esses testes contribuíram para o aumento da cobertura do código de todo o projeto.

Tabela 5.17: Kalibro - Análise de Cobertura dos Testes (utilizando critérios estruturais)

	Iteração 1	**Iteração 2**	**Iteração 2**	**Iteração 4**
All-Nodes-ei	13,23% (185 de 1398)	↑ 26,59% (417 de 1568)	↑ 49,17% (768 de 1562)	↑ 85,13% (1631 de 1916)
All-Nodes-ed	10,75% (10 de 93)	↑ 13,40% (13 de 97)	↑ 59,14% (55 de 93)	↑ 62,50% (60 de 90)
All-Edges-ei	9,01% (101 de 1121)	↑ 19,76% (247 de 1250)	↑ 42,52% (503 de 1183)	↑ 76,69% (1099 de 1433)
All-Edges-ed	6.58% (10 de 152)	↑ 7,69% (13 de 169)	↑ 32,93% (55 de 167)	↑ 48,78% (60 de 123)
All-Uses-ei	9.94% (210 de 2113)	↑ 19,38% (463 de 2389)	↑ 41,01% (922 de 2248)	↑ 77,84% (2132 de 2739)
All-Uses-ed	41.18% (7 de 17)	↓ 33,33% (7 de 21)	↑ 68,75% (11 de 16)	↑ 75,00% (9 de 12)
All-Pot-Uses-ei	6.24% (318 de 5098)	↑ 14,39% (804 de 5586)	↑ 43,59% (2106 de 4831)	↑ 78,42% (4809 de 6132)
All-Pot-Uses-ed	7.89% (24 de 304)	↑ 12,67% (38 de 300)	↑ 51,03% (148 de 290)	↑ 63,78% (162 de 254)



(a) Critérios estruturais EI - *Exception Independent*



(b) Critérios estruturais ED - *Exception Dependent*

Figura 5.11: Kalibro - Cobertura de Testes utilizando Critérios de Fluxo de Controle e Fluxo de Dados

No projeto Archimedes a cobertura de código, como previsto anteriormente na análise da quantidade de casos de teste e assertivas em relação ao tamanho do projeto, fez com que a cobertura da Archimedes fosse menor que a do projeto Kalibro. Também é possível constatar

que a cobertura se manteve praticamente constante durante as três iterações devido à pouca variação no código-fonte e no código de teste.

Tabela 5.18: Archimedes - Análise de Cobertura dos Testes (utilizando critérios estruturais)

	Iteração 1	**Iteração 2**	**Iteração 3**
All-Nodes-ei	62,74% (4685 de 7467)	62,93% (4740 de 7532)	61.86% (4640 de 7501)
All-Nodes-ed	20,82% (209 de 1004)	20,90% (209 de 1000)	20,64% (207 de 1003)
All-Edges-ei	49,15% (3753 de 7636)	49,33% (3799 de 7701)	48,36% (3710 de 7641)
All-Edges-ed	13,52% (203 de 1501)	13,51% (203 de 1503)	13,53% (201 de 1486)
All-Uses-ei	52,51% (7251 de 13834)	52,66% (7347 de 13951)	51,45% (7177 de 13950)
All-Uses-ed	32,31% (63 de 195)	34,81% (63 de 181)	31,75% (60 de 189)
All-Pot-Uses-ei	47,45% (20407 de 43010)	47,86% (20736 de 43327)	46,62% (20294 de 43528)
All-Pot-Uses-ed	12,66% (767 de 6058)	12,78% (767 de 6003)	12,45% (751 de 6034)

5.4 Discussão e Conclusões

Esse capítulo apresentou estudos de casos que tiveram como objetivo principal avaliar dois projetos ágeis acadêmicos utilizando as métricas de acompanhamento de teste sugeridos por esse trabalho. Os projetos foram primeiramente caracterizados por meio de um questionário baseado em um *framework* de avaliação para projetos desenvolvidos utilizando o método XP - *Extreme Programming Framework* (XP-EF), que coletaram informações gerais do projeto, as práticas ágeis utilizadas e também a aderência e maturidade da atividade de teste de software desses projetos. O código-fonte e casos de teste de unidade desses projetos também foram avaliados utilizando-se as métricas de acompanhamento de teste sugeridas por este trabalho, além de algumas métricas de software relacionadas a qualidade dos métodos produzidos e também a métricas de coesão e acoplamento.

Os principais resultados obtidos a partir da avaliação das informações coletadas pelo questionário e pelas demais métricas avaliadas no estudo de caso foram:

- **Diferenças gerais entre os projetos analisados:** os dois projetos são desenvolvidos no contexto acadêmico, utilizando licenças de software livre e também o método XP. Os projetos são de domínios diferentes, sendo que o projeto Kalibro está no domínio de apoio ao desenvolvimento de software e o projeto Archimedes é um software de

desenho técnico. Os dois projetos são bastante distintos sob diversos aspectos como o tamanho dos dois projetos em termos de linhas de código e código de teste de unidade, tempo em que vêm sendo desenvolvido, a quantidade de pessoas envolvidas e também a qualificação dos seus desenvolvedores.

- **Aderência e maturidade da atividade de testes:** os dois projetos possuem um equilíbrio em relação a aderência da atividade de testes. Apesar de possuírem um nível alto em relação aos testes de unidade automatizados, os testes de aceitação e a estratégia TDD ainda não são utilizados totalmente. A partir das questões relacionadas a maturidade da atividade de teste, foi constatado que os testes de unidade já possuem uma grande maturidade nas duas equipes e também há uma preocupação com o processo de testes e melhoria contínua. Nenhum dos projetos afirmaram a preocupação com testes de sistema e apenas o projeto Kalibro aplica testes de aceitação. Segundo o questionário, os dois projetos valorizam a atividade de teste como forma de apoiar a qualidade do código gerado, colaborando com as práticas de refatoração e integração contínua.

- **Evolução do código-fonte e testes:** analisando a evolução do código e dos testes no projeto Kalibro, foi possível verificar iterações que deram uma maior ênfase a quantidade de testes criados. Essas iterações trouxeram um aumento visível da cobertura de código. O aumento na quantidade e qualidade dos testes no projeto Kalibro fornece uma maior confiabilidade para que a equipe efetue mudanças e garanta que o software será entregue com uma quantidade menor de defeitos. No projeto Archimedes não foi possível constatar a evolução no código que possivelmente foi provocado pelo impedimento de avaliar versões mais recentes do projeto que foi totalmente reestruturado.

- **Cobertura de código-fonte:** ao analisar os resultados de cobertura do código segundo critérios estruturais, é possível verificar que a cobertura na ferramenta Archimedes se manteve constante, devido ao código-fonte e os casos de teste quase não terem sido modificados nas três iterações avaliadas. No projeto Kalibro foi possível constatar uma crescente preocupação com a cobertura dos testes, sendo que a equipe dedicou parte do ciclo de desenvolvimento para desenvolver mais testes que aumentem a cobertura e consequentemente a qualidade do conjunto de casos de teste do projeto. Além disso, a utilização de um *framework* para o teste de classes de interface gráfica no projeto Kalibro, contribuiu para o aumento da porcentagem de cobertura.

