UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

ANDRÉ MESQUITA RINCON

Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais

Goiânia
2011

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE DISSERTAÇÃO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais

Autor(a): André Mesquita Rincon

Goiânia, 19 de Maio de 2011.

André Mesquita Rincon – Autor

Dr. Auri Marcelo Rizzo Vincenzi – Orientador

ANDRÉ MESQUITA RINCON

Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Auri Marcelo Rizzo Vincenzi

Goiânia
2011

ANDRÉ MESQUITA RINCON

Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 19 de Maio de 2011, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Auri Marcelo Rizzo Vincenzi
Instituto de Informática – UFG
Presidente da Banca

Prof. Dr. Plínio de Sá Leitão Júnior
Universidade Federal de Goiás – UFG

Prof. Dr. Marcos Lordelo Chaim
Universidade de São Paulo – USP

André Mesquita Rincon

Graduou-se em Sistemas de Informação pelo Centro Universitário Luterano de Palmas CEULP/ULBRA e Possui Pós-Graduação Lato Sensu em Melhoria do Processo de Software pela Universidade Federal de Lavras - MG. Foi engenheiro de testes da Motorola no Brazil Test Center em Recife - PE e Gerente de Projetos na Diretoria de Tecnologia da Informação e Comunicação da UNITINS em Palmas - TO. Atualmente é Professor da Universidade do Tocantins (UNITINS) e Professor Efetivo do Instituto Federal de Educação, Ciência e Tecnologia do Tocantins (IFTO) - Campus Paraíso do Tocantins.

À Roberta, ao Matheus, ao Ruimar e à Carmelita.

Agradecimentos

- A Deus por me conceder uma vida com saúde e pelas boas pessoas que ele me permitiu conhecer durante essa jornada.

- À minha esposa Roberta pelo companheirismo, compreensão e paciência. Pelo apoio incondicional nos processos de mudança que enfrentamos para eu poder cursar o mestrado. Por acreditar em mim e me apoiar nos momentos em que achei que não conseguiria. Por ter renunciado aos seus interesses pessoais e aceitado me acompanhar na busca de um sonho.

- Ao meu orientador e amigo prof. Dr. Auri Marcelo Rizzo Vincenzi pelo apoio, confiança, contribuições, profissionalismo na orientação desta dissertação e por compreender/conduzir com muita paciência os momentos de dificuldades que enfrentamos.

- Ao meu filho Matheus que é minha fonte de inspiração e coragem para continuar buscando meus sonhos.

- Aos meus pais, Ruimar e Carmelita, pelas orações, confiança, apoio e pelos ensinamentos que foram muito importantes nesta caminhada até a realização deste sonho.

- Aos meus irmãos, Kárita, Vinicius e Jorge pelo convívio e companheirismo.

- Aos professores do CEULP/ULBRA pelos incentivos e orientações que me deram nesta busca pela carreira acadêmica, em especial ao Fabiano Fagundes, meu grande irmão, pela amizade, conselhos e ensinamentos que foram fundamentais para a realização deste trabalho.

- Aos amigos de sala de aula do mestrado que fizeram esta caminhada mais divertida e tornaram muito mais agradáveis os momentos dentro e fora da UFG, Bruno Machado, Marcelo Quinta, Fabiana Freitas, Renan Rodrigues, Elizabeth Kowata,

Glauber Boff, Jair Abul, Sofia Costa, Daniel Ferreira, Cassio Camilo, Elisangela, Patrícia Gomes, Thiago Rosa, Thiago Borges e Leandro Alexandre. Em especial ao Luiz Loja, Rogério de Paula Carvalho e Adriana Rocha pela amizade e por estarem junto comigo e me apoiarem nos momentos mais difíceis.

- Aos parentes de Goiânia e Orizona, Denise Mesquita, Leandro Brito, Luisa, Luana, Leandrinho, Rafael Leão, José Mesquita e Isaura Leão pelo convívio e companheirismo. Em especial, à Tia Lindaura e ao Victor Rocha pela moradia, amizade, apoio e incentivo.

- Aos meus grandes amigos, Edeilson Milhomem, Alison Alvares, Rafael Osório, Jorge Kleber, Múcio Renato, Alfredo Beckert, Leandro Maciel, Lucas Bechert, Jorges Borges, Carlos Eduardo de Lima e Fabio Varanda pelo companheirismo e incentivo constante.

- A todos os professores do Instituto de Informática da UFG pelo convívio e orientação acadêmica durante este período de curso.

- Aos servidores técnico-administrativos do Instituto de Informática da UFG, em especial ao Edir pela recepção e apoio.

- Aos amigos professores e técnico-administrativos da UNITINS, em especial Igor Yepes, Vinícius Rios, André Pugliese, Rodrigo Barbosa, Soely Kunz, Geraldo Gomes e Maurício Silva pelo convívio e pelas palavras de incentivo.

- Aos amigos que me possibilitaram e ajudaram a cursar este mestrado, Galileu Guarenghi, Claudemir Andreaci, Paula Karini, Maria Lourdes (Lula) e Marcelo Liberato.

- Aos amigos do IFTO - Campus Paraíso do Tocantins pelo apoio e palavras de incentivo.

- À Fundação Universidade do Tocantins (UNITINS) pelo suporte financeiro.