- **Qualidade do código:** analisando as métricas relacionadas ao tamanho dos métodos, a coesão e acoplamento das classes desenvolvidas, é possível constatar que além da equipe de desenvolvimento se preocupar em gerar códigos com o mínimo de defeitos. Segundo as métricas de software analisadas a ferramenta Kalibro possui métodos pequenos que facilitam o seu entendimento, tem classes com um bom grau de coesão entre

seus métodos e um baixo acoplamento entre suas classes. A ferramenta Archimedes possui métodos com uma média de linhas considerada regular e tem classes com um grau regular de coesão entre seus métodos e um grau de acoplamento regular entre suas classes. Por fim, chega-se a conclusão de que os dois projetos se preocupam também com a qualidade do código que está sendo produzido.

A partir dos resultados obtidos nos estudos de caso do projeto Kalibro e do projeto Archimedes foi possível caracterizar os dois projetos, principalmente em relação à atividade de teste. Foi possível analisar a evolução dos dois projetos durante suas iterações de desenvolvimento, que evidenciou uma preocupação com a melhoria dos casos de teste principalmente na ferramenta Kalibro. A partir dos resultados obtidos no estudo de caso foi possível analisar a evolução do projeto Kalibro durante suas iterações de desenvolvimento, que evidenciou uma preocupação com a melhoria dos casos de teste. Por meio do estudo de caso, pode-se confirmar com suas devidas limitações a hipótese de que a utilização de métricas de teste pode colaborar na detecção de problemas durante a condução dessa atividade, além de monitorar a evolução da qualidade dos casos de teste, que reflete na qualidade do código fonte produzido. Esta hipótese foi confirmada pelo projeto Kalibro que apresentou bons resultados em termos de métricas de acompanhamento de teste de unidade que resultaram em uma alta cobertura de requisitos de teste, com poucos defeitos nos testes de unidade e que possivelmente se refletiu no bom desempenho das métricas ligadas a qualidade do código fonte.

Devido a limitações nos *scripts* da ferramenta *JaBUTi* e da ferramenta ATMM não foi possível analisar versões mais recentes da ferramenta Archimedes, o que fez com que não fosse possível avaliar a evolução do código-fonte e casos de teste da ferramenta. No entanto, foi possível constatar que o projeto Archimedes possui uma menor quantidade de casos de teste em relação a quantidade de código-fonte produzido, que resultou em valores de cobertura de código menores que no projeto Kalibro. A análise da quantidade de defeitos encontrados durante cada iteração nos dois projetos poderia ser relacionada e comparada em relação a melhoria da cobertura dos testes e quantidade de testes produzidos, no entanto tal análise não foi conduzida.

A coleta de métricas de acompanhamento da atividade de teste fornece um *feedback* constante sobre os conjunto de casos de teste que são desenvolvidos durante as iterações do projeto. A partir desse cenário, é possível que a equipe de desenvolvimento e os gerentes de projeto utilizem as informações coletadas em reuniões diárias e retrospectivas de iteração para a melhoria contínua do processo de teste e também dos artefatos de teste produzidos. As informações podem detectar problemas ou evoluções em termos da qualidade dos casos de teste produzidos. Além disso, a equipe pode estabelecer uma metodologia na qual são estabelecidas metas de qualidade para as métricas de acompanhamento de teste antes do início de uma iteração. As métricas podem ser gerenciadas em um intervalo de tempo

pré-estabelecido (diariamente ou por iterações). Conforme a experiência da equipe podem ser estabelecidos intervalos de referência para que um determinado valor de uma métrica seja considerado bom, regular ou ruim.

Para coleta das métricas de acompanhamento de testes e métricas de qualidade do código observou-se a importância de utilizar-se ferramentas de apoio como a ferramenta ATMM. Utilizando essas ferramentas é possível coletar as métricas rapidamente, utilizando uma interpretação única a respeito de cada métrica, com saídas que possam ser facilmente interpretadas por desenvolvedores e gerentes de projeto.

No próximo capítulo são apresentadas as considerações finais do trabalho, as dificuldades e limitações encontradas, os trabalhos futuros e as publicações esperadas que podem ser realizados a partir deste trabalho.

Capítulo

6

Conclusões e Trabalhos Futuros

Conforme discutido no decorrer deste trabalho, métodos de desenvolvimento ágil são técnicas adequadas para o desenvolvimento de software sujeito a mudanças constantes. Essas mudanças não devem afetar o cronograma e o orçamento do projeto e devem assegurar o atendimento às necessidades do cliente. A equipe de desenvolvimento ágil também se preocupa com a melhoria constante do processo e do produto durante todo o ciclo de desenvolvimento do software, no qual o progresso do projeto é avaliado diariamente. Para apoiar o *feedback* constante da equipe em projetos ágeis são utilizadas diversas práticas como reuniões diárias, reuniões de revisão, retrospectivas e uma área de trabalho informativa que deve dispor de instrumentos que forneçam dados sobre o andamento do projeto. A condução da atividade de teste utilizando as práticas e estratégias de teste criadas ou adaptadas para métodos ágeis também servem como uma forma de *feedback* instantâneo, no qual o desenvolvedor pode detectar em pouco tempo se o código desenvolvido ainda precisa ser modificado, refatorado e se foi implementado de acordo com as necessidades do cliente.

As informações sobre o andamento do projeto ágil podem ser coletadas por meio de métricas de software que devem medir o progresso, apontar melhorias e dificuldades durante todo o projeto. Nesse sentido, é necessário estabelecer quais métricas serão coletadas durante o desenvolvimento, além de estratégias para relacionar essas métricas. As métricas deverão reforçar os princípios ágeis, incentivar a comunicação, ampliar o conhecimento de toda a equipe a respeito do andamento do projeto e revelar dificuldades ou avanços em diversos aspectos do processo e do produto de software. Finalmente, é importante que essas métricas incentivem um alto nível de qualidade do software desenvolvido, e que a equipe de desenvolvimento e os gerentes de projeto reajam da forma correta conforme os resultados obtidos com

a coleta dessas métricas. Nesse sentido é importante que a equipe estabeleça um conjunto de métricas relacionadas à atividade de teste para fornecer mecanismos de avaliação dos casos de teste produzidos pela equipe e que facilite o acompanhamento da atividade de testes, que tem como objetivo avaliar a qualidade do software produzido.

Este trabalho procurou caracterizar a atividade de teste de software no contexto de métodos ágeis e propôs a adoção de um conjunto de métricas de acompanhamento para atividade de teste de software. Essas métricas foram aplicadas a dois estudos de casos com projetos que utilizaram o método XP e conduziram testes de unidade durante o seu desenvolvimento. Os resultados deste trabalho foram coletados a partir de informações fornecidas pelos responsáveis pelos projetos avaliados nos estudos de caso e o código-fonte e código de teste armazenado em seus respectivos respositórios. Para coleta dos resultados foi utilizada a ferramenta ATMM implementada neste trabalho, um questionário de informações gerais sobre projetos ágeis que foi adaptado para este trabalho e a ferramenta Kalibro que coletou métricas relacionadas a qualidade dos métodos produzidos, e também métricas de coesão e acoplamento.

Além da ferramenta ATMM, desenvolvida neste trabalho, ter apoiado a condução dos estudos de caso, a partir do cenário de uso da ferramenta foi possível descrever como a equipe de desenvolvimento e teste pode utilizar as métricas de acompanhamento a cada iteração, seja durante o desenvolvimento ou nas retrospectivas de cada iteração.