Resumo

Rincon, André Mesquita. **Qualidade de conjuntos de teste de software de código aberto: uma análise baseada em critérios estruturais**. Goiânia, 2011. 95p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

O projeto QualiPSo (*Quality Platform for Open Source Software*) tem por objetivo investigar produtos de software de código gratuito/livre/aberto (*Free/Libre/Open Source Software* – FLOSS) para definir requisitos de qualidade que são importantes para se estabelecer a confiabilidade desses produtos. Uma das atividades do projeto QualiPSo visa avaliar a qualidade de conjuntos de teste desenvolvidos pelas comunidades de software livre. Esta dissertação de mestrado está inserida neste contexto e apresenta os resultados do emprego de critérios de teste estruturais como uma medida da qualidade de conjuntos de teste funcionais visando a identificar o estado-da-prática das atividades de teste desempenhadas pelas comunidades de software livre, bem como, a contribuir no estabelecimento de uma estratégia de teste incremental para evoluir os conjuntos de testes.

Palavras–chave

Teste de software, critérios estruturais, critérios funcionais, FLOSS, conjuntos de teste, estratégia incremental

Abstract

Rincon, André Mesquita. **The quality of open source software test sets: structural testing criteria-based analysis**. Goiânia, 2011. 95p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

The QualiPSo Project (Quality Platform for Open Source Software) has as goal to investigate Free/Libre/Open Source Software (FLOSS) products to define quality requirements that are important to set the products reliability. One of the QualiPSo Project activities is to evaluate the quality of the developed test sets by the FLOSS community. This work is part of the QualiPso Project and shows the results of the use of structural test criteria as a functional test sets quality measure to identify the state-of-the-practice of performed test activities by free software communities. Furthermore, this work contributes to establish an incremental test strategy to improve the test sets.

Keywords

Software testing, structural criteria, functional criteria, FLOSS, test sets, incremental testing strategy

Sumário

Lista de Figuras **11**

Lista de Tabelas **12**

Lista de Códigos de Programas **13**

1 Introdução **14**
- 1.1 Contexto e Motivação 16
- 1.2 Objetivos 18
- 1.3 Organização da Dissertação 18

2 Fundamentação Teórica **20**
- 2.1 Teste de Software 20
  - 2.1.1 Conceitos relacionados ao Teste de Software 21
- 2.2 Teste Estrutural 24
  - 2.2.1 Grafo de Fluxo de Controle 25
  - 2.2.2 Critérios de teste estruturais 28
    - Critérios baseados na complexidade 29
    - Critérios baseados em fluxo de controle 31
    - Critérios baseados em fluxo de dados 35
  - 2.2.3 Níveis de cobertura de Código 39
  - 2.2.4 Ferramentas automatizadas para apoio ao teste estrutural 45
    - EMMA 46
- 2.3 Teste de Software FLOSS 49
  - 2.3.1 FLOSS 49
  - 2.3.2 Desenvolvimento de Software em comunidades FLOSS 50
- 2.4 Contexto do trabalho: o projeto QualiPSo 53
  - 2.4.1 Objetivos do projeto QualiPSo 53
  - 2.4.2 Divisão do projeto QualiPSo em atividades 54
  - 2.4.3 Avaliação de conjuntos de teste de software baseada em critérios estruturais 55

3 Discussão e Análise **56**
- 3.1 Experimentos Realizados 56
  - 3.1.1 Processo geral dos experimentos com a EMMA 57
  - 3.1.2 Materiais 60
    - Hardware 60
    - Software 60
    - Projetos Java analisados 60
  - 3.1.3 Detalhes específicos de cada projeto 61

Canoo WebTest 61
HttpUnit 62
JFreeChart 64
JMeter 65
Log4j 67
Mondrian 69
Poi 70
Velocity 72
Weka 74
Xerces2 75
3.2 Considerações sobre os resultados dos experimentos 76
3.3 Estratégia incremental 79

4 Conclusões **85**
4.1 Contribuições 86
4.2 Produção Bibliográfica 86
4.3 Trabalhos Futuros 87

Referências Bibliográficas **89**

Lista de Figuras

2.1 Cenário tipico da atividade de teste (adaptada de Delamaro et al. (2007)) 22
2.2 Passos básicos para se aplicar um critério de teste de caixa branca 24
2.3 GFC correspondente à função principal do programa *identifier* 28
2.4 GFC correspondente à função principal do programa *identifier* com as regiões que representam a complexidade ciclomática 30
2.5 GFC correspondente ao Código *evaluate.java* 33
2.6 GFC correspondente ao Código *evaluate.java* com as regiões que representam a complexidade ciclomática 34
2.7 Grafo Def-Uso correspondente à função principal do programa *identifier* (extraído de Barbosa et al. (2007)) 36
2.8 Níveis de cobertura segundo Copeland (COPELAND, 2004) 40
2.9 GFC que representa o Código 2.4 41
2.10 Diferentes caminhos possíveis para o GFC que representa o Código 2.4 41
2.11 Grafo das condições múltiplas do código 2.5 43
2.12 Casos de teste e caminhos do grafo das condições múltiplas do código 2.5 43
2.13 Exemplo de cobertura de código utilizando a Emma (ROUBTSOV, 2010) 47

3.1 Processo geral para realização dos experimentos com a EMMA 57
3.2 Exemplo de relatório gerado pela ferramenta EMMA em nível de projeto 58
3.3 Exemplo de relatório gerado pela ferramenta EMMA em nível de pacote 59
3.4 Exemplo de relatório gerado pela ferramenta EMMA em nível de classe 59
3.5 Execução da estratégia no processo de teste 80
3.6 Modelo distribuído para auxiliar na evolução contínua do conjunto de teste 81

Lista de Tabelas

2.1 Elementos e critérios associados em relação à função principal do programa *identifier* 29
2.2 Casos de teste com valores que devem ser atribuídos em cada estrutura de decisão programa *evaluate* 35
2.3 Elementos requiridos pelo critério Todas-Definições para o Grafo Def-Uso da Figura 2.7 (exemplo adaptado de (BARBOSA et al., 2007)) 38
2.4 Elementos requiridos pelo critério Todos-Usos para o Grafo Def-Uso da Figura 2.7 (exemplo adaptado de (BARBOSA et al., 2007)) 39