A partir do questionário de informações gerais foi possível constatar diferenças sob diversos aspectos como o tempo em que os dois projetos vêm sendo desenvolvidos, a quantidade de pessoas envolvidas e também a qualificação dos seus desenvolvedores. Os resultados em termos da aderência e maturidade da atividade de teste mostraram que os dois projetos já possuem uma boa maturidade em relação ao teste de unidade e a estratégia TDD, mas no entanto os testes de aceitação possuem alguma importancia somente ao projeto Kalibro e não são executados testes de sistema nos dois projetos. Os resultados da coleta de métricas de acompanhamento de teste evidenciaram uma preocupação com a melhoria dos casos de teste principalmente na ferramenta Kalibro. Além disso, também foi constatado aspectos positivos em relação a métricas de qualidade de software, sendo que o projeto Kalibro possui métodos pequenos que facilitam o seu entendimento, tem classes com um bom grau de coesão entre seus métodos e um baixo acoplamento entre suas classes. O projeto Archimedes possui métodos com uma média de linhas considerada regular e tem classes com um grau regular de coesão entre seus métodos e um grau de acoplamento regular entre suas classes.

6.1 Contribuições

As principais contribuições desse trabalho são as seguintes:

- Caracterização do processo, estratégias e práticas ligadas à atividade de teste de software no contexto de métodos de desenvolvimento ágil. Essa caracterização pode ser utilizada por desenvolvedores e testadores que desejam conduzir a atividade de teste em projetos ágeis.

- Descrição de um conjunto de métricas para acompanhamento da atividade de teste de software em projetos ágeis. Esse conjunto de métricas tem como objetivo fornecer informações para que a equipe acompanhe, avalie e apoie a melhoria contínua do processo, das práticas e a utilização eficiente de ferramentas de teste de software em projetos de desenvolvimento ágil.

- Implementação da ferramenta ATMM (*Agile Testing Metrics Management Tool*) que apoia o gerenciamento de iterações de desenvolvimento de software, fornecendo algumas das métricas de acompanhamento de teste sugeridas por este trabalho. Juntamente com a descrição da ferramenta, foi apresentado um cenário de uso explicando como a equipe de desenvolvimento e teste pode utilizar a ferramenta em um projeto de desenvolvimento ágil.

- Análise de dois projetos de software de domínios diferentes que utilizaram o método XP. A qualidade do software e dos casos de teste desenvolvidos nesses projetos foram avaliados segundo as métricas de acompanhamento de teste sugeridas neste trabalho e também segundo algumas métricas de qualidade implementadas pela ferramenta Kalibro que avaliam qualidade dos métodos produzidos, e também a coesão e acoplamento desses projetos.

- Um questionário para coleta de informações gerais de um projeto ágil a partir de termos e categorias sugeridas pelo *Extreme Programming Evaluation Framework* (XP-EF) (Williams et al., 2004b), além de outras informações úteis para caracterização de cada projeto, principalmente em termos de práticas ágeis e diversos aspectos da atividade de teste de software. Nesse questionário foi proposto um guia para medir o nível de aderência e também de maturidade da atividade de teste em projetos de desenvolvimento ágil.

6.2 Dificuldades e Limitações

A condução desta dissertação apresentou algumas dificuldades e limitações descritas a seguir:

- as métricas sugeridas neste trabalho foram adotadas de outros trabalhos ou propostas neste trabalho. No entanto, essas métricas foram validadas apenas utilizando estudos de caso com dois projetos acadêmicos. É necessário avaliar a utilização de métricas segundo métodos de validação teóricos e experimentais descritos na literatura (Briand et al., 1995; Kitchenham et al., 1995).
- limitações na implementação da ferramenta ATMM e dos *scripts* da ferramenta *JaBUTi* impossibilitaram a análise de versões mais recente do projeto Archimedes. A análise de versões mais recentes da ferramenta Archimedes poderia resultar em conclusões diferentes a respeito do projeto.
- foram analisadas e implementadas pela ferramenta ATMM apenas as métricas de acompanhamento de teste relacionadas ao teste de unidade.
- os trabalhos utilizados nos estudos de casos foram desenvolvidos em nível acadêmico, com um grau específico de conhecimento a respeito do domínio e também foram analisados projetos com tecnologias específicas que a ferramenta ATMM poderia dar suporte.

6.3 Trabalhos Futuros

Para dar continuidade as atividades realizadas neste trabalho, poderão ser realizados os seguintes trabalhos futuros:

- validação mais aprofundada das métricas utilizando métodos de validação teórica e experimental, que permitam afirmar que as métricas são validas a partir de critérios já definidos, além de fornecer evidências da validade ou invalidade das métricas propostas (Briand et al., 1995; Kitchenham et al., 1995).
- conduzir outro estudo de caso para que possa ser descrito a relação métricas de acompanhamento de teste sugeridas, definir valores de referência, além de compor novas métricas de acompanhamento de teste a partir das métricas propostas.
- replicação do estudo de caso em disciplinas de engenharia de software ou teste de software e também em projetos ágeis da indústria.
- aplicação em projetos reais, durante o seu desenvolvimento.

- melhorias na ferramenta ATMM para dar suporte a métricas de outras linguagens, métricas de teste de aceitação, possibilidade de coletar o código diretamente do repositório de código e integração com a ferramenta Kalibro para que seja possível configurar valores de referência para as métricas.

6.4 Publicações Esperadas

No decorrer do mestrado foi publicado um artigo que investigou questões relacionadas à importância da atividade de testes em métodos ágeis, ferramentas de teste utilizadas e estudos experimentais que evidenciam as implicações de uma abordagem ágil de testes.

- Vicente, A. A.; Delamaro, M. E.; Maldonado, J. C. Uma Revisão Sistemática sobre a Atividade de Teste de Software em Métodos Ágeis. In: *XXXV Conferencia Latinoamericana de Informática (XXXV CLEI)*, Pelotas - RS, Brasil: CLEI, 2009.

A partir dos resultados alcançados com o trabalho planeja-se submeter um artigo a alguma conferência da área de métodos ágeis.

- **Ferramenta ATMM e resultados do estudo de caso:** nesse artigo deve ser apresentada as métricas de acompanhamento de testes propostas no trabalho, a ferramenta de apoio que implementa algumas dessas métricas (ATMM) e os resultados do estudo de caso conduzido nesse trabalho.

Referências Bibliográficas

Abrahamsson, P.; Salo, O.; Ronkainen, J.; Warsta, J. *Agile software development methods - review and analysis.* Relatório Técnico 478, VTT PUBLICATIONS, Disponível em: `http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf`. Acesso em: 27/02/2010, 2002.

Abrahamsson, P.; Warsta, J.; Siponen, M. T.; Ronkainen, J. New directions on agile methods: a comparative analysis. In: *ICSE '03: Proceedings of the 25th icse, Portland, Oregon, USA*, Washington, DC, USA: IEEE Computer Society, 2003, p. 244–254.

Abran, A.; Bourque, P.; Dupuis, R.; Moore, J. W.; Tripp, L. L. *Guide to the software engineering body of knowledge - SWEBOK.* 2004 ed. Piscataway, NJ, USA: IEEE Press, 1–202 p., Disponível em: `http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf`. Acesso em: 15/01/2010, 2004.

Abrantes, J.; Travassos, G. Caracterização de Métodos Ágeis de Desenvolvimento de Software. In: *Proceedings of the 1st Workshop on Rapid Application Development (WDRA'2007)*, Brazilian Symposium on Software Quality, Porto de Galinhas, PE, Brasil, 2007.

Ambler, S. W.; Jeffries, R. *Agile modeling: effective practices for extreme programming and the unified process.* New York, NY, USA: John Wiley & Sons, Inc., 2002.

Ambu, W.; Concas, G.; Marchesi, M.; Pinna, S. Studying the evolution of quality metrics in an agile/distributed project. In: Abrahamsson, P.; Marchesi, M.; Succi, G., eds. *Extreme Programming and Agile Processes in Software Engineering, XP 2006, Oulu, Finland*, Springer, 2006, p. 85–93 (*Lecture Notes in Computer Science*, v.4044).

Barbosa, E. F. *Uma contribuição ao processo de desenvolvimento e módulos educacionais.* Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, orientador: Prof. Dr. José Carlos Maldonado, 2004.

Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*, cáp. 4 (Teste Estrutural) Rio de Janeiro, RJ: Elsevier, 2007.

Baresi, L.; Nitto, E. D.; Ghezzi, C. Towards open-world software: Issue and challenges. In: *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, Columbia, MD, USA*, Washington, DC, USA: IEEE Computer Society, 2006, p. 249–252.

Basili, V.; Caldiera, G.; Rombach, D. H. The goal question metric approach. In: *Encyclopedia of Software Engineering*, Wiley, 1994.

Beck, K. *Extreme programming explained: Embrace change.* 1st ed. Boston, MA, USA: Addison-Wesley Professional, 2000.

Beck, K. *Test driven development: By example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Beck, K.; Andres, C. *Extreme programming explained: Embrace change.* 2nd ed. Boston, MA, USA: Addison-Wesley, 2004.

Beck, K.; Beedle, M.; van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R. C.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D. Manifesto for agile software development. Disponível em: `http://www.agilemanifesto.org/` [23/12/2009], 2001.

Beck, K.; Fowler, M. *Planning extreme programming.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

Begel, A.; Nagappan, N. Usage and perceptions of agile software development in an industrial context: An exploratory study. In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain*, IEEE Computer Society, 2007, p. 255–264.

Bertolino, A. Software testing research: Achievements, challenges, dreams. *Future of Software Engineering, 2007. FOSE '07*, p. 85–103, 2007.

Binder, R. V. *Testing object-oriented systems: models, patterns, and tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

Bluemke, I.; Rembiszewski, A. Dataflow approach to testing java programs. In: *Fourth International Conference on Dependability of Computer Systems (DepCos-RELCOMEX '09)*, Brunów, Poland, 2009, p. 69–76.

Boehm, B. Get ready for agile methods, with care. *IEEE Computer*, v. 35, n. 1, p. 64–69, 2002.

Briand, L.; Emam, K. E.; Morasca, S.; El, K.; Morasca, E. S.; Informatique, C. D. R.; De, C. D. R. I.; Vinci, P. L. D. *Theoretical and empirical validation of software product measures.* Relatório Técnico, ISERN-95-03, International Software Engineering Research Network, 1995.

Budd, T. A. *Mutation analysis: Ideas, example, problems and prospects.* North-Holand Publishing Company, 1981.

Champion, S. Keys to agile testing success. Disponível em: `http://blog.utest.com/?p=55`. Acesso em: 06/11/2008, 2008.

Cockburn, A. *Agile software development.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Cockburn, A. *Agile software development: The cooperative game (2nd edition).* Addison-Wesley Professional, 2006.

Colette, D. Agile metrics presentation. Apresentação na Agile Conference 2009. Disponível em: `http://davenicolette.wikispaces.com/file/view/agile-metrics-v6.ppt`. Acesso em: 15/01/2010, 2009.

Crispin, L.; Gregory, J. *Agile testing: A practical guide for testers and agile teams.* Addison-Wesley Professional, 2009.

Davidsson, M.; Zheng, J.; Nagappan, N.; Williams, L.; Vouk, M. Gert: an empirical reliability estimation and testing feedback tool. In: *15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, IEEE Computer Society, 2004, p. 269–280.

Delamaro, M. E. *Proteum - um ambiente de teste baseado na análise de mutantes.* Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, orientador: Prof. Dr. José Carlos Maldonado, 1993.

Delamaro, M. E.; Barbosa, E. F.; Maldonado, J. C. *Introdução ao teste de software*, cáp. 5 (Teste de Mutação) Rio de Janeiro, RJ: Elsevier, 2007a.

Delamaro, M. E.; Maldonado, J. C. Proteum - a tool for the assessment of test adequacy for c programs: User's guide. In: *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, 1996, p. 79–95.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*, cáp. 1 (Conceitos Básicos) Rio de Janeiro, RJ: Elsevier, 2007b.

Delamaro, M. E.; Maldonado, J. C.; Mathur, A. P. Interface mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, v. 27, n. 3, p. 228–247, 2001.

DeMillo, R.; Guindi, D.; McCracken, W.; Offutt, A.; King, K. An extended overview of the Mothra software testing environment. *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, p. 142–151, 1988.

Deng, C. *FitClipse: a testing tool for supporting executable acceptance test driven development.* Tese de Doutoramento, Faculty of Graduate Studies, University of Calgary, Calgary, Alberta, Orientador: Prof. Dr. Frank Oliver Maurer, 2007.

Dingsøyr, T.; Dybå, T.; Abrahamsson, P. A preliminary roadmap for empirical research on agile software development. In: *Proceedings of Agile Conference 2008 (AGILE '08). Toronto, Canada*, Washington, DC, USA: IEEE Computer Society, 2008, p. 83–94.

Domingues, A. L. S. *Avaliação de critérios e ferramentas de teste para programas oo.* Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, orientador: Prof. Dr. José Carlos Maldonado, 2002.

Don Wells Extreme programming: A gentle introduction. Disponível on-line: `http://www.extremeprogramming.org/` [15/01/2008], 2006.

DSDM DSDM consortium. Disponível em: `http://www.dsdm.org`. Acesso em: 18/01/2010, 2009.

Dybå, T.; Dingsøyr, T. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, v. 50, n. 9-10, p. 833–859, 2008.

Dybå, T.; Dingsøyr, T. What do we know about agile software development? *IEEE Software*, v. 26, p. 6–9, 2009.

Eler, M. M.; Endo, A. T.; Masiero, P. C.; Delamaro, M. E.; Maldonado, J. C.; Vincenzi, A. M. R.; Chaim, M. L.; Beder, D. M. JaBUTiService: A Web Service for Structural Testing of Java Programs. In: *33rd Annual IEEE Software Engineering Workshop (SEW-33)*, Skövde, Sweden, 2009.

Fabbri, S. C. P. F.; Vincenzi, A. M. R.; Maldonado, J. C. *Introdução ao teste de software*, cáp. 2 (Teste Funcional) Rio de Janeiro, RJ: Elsevier, 2007.

Ferrari, F.; Maldonado, J.; Rashid, A. Mutation testing for aspect-oriented programs. In: *1st International Conference on Software Testing, Verification, and Validation*, 2008, p. 52–61.