3.1 Canoo WebTest: resumo da cobertura obtida 62
3.2 HttpUnit: resumo da cobertura obtida 63
3.3 JFreeChart: resumo da cobertura obtida 65
3.4 JMeter: resumo da cobertura obtida 67
3.5 Log4j: resumo da cobertura obtida 68
3.6 Mondrian: resumo da cobertura obtida 70
3.7 Poi: resumo da cobertura obtida 72
3.8 Velocity: resumo da cobertura obtida 73
3.9 Weka: resumo da cobertura obtida 75
3.10 Xerces2: resumo da cobertura obtida 76
3.11 Resumo das coberturas e quantidades de classes dos projetos analisados 77
3.12 Detalhamento das coberturas dos projetos analisados 78

Lista de Códigos de Programas

2.1 identifier.c 26
2.2 Complemento do programa indentifier 27
2.3 evaluate.java 32
2.4 Código para exemplos dos níveis 1 e 2 de cobertura 40
2.5 Código para exemplos dos Nível 3 de cobertura 42
2.6 Código para análise do Nível 4 de cobertura 42
2.7 Aumento da quantidade de caminhos 44

CAPÍTULO 1

Introdução

Existem sinais de uma ampla divulgação dos conceitos de código aberto na indústria e no governo. Grandes empresas como IBM e Nokia já consideram código aberto em suas operações de pesquisa e desenvolvimento. Governos dos países membros da União Européia, do Brasil e da China consideram *Free/Libre/Open Source Software* (FLOSS) como uma oportunidade chave para o desenvolvimento de uma indústria de software independente (QUALIPSO, 2010).

No entanto, ainda há uma relutância quanto à adoção massiva de FLOSS devido, principalmente, à falta de "confiança" (QUALIPSO, 2010). Segundo Meirelles (2003), em termos quantitativos, a utilização de FLOSS ainda é expressivamente inferior à utilização de alternativas proprietárias. (QUALIPSO, 2005) afirma que essa falta de "confiança" está relacionada a: questões jurídicas; um modelo de negócio que possa garantir sustentabilidade; e aspectos de qualidade do software (por exemplo: ciclo de desenvolvimento, suporte, confiabilidade e desempenho).

Normalmente, produtos de software que são amplamente utilizados pelos usuários e considerados confiáveis (como, por exemplo, o Servidor Apache (Apache Foundation, 2010a), o Hibernate (JBOSS, 2010), o JasperReports (Jasper Forge, 2010), o Eclipse (ECLIPSE, 2010), o JUnit (JUNIT, 2010), entre outros) adquiriram essa reputação à medida que foram se consolidando como um software de bom desempenho durante sua utilização, ou seja, cria-se um consenso geral de que eles possuem qualidade e podem ser utilizados em um contexto profissional, mas não há critérios formais que permitam chegar a esse consenso. Por outro lado, têm-se os produtos que não possuem o nível de confiabilidade atribuída pelos usuários, seja por ser um produto novo ou por ser pouco utilizado, o que pode contribuir para uma desconfiança e dificultar sua adoção em um contexto profissional.

Embora se acredite que o desenvolvimento de FLOSS gera produtos de qualidade e que possam ser amplamente utilizados, os debates sobre como isso acontece e o que é necessário para repetir o sucesso de produtos FLOSS existem há algum tempo (MCCONNELL, 1999). Além disso, estudos recentes, como (MORASCA et al., 2009) e (PETRINJA et al., 2009) são exemplos que sugerem que é necessário estabelecer elementos que possam

garantir a qualidade de produtos FLOSS.

Uma das iniciativas que surgiu neste contexto foi o projeto QualiPSo (*Quality Platform for Open Source Software*) (QUALIPSO, 2010). O QualiPSo é considerado um projeto integrado que visa consolidar a aliança natural da indústria, do governo e da academia com FLOSS provendo o nível de qualidade que a indústria necessita relacionado a questões legais e de modelos de negócio, além de buscar estabelecer requisitos de confiabilidade aos projetos FLOSS por meio da investigação dos processos de garantia da qualidade adotados na sua produção (QUALIPSO, 2005).

Um processo de garantia da qualidade em software deve se ater a diversas dimensões do produto e do processo. O software deve ser considerado dentro de uma perspectiva multidimensional em que o processo de desenvolvimento está por trás dele. Essa multidimensionalidade reside no fato de que, para desenvolver um código fonte de alta qualidade, outros artefatos são necessários como, por exemplo: especificações, restrições, documentação e testes (MENS et al., 2005).

A geração desses artefatos ocorre durante todo o desenvolvimento do software por meio da realização de uma série de atividades que devem ser executadas de forma disciplinada e sistemática no ciclo de vida do software. No entanto, mesmo com a evolução significativa da Engenharia de Software, com o estabelecimento de técnicas, métodos e ferramentas, os produtos de software desenvolvidos ainda podem conter defeitos[1]. Assim sendo, atividades agregadas sob o nome de Garantia de Qualidade de Software são introduzidas ao longo de todo o processo de desenvolvimento, dentre elas as atividades de Verificação e Validação (V&V), das quais o teste é uma das mais utilizadas visando à redução da ocorrência de falhas e riscos associados (VINCENZI, 2004).

Dentre as atividades de V&V, o teste pode ser visto como uma atividade complementar a outras, como por exemplo, o uso de revisões e de técnicas formais de especificação e de verificação, e constitui um dos elementos que fornecem evidências da confiabilidade e qualidade do produto de software desenvolvido (MALDONADO et al., 1998).

O teste de produtos de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados (PRESSMAN, 2006). Dentro deste cenário, visando fornecer uma maneira organizada para geração e avaliação de conjuntos de teste, técnicas, critérios e ferramentas são desenvolvidas de maneira a fornecer ao testador uma abordagem sistemática e teoricamente fundamentada que constitui um mecanismo que pode auxiliar a avaliar a qualidade da atividade de teste (VINCENZI, 2004).

Sendo assim, o projeto QualiPSo acredita que as atividades de garantia da qua-

[1]Neste texto utilizam-se as definições de engano, defeito, erro e falha, conforme descrito no *IEEE Standard Glossary of Software Engineering Terminology* – 610.12-1990 (R2002) (IEEE, 2002)

lidade aplicadas pelas comunidades FLOSS no desenvolvimento dos softwares devem ser investigadas para se propor melhorias, quando necessário, de forma a agregar valor à confiabilidade desses produtos e que isto, aliado aos outros aspectos que estão sendo analisados dentro do projeto, poderão levar os produtos FLOSS aos patamares de qualidade que a indústria necessita.

1.1 Contexto e Motivação

No contexto do projeto QualiPSo, conforme citado anteriormente, diversos aspectos relacionados à confiabilidade dos produtos FLOSS estão sendo investigados, além disso, citou-se que para se atribuir qualidade e confiabilidade a produtos de software em geral, diversos processos, como, por exemplo, V&V, devem ser considerados. No entanto, em produtos FLOSS, decorrente das características do modelo de desenvolvimento (que serão discutidos na seção 2.3.2) com pouco foco no planejamento e na especificação de requisitos, quando um produto FLOSS evolui, geralmente o principal artefato que é considerado nesse processo de evolução é o código fonte. Sendo assim, este artefato pode ser considerado como uma fonte rica de informação sobre a qualidade do software desenvolvido.

Para avaliar a qualidade dos códigos fonte, podem-se utilizar métodos estáticos e dinâmicos de verificação. A verificação estática é focada em análises do código fonte sem levar em consideração a execução do programa. Dentre alguns trabalhos que tratam da análise estática encontram-se os seguintes.

Morasca et al. (2009) propuseram um modelo de maturidade do processo de teste de software para projetos de software livre chamado *Open-Source Software Testing Maturity Model* (OSS-TMM). Para demonstrar sua aplicabilidade, os autores utilizaram o OSS-TMM para analisar dois projetos, BusyBox (ANDERSEN, 2010) e HTTP Apache (Apache Foundation, 2010a). Além disso, quatro projetos representativos foram avaliados com o OSS-TMM para correlacionar os níveis de maturidade com as taxas de defeito apresentadas pelos projetos. O objetivo foi avaliar se quanto maior a maturidade do processo de teste, maior qualidade do produto.

Stamelos et al. (2002) analisaram o tamanho dos componentes e a qualidade em termos de satisfação dos usuários. Eles observaram que, até certo ponto, o tamanho médio de um componente é negativamente correlacionado com a satisfação do usuário para a aplicação.

Midha (2008) avaliou 450 projetos disponíveis no site `sourcefourge.net`. Os modelos elaborados indicam que, em média, FLOSS com alta complexidade estrutural (complexidade ciclomática de McCabe (MCCABE, 1976) e métrica de esforço de Halstead

(HALSTEAD, 1977)) estão significativamente associados com a presença de defeitos, com maior tempo de correção e com menor atratividade de novos desenvolvedores.

Meirelles et al. (2010) avaliaram a correlação entre qualidade do código e a sua atratividade, isto é, a habilidade do projeto em atrair usuários e desenvolvedores. Eles observaram que a atratividade é correlacionada com métricas de código fonte.

Ploesch et al. (2010) desenvolveram um método para avaliação de qualidade interna do software – EMISQ – que proporciona um arcabouço metodológico para análise de código utilizando métricas estáticas. Além disso, eles construíram uma ferramenta baseada em Eclipse que suporta o método EMISQ.

Gruber et al. (2008) aplicaram os métodos e as ferramentas do projeto QBench (QBENCH, 2010) a fim de investigar os pontos fortes e fracos da abordagem. Eles concluíram que o cálculo do índice de qualidade não conduz a resultados satisfatórios e, por isso, desenvolveram uma série de métodos alternativos e os compararam com os resultados do cálculo original. Segundo os autores, algumas dessas variantes de cálculo levam a uma melhor caracterização da qualidade de software em relação ao algoritmo QBench original.

Os estudos relatados envolvem análise do processo de teste ou do código fonte dos projetos de software livre. Quando o processo de teste é analisado, o que foi verificado é se tarefas e produtos de teste são gerados. Nesses estudos não é avaliada a qualidade intrínseca dos produtos, em especial dos conjuntos de casos de teste. No caso dos estudos de código fonte, há uma análise do código desenvolvido, mas não dos testes criados.

Diferentemente dos trabalhos anteriores, este trabalho está focado na verificação dinâmica e tem por objetivo verificar a qualidade dos testes desenvolvidos nos projetos de software livre por meio da avaliação da cobertura fornecida por eles. Para isso, projetos FLOSS que possuem conjuntos de teste funcionais desenvolvidos pela comunidade, em geral, no formato do arcabouço JUnit (JUNIT, 2010), foram avaliados visando aferir a qualidade dos conjuntos de teste disponíveis utilizando-se como métrica, a cobertura que os mesmos obtiveram em relação a critérios de teste estruturais conhecidos.