Fowler, M. *Refactoring: improving the design of existing code.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

Freire, A. *Reflexões sobre o ensino de metodologias ágeis na academia, na indústria e no governo.* Dissertação de Mestrado, Instituto de Matemática e Estatística - Univesidade de São Paulo, São Paulo, SP, Orientador: Prof. Dr. Fabio Kon, 2007.

George, B.; Williams, L. A structured experiment of test-driven development. *Information and Software Technology*, v. 46, n. 5, p. 337 – 342, 2004.

Harrold, M. J.; Rothermel, G. Performing data flow testing on classes. In: *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, New York, NY, USA: ACM, 1994, p. 154–163.

Hartmann, D.; Dymond, R. Appropriate agile measurement: Using metrics and diagnostics to deliver business value. In: *Proceedings of Agile Conference 2006 (AGILE '06). Minnesota, USA*, Washington, DC, USA: IEEE Computer Society, 2006, p. 126–134.

Highsmith, J. *Agile software development ecosystems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Highsmith, J.; Cockburn, A. Agile software development: The business of innovation. *Computer*, v. 34, n. 9, p. 120–122, 2001.

Howden, W. E. *Functional program testing and analysis.* New York, NY, USA: McGraw-Hill, Inc., 1986.

Huo, M.; Verner, J.; Zhu, L.; Babar, M. A. Software quality and agile methods. *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004), Hong Kong, China*, v. 1, p. 520–525, 2004.

Itkonen, J.; Rautiainen, K. Exploratory testing: a multiple case study. In: *International Symposium on Empirical Software Engineering*, 2005, p. 84–93.

Jakobsen, C.; Sutherland, J. Scrum and CMMI going from good to great. In: *Proceedings of Agile Conference 2009 (AGILE '09). Chicago, IL, USA*, 2009, p. 333–337.

Janzen, D.; Saiedian, H. Test-driven development concepts, taxonomy, and future direction. *IEEE Computer*, v. 38, n. 9, p. 43–50, 2005.

Janzen, D. S. *An empirical evaluation of the impact of test-driven development on software quality.* Tese de Doutoramento, Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas, Kansas, Orientador: Hossein Saiedian, 2006.

Jeffries, R.; Melnik, G. Guest editors' introduction: TDD–the art of fearless programming. *IEEE Software*, v. 24, n. 3, p. 24–30, 2007.

Jeffries, R. E. A metric leading to agility. Disponível em: `http://xprogramming.com/xpmag/jatrtsmetric`. Acesso em: 13/01/2010, 2004.

Jeffries, R. E.; Anderson, A.; Hendrickson, C. *Extreme programming installed.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

Kan, S. H. *Metrics and models in software quality engineering.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Kitchenham, B.; Pfleeger, S. L.; Fenton, N. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, v. 21, p. 929–944, 1995.

Knoernschild, K. Using metrics to help drive agile software. Agile Journal. Disponível em: `http://www.agilejournal.com/articles/columns/the-agile-developer/56-using-metrics-to-help-drive-agile-software`. Acesso em: 15/01/2010, 2006.

Krebs, W. Turning the knobs: A coaching pattern for xp through agile metrics. In: Wells, D.; Williams, L. A., eds. *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Chicago, IL, USA*, Springer, 2002, p. 60–69 (*Lecture Notes in Computer Science*, v.2418).

Kulik, P. A practical approach to software metrics. *IT Professional*, v. 2, n. 1, p. 38–42, 2000.

Larman, C. *Agile and iterative development: A manager's guide.* Pearson Education, 2003.

Layman, L.; Williams, L.; Cunningham, L. Exploring extreme programming in context: An industrial case study. In: *ADC '04: Proceedings of the Agile Development Conference*, Washington, DC, USA: IEEE Computer Society, 2004, p. 32–41.

Lemos, O. A. L. *Teste de programas orientados a aspectos: Uma abordagem estrutural para AspectJ.* Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, Orientador: Prof. Dr. Paulo Cesar Masiero, 2005.

Ma, Y.-S.; Offutt, J.; Kwon, Y. R. Mujava: an automated class mutation system: Research articles. *Software Testing, Verification & Reliability*, v. 15, n. 2, p. 97–133, 2005.

Madeyski, L. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information & Software Technology*, v. 52, n. 2, p. 169–184, 2010.

Madeyski, L.; Radyk, N. Judy - a mutation testing tool for java. *IET Software*, *Accepted* - Disponível em: `http://madeyski.e-informatyka.pl/download/Madeyski10b.pdf`. Acesso em: 27/01/2010, 2010.

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, SP, orientador: Prof. Dr. Mário Jino, 1991.

Maldonado, J. C.; Fabbri, S. C. P. F. *Qualidade de software*, cáp. Teste de Software Prentice Hall, p. 73–84, 2001.

Marchenko, A.; Abrahamsson, P. Scrum in a multiproject environment: An ethnographically-inspired case study on the adoption challenges. In: *Agile Development Conference (AGILE 2008), Toronto, Canada*, IEEE, 2008, p. 15–26.

Marchesi, M. The new XP. Disponível em: `http://andvicente.googlepages.com/TheNewXP.pdf`. Acesso em: 15/01/2008, 2004.

Martin, R. C. Professionalism and test-driven development. *IEEE Software*, v. 24, n. 3, p. 32–36, 2007.

Masiero, P. C.; Lemos, O.; Ferrari, F. C.; Maldonado, J. C. *Jornadas de atualização em informática*, v. 1, cáp. Teste de Software Orientado a Objetos e a Aspectos: Teoria e Prática Editora PUC-Rio, p. 13–71, disponível em: `http://www.sbc.org.br/bibliotecadigital/download.php?paper=637`. Acesso em: 08/11/2007, 2006.

Meirelles, P. R. M.; Cóbe, R.; Hanazumi, S.; Nunes, P.; Challco, G.; Straus Martins, E. M.; Kon, F. Crab: Uma ferramenta de configuração e interpretação de métricas de software para avaliação de qualidade de código. In: *XXIII SBES - Simpósio Brasileiro de Engenharia de Software (XVI Sessão de Ferramentas)*, Fortaleza, CE, Brasil, 2009.

Melnik, G. *Empirical analysis of executable acceptance test driven development*. Tese de Doutoramento, Department of Computer Science, University of Calgary, Calgary, Alberta, Orientador: Prof. Dr. Frank Oliver Maurer, 2007.

Meszaros, G. *Xunit test patterns: Refactoring test code*. Addison-Wesley, 2007.

Mosley, H.; Mayer, A. *Benchmarking national labour market performance: A radar chart approach*. Relatório Técnico, European Commission, Employment, Industrial Relations and Social Affairs, 1998.

Mugridge, R. Managing agile project requirements with storytest-driven development. *IEEE Software*, v. 25, n. 1, p. 68–75, 2008.

Mugridge, R.; Cunningham, W. *Fit for developing software: Framework for integrated tests*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

Myers, G. J.; , C. S.; , T. B.; Thomas, T. M. *The art of software testing*. 2 ed. Wiley, 2004.

Nagappan, N. Toward a software testing and reliability early warning metric suite. In: *26th International Conference on Software Engineering (ICSE 2004), Edinburgh, United Kingdom*, IEEE Computer Society, 2004, p. 60–62.