Segundo Zaidman et al. (2008), os códigos de teste e códigos de produção devem ser desenvolvidos e mantidos de forma síncrona, pois: 1) novas funcionalidades devem ser testadas o mais breve possível no processo de desenvolvimento, por exemplo, por meio de testes de unidade, muito disseminados na comunidade FLOSS (RUNESON, 2006); 2) quando mudanças são aplicadas, a preservação de comportamento do software precisa ser verificada (DEMEYER et al., 2002); e 3) mesmo quando as alterações preservam o comportamento, os testes podem ser invalidados (MOONEN et al., 2008), pois pequenas alterações no código de produção podem ter sérias consequências sobre a cobertura de código que o conjunto de teste irá abranger (ELBAUM et al., 2001).

Dados que os produtos FLOSS investigados possuem um conjunto de teste funci-

onal (caixa-preta) criado e disponibilizado pela comunidade de desenvolvimento visando a verificação constante do FLOSS desenvolvido, a pergunta que motivou a realização deste trabalho foi "Qual a qualidade desses conjuntos de testes funcionais?". Acredita-se que os conjuntos de testes empregados pelas comunidades devam ser analisados na busca por requisitos que possam auxiliar no estabelecimento de confiabilidade a produtos FLOSS.

Além disso, outro fator de motivação para avaliação da qualidade dos conjuntos de teste está em um estudo realizado por Laprévote et al. (2009). Segundo os autores, após analisar 11 projetos FLOSS de renome, constatou-se que a atividade de teste é realizada de forma *ad hoc*. Eles observaram que apenas três projetos possuem planos de teste e que não havia uma estratégia clara para desenvolver os testes em sete dos onze projetos. Além disso, segundo eles, o fato mais preocupante é que os testes de unidade só são realizados em cinco desses projetos. Os estudos de Laprévote et al. (2009) contataram ainda que os conjuntos de testes associados a esses projetos FLOSS também foram desenvolvidos de forma *ad hoc*.

1.2 Objetivos

Considerando o contexto e a motivação apresentados acima, os objetivos deste trabalho são:

- Avaliar a qualidade dos conjuntos de teste disponibilizados pelas comunidades em relação a critérios de teste estruturais visando investigar qual a porcentagem de cobertura de código de produção que é efetivamente executada por tais conjuntos de teste, considerando produtos FLOSS desenvolvidos na linguagem Java (Sun Microsystems, 2010);

- Apresentar os procedimentos necessários e os resultados das análises que foram realizadas em produtos FLOSS, bem como os artefatos gerados durante o processo;

- Propor uma estratégia de teste para comunidades FLOSS que propicie a evolução dos conjuntos de teste existentes de modo incremental.

1.3 Organização da Dissertação

Além deste capítulo inicial que apresentou a introdução, motivação e objetivos, o restante do texto desta dissertação está organizado conforme descrito nos próximos parágrafos.

No Capítulo 2, são apresentadas as fundamentações teóricas que são utilizadas para o desenvolvimento deste trabalho. Os principais assuntos discutidos são: teste de software, teste estrutural, teste de software FLOSS e o projeto QualiPSo.

No Capítulo 3, apresentam-se os detalhes dos experimentos realizados caracterizando: o processo de execução dos experimentos, os materiais utilizados, os resultados obtidos e uma proposta de estratégia de testes incremental.

No Capítulo 4, apresentam-se as conclusões finais, contribuições, produção bibliográfica e as perspectivas de trabalhos futuros.

Além disso, os arquivos dos projetos analisados com as modificações, bem como os arquivos dos relatórios de cobertura compõem a dissertação em mídia digital (DVD) conforme descrito na Seção 3.1.

CAPÍTULO 2

Fundamentação Teórica

Este Capítulo apresenta as fundamentações teóricas que são utilizadas para o desenvolvimento deste trabalho. A Seção 2.1 traz os conceitos relacionados ao teste de software e algumas considerações sobre o processo de software e a garantia da qualidade. Na Seção 2.2 são apresentados os critérios de teste estrutural, o Grafo de Fluxo de Controle que é utilizado no estudo desses critérios, os níveis de cobertura de código que representa a hierarquia entre os critérios de teste estrutural, ferramentas para apoio ao teste estrutural e detalha a ferramenta EMMA que foi utilizada neste trabalho. A Seção 2.3 discute o conceito de FLOSS e apresenta algumas características do seu modelo de desenvolvimento e de teste. Por fim, a Seção 2.4 descreve o projeto QualiPSo que forneceu o contexto ao qual este trabalho está inserido.

2.1 Teste de Software

O teste de software consiste em uma das atividades de garantia da qualidade que possui a finalidade de verificar se o produto em desenvolvimento está em conformidade com sua especificação (DELAMARO et al., 2007). O IEEE (2004) define teste de software como uma verificação dinâmica do comportamento do programa, utilizando um conjunto de teste finito, devidamente selecionado do domínio de execuções, para ver se ele está de acordo com o esperado.

O teste de software, ou processo de teste de software, é considerado uma das áreas de conhecimento da Engenharia de Software conforme o IEEE (2004). Engenharia de Software consiste no estabelecimento e uso de sólidos princípios de Engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais (PRESSMAN, 2006).

Para garantir que o software possua as características mencionados acima, além da verificação dinâmica do comportamento do software, existem outras atividades de garantia de qualidade que são denominadas de Verificação e Validação (V&V). Juntas, as atividade de V&V ajudam a descobrir os defeitos antes do software ser liberado para utilização.

Como o objetivo principal desde trabalho é o teste de software, outras atividades de V&V, como, por exemplo, revisões técnicas e *walkthroughs*, não serão tratadas neste texto, mas salienta-se que são atividades que permitem a eliminação de outros tipos de defeitos desde as fases iniciais do processo de desenvolvimento, o que, em geral, representa uma economia significativa de recursos (DELAMARO et al., 2007).

Para utilizar processos de garantia de qualidade de software, dentre eles o teste, se faz necessário decorrentes das características do software que tornam o seu desenvolvimento não trivial, como, por exemplo: a complexidade, a instabilidade de requisitos e a invisibilidade (BROOKS JR., 1987).

Crespo et al. (2004) afirmam que existem dificuldades na realização do processo de teste que são decorrentes das características do software, dentre elas: o teste de software é considerado um processo caro, pois existe uma falta de conhecimento sobre a relação custo/benefício do teste; há o desconhecimento de técnicas de teste adequadas; há o desconhecimento sobre como planejar a atividade de teste; e o fato da preocupação com a atividade de teste existir, em geral, somente na fase final do projeto.

Para minimizar as dificuldades na condução do processo de teste, bem como reduzir a presença de defeitos, IEEE (2004) defende que o teste de software deve ser definido como parte integrante do ciclo de vida do software e envolver pessoas, ferramentas, políticas e medições na realização das seguintes atividades: planejamento e organização do ambiente de teste; geração de casos de teste; execução dos testes; avaliação dos resultados; comunicação e armazenamento de problemas encontrados; e rastreamento de defeitos (*defect tracking*).

A Seção 2.1.1 apresenta alguns conceitos relacionados ao processo de teste de software.

2.1.1 Conceitos relacionados ao Teste de Software

Na Seção 2.1 afirma-se que as atividades de V&V ajudam a descobrir defeitos, no entanto, vale tecer alguns comentários sobre o significado da palavra “defeito” no contexto de teste e desenvolvimento de software. Segundo o *IEEE Standard Glossary of Software Engineering Terminology* – 610.12-1990 (R2002) (IEEE, 2002), **defeito** (do inglês *fault*) consiste em um passo, processo ou definição de dados incorretos e **engano** a ação humana que produz o defeito, este defeito, se não for corrigido, poderá ocasionar a existência de um **erro** no programa que, por sua vez, poderá levar a uma **falha** (do inglês *failure*) que significa a apresentação de um resultado diferente do esperado.

Outros termos comumente encontrados na literatura sobre teste de software são: dado de teste, caso de teste e conjunto de testes. A Figura 2.1 ilustra um cenário típico da atividade de teste e pode auxiliar na compreensão desses termos.

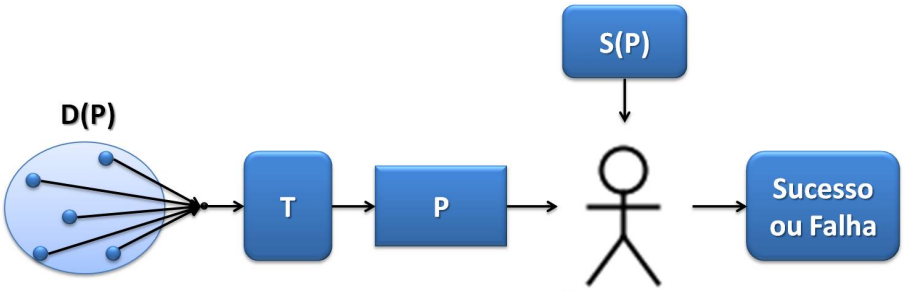


Figura 2.1: *Cenário tipico da atividade de teste (adaptada de Delamaro et al. (2007))*

A Figura 2.1 ilustra um cenário típico da atividade de teste. O conjunto de elementos denotado por **D(P)** na Figura 2.1 consiste em todos os possíveis valores que podem ser utilizados para executar o programa **P**, em outras palavras, **D(P)** é o domínio de entrada de **P**. Considerando o exemplo apresentado em Delamaro et al. (2007), dado um programa que recebe como parâmetros de entrada dois números inteiros x e y, com x e $y >= 0$, e computa o valor de $x + y$, indicando um erro caso os argumentos estejam fora do intervalo especificado, o domínio de entrada deste programa será formado por todos os possíveis pares de números inteiros (x, y) e o domínio de saída será o conjunto de todos os possíveis resultados produzido pelo programa, ou seja, o conjunto de números inteiros e mensagens de erro.

Nesse contexto, um dado de teste para o programa **P** é um elemento do domínio de entrada **D(P)**. Já um caso de teste será um par formado por um dado de teste mais o resultado esperado para execução do programa, por exemplo: [(2,2),4], [(3,3),6], [(4,a),“Erro”]. Ao conjunto de todos os casos de teste utilizados durante uma determinada atividade de teste denomina-se conjunto de teste (DELAMARO et al., 2007). Observa-se então que, para a criação de casos de teste é de fundamental importância a existência de um oráculo que é capaz de indicar, para cada dado de teste fornecido, qual a saída esperada conforme a especificação do produto em teste. Em geral, o papel de oráculo é desempenhado pelo testador em função da dificuldade de automatização deste processo.

Após a definição de um conjunto de teste **T** extraído a partir da análise do domínio de entrada do programa **D(P)**, executa-se o programa **P** com as entradas **T** e verifica-se quais os resultados obtidos. Se, para algum caso de teste, o testador encontrar um resultado diferente do esperado em relação à especificação **S(P)**, então um defeito foi revelado (DELAMARO et al., 2007).

O cenário apresentado na Figura 2.1 ocorre em todas as fases do teste do software. Delamaro et al. (2007) definem as fases do teste de software como: **teste de unidade** em que pequenas partes do código (como classes, métodos ou funções) são testadas separadamente pelo próprio desenvolvedor à medida que o código é construído; **teste de integração** que ainda deve ser realizado pela própria equipe de desenvolvimento

para verificar se as unidades testadas na fase anterior funcionam corretamente quando colocadas para trabalhar em conjunto; e **teste de sistemas** em que o sistema será testado por completo e as funcionalidades especificadas nos documentos serão analisadas (o ideal é que este teste seja realizado por uma equipe que não tenha contato direto com os desenvolvedores).