Nagappan, N.; Williams, L.; Osborne, J.; Vouk, M.; Abrahamsson, P. Providing test quality feedback using static source code and automatic test suite metrics. In: *16th IEEE International Symposium on Software Reliability Engineering, 2005 (ISSRE 2005), Chicago, IL, USA*, IEEE Computer Society, 2005, p. 85–94.

Nakagawa, E. Y. *Uma contribuição ao projeto arquitetural de ambientes de engenharia de software*. Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, Orientador: Prof. Dr. José Carlos Maldonado, 2006.

Nascimento, G. V. *Um modelo de referência para o desenvolvimento ágil de software*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e Computação - Universidade de São Paulo, São Carlos, SP, Orientadora: Profa. Dra. Rosely Sanches, 2007.

Nerur, S.; Balijepally, V. Theoretical reflections on agile development methodologies. *Communications of the ACM*, v. 50, n. 3, p. 79–83, 2007.

Nerur, S.; Mahapatra, R.; Mangalaraj, G. Challenges of migrating to agile methodologies. *Communications of the ACM*, v. 48, n. 5, p. 72–78, 2005.

North, D. Introducing BDD. *Better Software magazine*, 2006.

Oliveira Filho, C. M. d. Kalibro: Uma ferramenta de configuração e interpretação de métricas de código-fonte. Trabalho de Conclusão de Curso, IME/USP. Orientador: Prof. Dr. Fabio Kon, 2009.

Paetsch, F. *Requirements engineering in agile software development*. Tese de Doutoramento, University of Calgary, Calgary, Alberta, Orientador: Prof. Dr. Frank Oliver Maurer, 2003.

Palmer, S. R.; Felsing, M. *A practical guide to feature-driven development*. Pearson Education, 2001.

Poppendieck, M.; Poppendieck, T. *Lean software development: An agile toolkit*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

Poppendieck, M.; Poppendieck, T. *Implementing lean software development: From concept to cash*. Addison-Wesley Professional, 2006.

Pressman, R. S. *Engenharia de software*. 6 ed. São Paulo: McGraw-Hill, 2006.

Puleio, M. How not to do agile testing. In: *AGILE 2006 Conference (AGILE 2006), Minneapolis, Minnesota, USA*, Washington, DC, USA: IEEE Computer Society, 2006, p. 305–314.

Qualipso Project QualiPSo (Quality Platform for Open Source Software). Disponível em: `http://www.qualipso.org`. Acesso em: 08/09/2007, 2006.

Qumer, A.; Henderson-Sellers, B. An evaluation of the degree of agility in six agile methods and its applicability for method engineering. *Information and Software Technology*, v. 50, n. 4, p. 280 – 295, 2008.

Ragland, B. Measure, metric, or indicator: What's the difference?, Disponível em: `http://www.stsc.hill.af.mil/crosstalk/1995/03/Measure.asp`. Acesso em: 05/01/2010, 1995.

Rapps, S.; Weyuker, E. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985.

Rapps, S.; Weyuker, E. J. Data flow analysis techniques for test data selection. In: *ICSE '82: Proceedings of the 6th international conference on Software engineering, Tokyo, Japan*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, p. 272–278.

Sato, D.; Bassi, D.; Goldman, A. Extending extreme programming with practices from other agile methodologies. In: *Proceedings of the 1st Workshop on Rapid Application Development (WDRA'2007)*, Porto de Galinhas: Brazilian Symposium on Software Quality, disponível em: `http://www.dtsato.com/resources/default/wdra2007.pdf` [14/01/2009], 2007a.

Sato, D.; Goldman, A.; Kon, F. Tracking the evolution of object-oriented quality metrics on agile projects. In: Concas, G.; Damiani, E.; Scotto, M.; Succi, G., eds. *Proceedings of Agile Processes in Software Engineering and Extreme Programming, 8th International Conference (XP 2007), Como, Italy, June*, Springer, 2007b, p. 84–92 (*Lecture Notes in Computer Science*, v.4536).

Sato, D. T. *Uso eficaz de métricas em métodos Ágeis de desenvolvimento de software.* Dissertação de Mestrado, Instituto de Matemática e Estatística - Universidade de São Paulo, São Paulo, SP, Orientador: Prof. Dr. Alfredo Goldman, 2007.

Schwaber, K. Scrum development process. In: *Workshop on Business Object Design and Implementation*, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95), Austin, Texas, disponível em: `http://jeffsutherland.com/oopsla/schwaber.html` [08/02/2007], 1995.

Schwaber, K. *Agile project management with scrum.* Redmond, WA, USA: Microsoft Press, 2004.

Schwaber, K.; Beedle, M. *Agile software development with scrum.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

Simons, A. J. H. Testing with guarantees and the failure of regression testing in extreme programming. In: Baumeister, H.; Marchesi, M.; Holcombe, M., eds. *Proceedings of Extreme Programming and Agile Processes in Software Engineering, 6th International Conference (XP 2005), Sheffield, UK*, Springer, 2005, p. 118–126 (*Lecture Notes in Computer Science*, v.3556).

Siniaalto, M.; Abrahamsson, P. A comparative case study on the impact of test-driven development on program design and test coverage. In: *First International Symposium on Empirical Software Engineering and Measurement, 2007 (ESEM 2007), Madrid, Spain*, IEEE Computer Society, 2007, p. 275–284.

Sommerville, I. *Software engineering.* 8 ed. Addison Wesley, 2006.

Stapleton, J. *DSDM: business focused development.* 2 ed. Pearson Education, 2003.

Strode, D. E. *The agile methods: An analitical comparison of five agile methods and an investigation of their target environment.* Dissertação de Mestrado, Massey University, New Zeland, Orientador: Prof. Dr. Alexei Tretiakov, 2005.

Thomas, S. An agile comparison. Disponível em: `http://www.itsadeliverything.com/articles/agile_comparison.htm`. Acesso em: 15/01/2010, 2006.

Tian, J. *Software quality engineering: Testing, quality assurance, and quantifiable improvement.* John Wiley and Sons Ltd., 2005.

Version One Survey: The state of agile development (2008), Disponível em: `http://www.versionone.com/pdf/3rdAnnualStateOfAgile_FullDataReport.pdf`. Acesso em: 12/11/2008, 2008.

Version One Survey: The state of agile development (2009), Disponível em: `http://www.versionone.com/pdf/2009_State_of_Agile_Development_Survey_Results.pdf`. Acesso em: 21/01/2010, 2010.

Vicente, A. A.; Delamaro, M. E.; Maldonado, J. C. Uma Revisão Sistemática sobre a Atividade de Teste de Software em Métodos Ágeis. In: *XXXV Conferencia Latinoamericana de Informática (XXXV CLEI)*, Pelotas - RS, Brasil: CLEI, 2009.

Vincenzi, A.; Delamaro, M. E.; Maldonado, J. C. JaBUTi - Java Bytecode Understanding and Testing (Project site). Disponível em: `http://ccsl.ime.usp.br/pt-br/project/jabuti`. Acesso em: 05/01/2010, 2006.

Vincenzi, A. M. R. *Orientação a objetos: Definição, implementação e análise de recursos de teste e validação.* Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, São Carlos, SP, orientador: Prof. Dr. José Carlos Maldonado, 2004.