Independente da fase, um ponto chave para execução dos testes está na seleção dos dados de teste de um determinado domínio. Idealmente, o programa em teste deveria ser executado com todos os elementos do domínio para garantir que não tem defeitos, mas tal abordagem é infactível por causa da quantidade de elementos do domínio (DELAMARO et al., 2007), em geral, infinita. Retomando o exemplo utilizado na explicação da Figura 2.1, tem-se que o domínio é formado por todos os pares de números inteiros (x, y), ou seja, considerando o tipo inteiro com 32 bits, o total de dados de teste seria de $2^{32} + 2^{32} = 2^{64}$.

Assim, o ideal é encontrar formas de utilizar apenas um subconjunto reduzido de **D(P)**, mas que tenha alta probabilidade de revelar defeitos. Para isso, utiliza-se o teste de partição em que o domínio é dividido em subdomínios que possuem valores com características semelhantes e, posteriormente, alguns elementos representativos desses subdomínios são selecionados para execução dos testes. Para divisão dos subdomínios são estabelecidas algumas regras para identificar quais dados de teste farão parte de quais subdomínios. Em geral, são definidos "requisitos de teste" como, por exemplo, executar uma determinada estrutura do programa. Os dados de teste que satisfazem esse requisito tendem a pertencer ao mesmo subdomínio (DELAMARO et al., 2007).

Diferentes tipos de testes podem ser utilizados para verificar se um programa se comporta como o especificado e, para cada tipo de teste, a definição dos subdomínios e dos requisitos de teste são definidas pelo tipo de informação utilizada para realização do teste. Além disso, o tipo de informação disponível define a técnica de teste que será empregada. Segundo Delamaro et al. (2007), as técnicas de teste são classificadas em: **funcionais** (caixa-preta ou *black-box testing*) cujos testes são baseados exclusivamente na especificação de requisitos do programa e nenhum conhecimento de como o programa está implementado é requerido; **estruturais** (caixa-branca ou *white-box testing*) em que os testes são baseados na estrutura interna do programa, ou seja, na codificação dele; e **baseada em defeito** (*fault-based testing*) cujos testes são baseados em informações históricas sobre defeitos cometidos frequentemente durante o processo de desenvolvimento de software. Dado que este trabalho está focado nos critérios de teste estruturais, a Seção 2.2 apresenta detalhes sobre essa técnica e critérios relacionados.

2.2 Teste Estrutural

A técnica de teste estrutural (caixa branca ou *white box testing*) se baseia nos caminhos internos, estrutura e implementação do produto em teste, ou seja, demanda conhecimento do código do produto em teste para ser aplicada. Ela estabelece os requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou componentes elementares do programa (PRESSMAN, 2006).

Nesta técnica, os caminhos lógicos do software são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis. A técnica estrutural é vista como complementar às demais técnicas de teste existentes, uma vez que cobre classes distintas de defeitos (BARBOSA et al., 2007). As informações obtidas pela aplicação de critérios estruturais podem auxiliar nas atividades de manutenção e depuração do Código (PRESSMAN, 2006), pois, ao contrário da técnica de caixa preta, os resultados dos testes possibilitam análises relacionadas diretamente ao código fonte do produto em teste.

As técnicas de teste possuem diferentes critérios. Cada critério possibilita um particionamento diferente do domínio de entrada e, consequentemente, a análise diferente dos resultados dos testes. Os critérios de teste definem os requisitos e dados de teste que serão selecionados do domínio de entrada do programa de acordo com o objetivo do teste. Os critérios de teste de caixa branca podem ser utilizados em todas as fases de teste, mas são mais comuns no teste de unidade e de integração (BARBOSA et al., 2007). A Figura 2.2 representa os passos básicos para se aplicar um critério de teste de caixa branca.

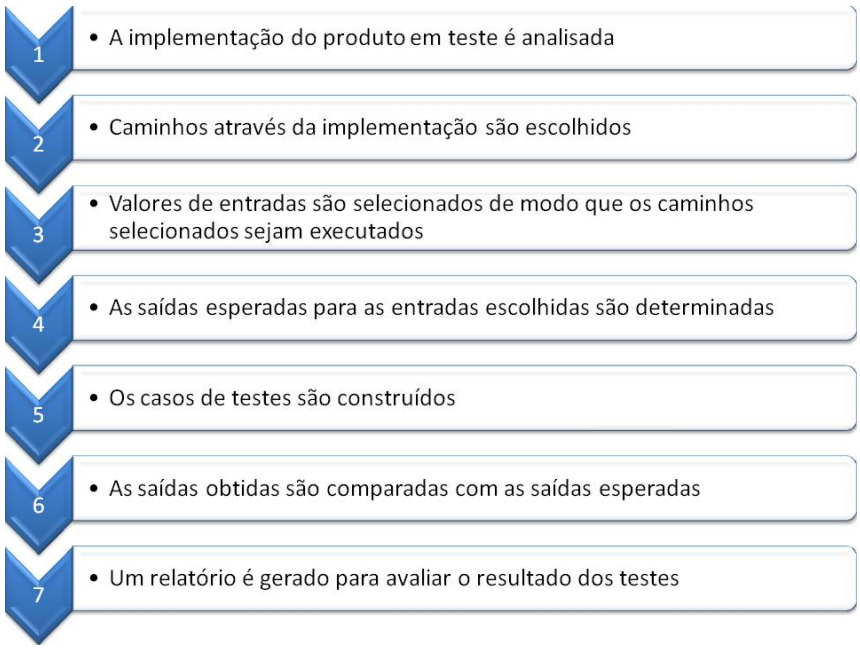


Figura 2.2: *Passos básicos para se aplicar um critério de teste de caixa branca*

O primeiro passo para se aplicar um critério de teste de caixa branca consiste na análise da implementação do produto em teste, em outras palavras, deve-se estudar o código fonte da aplicação. Depois se definem os caminhos da implementação que se pretende executar (por exemplo, partes essenciais ou críticas do programa). O terceiro passo é selecionar dados de teste do domínio de entradas do programa que garantam que os caminhos selecionados serão executados. Em seguida, as saídas esperadas para cada uma das entradas escolhidas são determinadas. O quinto passo é a construção dos casos de teste. O sexto passo é a comparação das saídas obtidas com as saídas esperadas para verificar o sucesso ou falha de cada caso de teste. Por fim, um relatório com os resultados deve ser gerado para análise.