Vincenzi, A. M. R.; Maldonado, J. C.; Delamaro, M. E.; Spoto, E. S.; Wong, W. E. Component-based software: An overview of testing. In: Cechich, A.; Piattini, M.; Vallecillo, A., eds. *Component-Based Software Quality*, Springer, 2003, p. 99–127 (*Lecture Notes in Computer Science*, v.2693).

Vincenzi, A. M. R.; Maldonado, J. C.; Wong, W. E.; Delamaro, M. E. Coverage testing of Java programs and components. *Science of Computer Programming*, v. 56, n. 1-2, p. 211–230, 2005.

Voigt, B. J. J. Dynamic system development method, Seminar of Department of Information Technology - University of Zurich, Disponível em: `http://www.ifi.uzh.ch/rerg/fileadmin/downloads/teaching/seminars/seminar_ws0304/14_Voigt_DSMD_Ausarbeitung.pdf`. Acesso em: 17/02/2008, 2004.

Vriens, C.; Barto, R. 7 years of agile management. In: *Agile Development Conference (AGILE 2008), Toronto, Canada*, IEEE, 2008, p. 390–394.

Warden, S.; Shore, J. *The art of agile development: With extreme programming.* O'Reilly Media, Inc., 2007.

Wege, C. *Automated support for process assessment in test-driven development.* Tese de Doutoramento, University of Tübingen, Tübingen, Germany, Orientador: Prof. Dr. Ulrich Güntzer, 2004.

Williams, L.; Cockburn, A. Agile software development: It's about feedback and change. *IEEE Computer*, v. 36, n. 6, p. 39–43, 2003.

Williams, L.; Krebs, W.; Layman, L.; Antón, A. I.; Abrahamsson, P. Toward a framework for evaluating extreme programming. In: *Empirical Assessment in Software Engineering (EASE), Edinburgh, Scotland*, 2004a, p. 11–20.

Williams, L.; Layman, L.; Krebs, W. *Extreme programming evaluation framework for object-oriented languages (version 1.4)*. Relatório Técnico, North Carolina State University, TR 2004-18, 2004b.

Zhu, H.; Hall, P. A. V.; May, J. H. R. Software unit test coverage and adequacy. *ACM Computing Surveys*, v. 29, n. 4, p. 366–427, 1997.

Apêndice

A

Questionário Informações Gerais (Projetos Ágeis)

O questionário a seguir tem como objetivo coletar dados gerais sobre projetos que utilizam métodos ágeis de desenvolvimento. Este questionário foi elaborado a partir de alguns dos termos e categorias sugeridas pelo *Extreme Programming Evaluation Framework* (XP-EF) (Williams et al., 2004b), além de informações gerais do projeto, foram coletadas informações referentes a práticas ágeis e diversos aspectos da atividade de teste de software.

A.1 Informações Gerais do Projeto e Contato

1.1 Nome do Projeto: _____

1.2 E-mail para contato: _____

1.3 Endereço(s) Web do projeto: _____

1.4 Endereço do Respositório de Código: _____

1.5 Licença do Software: ◯ Software Livre ◯ Software Comercial

1.6 Classificação do Software: ◯ Sistema de Software ◯ Software Comercial
◯ Sistema de Informação ◯ Software *Outsourced* ◯ Software Militar
◯ Software para Usuário Final

1.7 Breve Descrição: _____

A.2 Fator Ergonômico

2.1. Comunicação com o Cliente (meios utilizados):
☐ E-mail ☐ Pessoalmente ☐ Telefone

2.2. Comunicação com o Cliente (frequência): ☐ Alta ☐ Média ☐ Raramente

2.3. Comunicação com o Cliente (informações adicionais): _____

A.3 Fatores Tecnológicos

3.1 Método de Desenvolvimento: ☐ Programação Extrema (XP) ☐ SCRUM ☐ FDD (*Feature Driven Development*) ☐ *Lean Development* ☐ Outro: _____

3.2 Método de Desenvolvimento (Processo Híbridos ou Customizados): *(Se o projeto utilizou processos híbridos ou customizados, defina resumidamente que aspectos ou práticas que foram utilizadas).*

3.3 Linguagens: ☐ Java (J2SE) ☐ Java (J2EE) ☐ Python ☐ Ruby ☐ C ☐ C++ ☐ .NET ☐ Outras: _____

3.4 Tecnologias e/ou *Frameworks* utilizados: (Opcional)

3.5 Ferramentas (Ambientes de Desenvolvimento): (Opcional)

3.6 Ferramentas (Gerenciamento de Projeto): (Opcional)

3.7 FERRAMENTAS de Teste de Software: ☐ Ferramentas de Teste de Unidade ☐ Cobertura de Testes ☐ Gerenciamento de Casos de Teste ☐ Testes de Aceitação ☐ Teste de GUI ☐ Testes Funcionais (Selenium e similares) ☐ Testes de Desempenho ☐ Outras: _____

3.7.1 Ferramentas utilizadas (Teste de Software):

3.8 Outras Ferramentas: (Opcional)

A.4 Fatores Geográficos

4.1 Localização do Time: ☐ Time Local ☐ Time Distribuído ☐ Outra: _____

4.2 Clientes (Quantidade): ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ Outra: _____

4.3 Clientes (Localização): ○ Próprio Ambiente ○ Locais Diferentes ○ Próprio Ambiente e Locais Diferentes

A.5 Fatores Sociológicos

5.1.1 Tamanho da Equipe (Desenvolvedores) - Dedicação Exclusiva:

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ >5 (especificar qtde 5.1.4)

5.1.2 Tamanho da Equipe (Testadores) - Dedicação Exclusiva:

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ >5 (especificar qtde 5.1.4)

5.1.3 Tamanho da Equipe (*Coach*) - Dedicação Exclusiva:

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ >5 (especificar qtde 5.1.4)

5.1.4 Tamanho da Equipe (Outros): _____

5.2. Nível de Educação da Equipe (Ensino Básico, Técnico, Superior, Mestrado, Doutorado): _____

5.3 Nível de Experiência da Equipe:

○ Alto ○ Médio ○ Baixo ○ Sem conhecimento

5.4 Conhecimento do Domínio pela equipe:

○ Alto ○ Médio ○ Baixo ○ Sem conhecimento

A.6 Fatores Específicos do Projeto

6.1 Domínio(s): *(Exemplos de Domínio: Vendas Online, Computação Móvel...)*

_____

6.2 Duração do Projeto e Iterações: _____

6.3 Total de Histórias ou Itens de *Backlog* Entregues: (Opcional)

_____

6.4 Práticas Ágeis utilizadas (principais):

☐ Integração Contínua ☐ Cliente Presente ☐ Reuniões em-pé
☐ Área de Trabalho Informativa ☐ Código Compartilhado ☐ Programação em pares
☐ Planejamento da Iteração ☐ Refatoração

6.4.1 Outras Práticas Ágeis Utilizadas: _____

6.5 Práticas Ágeis de teste utilizadas (principais):

☐ Testes de Unidade ☐ Testes de Aceitação (*Business Testing*)
☐ TDD ☐ Testes de GUI / Usabilidade
☐ Testes Exploratórios ☐ Métricas de Código e Testes
☐ Testes de Requisitos Não-Funcionais (Ex.: Segurança, Usabilidade...)

6.5.1 Outras Práticas de Teste Utilizadas: _____

A.7 Atividade de Teste de Software

7.1 Aderência a Prática de Teste de Software[1]

7.1.1 Testes de Unidade Automatizados

Nenhuma das atividades ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 Segue todas as atividades

7.1.2 Testes de Aceitação com o Cliente (*Business Testing*)

Nenhuma importância ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 Total importância

7.1.3 Desenvolvimento Dirigido a Testes (TDD)

Não aplicado ○ 0 ○ 1 ○ 2 ○ 3 ○ 4 Totalmente aplicado

[1] Níveis de aderência conforme a Seção 5.2

7.2 Maturidade da atividade de Teste de Software

7.2.1 Testes de Unidade e TDD

☐ Não há nenhum teste formal. Os clientes geralmente avisam se encontrarem algum erro.

☐ Testes são executados no final de cada ciclo de desenvolvimento, no entanto há diversos *bugs* e diversas correções.

☐ Os Testes de Unidade são desenvolvidos para o código SOMENTE antes de entregá-lo para o time de teste.

☐ Após pensar no *design* e escrever um pouco de código, é escrito um teste automatizado.

☐ O TDD é utilizado, contribuindo para o *design* e qualidade do código.

☐ Preocupa-se com a utilização de padrões para criação de testes que incluem preocupação com a testabilidade do código.

7.2.2 Testes de Aceitação e Testes de Sistema

☐ Os clientes ou desenvolvedores rodam testes de aceitação para validar o software produzido.

☐ Execução de Testes mais complexos: testes de sistema (carga, performance...), testes com BD, e teste de interface.

7.2.3 Aspectos de Automatização dos Testes

☐ Utilizam-se SOMENTE testes manuais (sem apoio de nenhuma ferramenta).

☐ Utilizam-se ferramentas para facilitar o desenvolvimento e execução dos testes.

☐ Utilizam-se ferramentas para gerenciar os testes.

☐ Utilizam-se ferramentas para apoiar a melhoria dos testes.

☐ Treinamento para utilização de ferramentas de teste.

☐ Utilizam-se testes manuais.

7.2.4 Processo de Testes e Melhoria Contínua

☐ A atividade de teste é planejada e gerenciada desde o início do projeto.

☐ É estabelecida uma organização das atividades de teste.

☐ Utilização de métricas que controlam e monitoram os testes. Essas métricas são expostas na área informativa e discutidas em *stand-up meetings*, retrospectivas e reuniões de planejamento.

☐ Preocupação com a melhoria do código de testes e com a otimização do processo.

7.2.5 Importância, benefícios e dificuldades da atividade de teste de software

---

7.2.6 Melhoria da condução da atividade de Teste de Software

(O que você acredita que pode ser melhorado ou será melhorado nas próximas iterações do projeto a respeito da atividade de Testes)

A.8 Observações Finais / Sugestões

8. Observações Finais / Sugestões

Apêndice

B

Guia de Execução de *scripts JaBUTi*

Para facilitar a execução da ferramenta *JaBUTi*, foram utilizados quatro *shell scripts* para instrumentar, executar e gerar informações de cobertura de código de um projeto. Os *scripts* estão disponíveis em `http://ccsl.ime.usp.br/files/jabuti-scripts.zip`. São necessários os seguintes passos para executar os *scripts* da *JaBUTi*:

1. **Criar o jar do projeto:** Dentro do diretorio dos binários executar o comando "jar -cf ../nomedoarquivo.jar"

2. **Configurar o arquivo de configuração jbt:** As informações necessárias são descritas a seguir (arquivo jbt exemplo é apresentado por meio Código B.1):

   - JABUTI_HOME: Diretório da *JaBUTi*.
   - ORIG_DIR: Diretório original das classes (*.class).
   - TEST_SCRIPT: Diretório das classes de teste.
   - TEST_SCRIPT_OUT: Diretório aonde as saidas do TEST_SCRIPT são geradas.
   - TEST_EXEC_CMD: Comando para execução dos testes.
   - SPAGO_DIR: Diretório aonde o XML com informações de cobertura serão salvos.
   - SPAGO_ID: ID do projeto.
   - ORIG_JAR: Diretório e nome do pacote Jar com as classes originais.
   - INSTRUM_JAR: Diretório e nome do pacote Jar que será gerado pela instrumentação.

3. **Execução dos *scripts***: cada script tem que receber o argumento "-cfg" que corresponde ao arquivo de configuração. Por exemplo: "jabuti-instrument -cfg example.jbt". Quatro *scripts* serão executados na seguinte ordem:

   (a) **jabuti-initialize:** *script* analisa a aplicação que está sendo testada, então cria e configura alguns arquivos da *JaBUTi* e diretórios que serão utilizados pelos próximos *scripts*.

(b) **jabuti-instrument:** responsável por instrumentar as classes dentro do $ORIG_JAR. Se a variável não for informada, as classes do $ORIG_DIR serão instrumentadas. Como resultado, o pacote jar com as classes instrumentadas serão salvas e nomeadas de acordo com o valor passado na variável $INSTRUM_JAR. Se $INSTRUM_JAR. estiver vazia, um arquivo temporário com as classes instrumentadas serão gerados.

(c) **jabuti-execute:** pode ser executado esse script que executará o comando especificado em $TEST_EXEC_CMD, gerando em seguida os arquivos de trace. De forma alternativa, os casos de teste poderão ser executados pelo Eclipse utilizando os seguintes passos: (1) Run as ≫ Run Configurations, (2) Guia "Class Path" (incluir jar instrumentado no Classpath - user entries), (3) Guia Environment (incluir a variável BATCH_MODE (vazia) e JABUTI_HOME (caminho da *JaBUTi*).

(d) **jabuti-consolidate:** os arquivos de *trace* gerados durante a execução do conjunto de casos de teste serão analisados pelo *script*. Ele é responsável por gerar os relatórios com dados sobre a cobertura. No final do processo todos esses relatórios serão consolidados em um arquivo XML, com o identificador do projeto especificado no arquivo de configuração.

4. O arquivo XML com informações sobre a cobertura de todo o projeto analisado poderá ser informado a ferramenta ATMM, que irá extrair as informações de cobertura sobre o projeto.

```
############### CONFIGURATION FILE ###############
##################################################
# Instructions:
# 1. Fill the fields using the absolut path
# 2. Do use the "/" for directory names
##################################################
# ----------------------------------------------
#                    Mandatory
# ----------------------------------------------
BASE_DIR="/home/avicente/workspace/Kalibro01"

# JABUTI INFO
JABUTI_HOME="/home/avicente/JaBUTi"

# Original classes Directory
ORIG_DIR="$BASE_DIR/bin"

# Test Execution
TEST_SCRIPT="$BASE_DIR/test"
TEST_SCRIPT_OUT="$BASE_DIR"
TEST_EXEC_CMD=""

# Spago Information
SPAGO_DIR=$BASE_DIR
SPAGO_ID=CoberturaKalibro01
# ----------------------------------------------
#                    Optional
# ----------------------------------------------
# Original classes Jar
ORIG_JAR="$BASE_DIR/kalibro.jar"

# Instrumented classes Jar
INSTRUM_JAR="$BASE_DIR/kalibro_instrum.jar"
```

Listing B.1: Arquivo JBT - Informações do Projeto que será analisado a cobertura