Os Passos 2 e 3 da Figura 2.2 citam caminhos que perpassam a implementação. Esses caminhos são parte de um grafo denominado de Grafo de Fluxo de Controle (GFC) que é utilizado pelos critérios de teste estrutural. Os GFC's são utilizados para abstrair o fluxo de controle lógico de um programa. Eles são compostos por nós e arcos. Um **nó** consiste em uma ou mais instruções as quais são sempre executadas em sequência, ou seja, uma vez executada a primeira instrução de um nó, todas as demais instruções daquele nó também são executadas; já um **arco** (ou aresta) representa o fluxo de controle entre blocos de comandos (nós). A seção seguinte apresenta um exemplo de programa escrito na linguagem C e seu GFC correspondente.

2.2.1 Grafo de Fluxo de Controle

Esta seção apresenta uma definição de Grafo de Fluxo de Controle e ilustra, por meio de um exemplo para um código escrito na linguagem C, como se dá esse tipo de representação. GFC pode ser definido como um grafo orientado com um único nó de entrada e um único nó de saída, em que cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro (BARBOSA et al., 2007).

O programa denominado *identifier* (Códigos 2.1 e 2.2) é um exemplo adaptado de (BARBOSA et al., 2007) e será utilizado para exemplificar como um GFC pode representar a estrutura lógica de um software. O programa tem a funcionalidade de determinar se um dado identificador é válido ou não baseado nas seguintes regras: ele deve começar com uma letra; deve conter apenas letras ou dígitos; deve ter no mínimo 1 e no máximo 6 caracteres de comprimento. O código foi dividido em função principal (Código 2.1) e funções complementares (Código 2.2) apenas para efeito de organização do texto, no entanto, para o correto funcionamento do programa deve-se considerar os códigos em apenas um arquivo.

Código 2.1 identifier.c

```
1                main()
2  /*  1 */     {
3  /*  1 */        char achar;
4  /*  1 */        int length, valid_id;
5  /*  1 */        length = 0;
6  /*  1 */        valid_id = 1;
7  /*  1 */        printf("Identificador: ");
8  /*  1 */        achar = fgetc (stdin);
9  /*  1 */        valid_id = valid_s(achar);
10 /*  1 */        if(valid_id)
11 /*  2 */        {
12 /*  2 */             length = 1;
13 /*  2 */        {
14 /*  3 */        achar = fgetc (stdin);
15 /*  4 */        while(achar != '\n')
16 /*  5 */        {
17 /*  5 */             if(!(valid_f(achar)))
18 /*  6 */             {
19 /*  6 */                   valid_id = 0;
20 /*  6 */             {
21 /*  7 */             length++;
22 /*  7 */             achar = fgetc (stdin);
23 /*  7 */        }
24 /*  8 */        if(valid_id && (length >= 1) && (length < 6))
25 /*  9 */        {
26 /*  9 */             printf("Valido\n");
27 /*  9 */        }
28 /* 10 */        else
29 /* 10 */        {
30 /* 10 */             printf("Invalido\n");
31 /* 10 */        }
32 /* 11 */     }
```

Código 2.2 Complemento do programa indentifier

```
1                int valid_s(char ch)
2   /*  1 */     {
3   /*  1 */        if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a')
4                           && (ch <= 'z')))
5   /*  2 */        {
6   /*  2 */             return (1);
7   /*  2 */        }
8   /*  3 */        else
9   /*  3 */        {
10  /*  3 */             return (0);
11  /*  3 */        }
12  /*  4 */     }
13
14               int valid_f(char ch)
15  /*  1 */     {
16  /*  1 */        if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a')
17                          && (ch <= 'z')) || ((ch >= '0') && (ch <='9')))
18  /*  2 */        {
19  /*  2 */             return (1);
20  /*  2 */        }
21  /*  3 */        else
22  /*  3 */        {
23  /*  3 */             return (0);
24  /*  3 */        }
25  /*  4 */     }
```

A função principal do programa *identifier* (Código 2.1) possui números representados como comentários antes de cada linha de código e cada número corresponde a um nó do GFC do programa. O primeiro bloco de comandos é formado pelas linhas 2 a 10. O segundo bloco refere-se às linhas 11 a 13. O terceiro bloco é formado apenas pela linha 14 e o quarto apenas pela linha 15. O quinto bloco é formado pelas linhas 16 e 17. O sexto pelas linhas 18 a 20. O sétimo pelas linhas 21 a 23. A linha 24 corresponde ao oitavo bloco. O nono bloco é formado pelas linhas 25 a 27. O décimo pelas linhas 28 a 31 e o último bloco que representa o encerramento do programa, está na linha 11.

A estruturação do GFC leva em consideração os tipos de comandos que representa a lógica do programa. Por exemplo, blocos de código que possuem estruturas de seleção podem gerar desvios no GFC da seguinte maneira: o comando *if* (linha 10 do Código 2.1) consiste em um desvio de execução entre os nós do programa em que, caso

sejam exercitados os comandos internos do *if*, tem-se um desvio de execução do nó 1 para o nó 2, do contrário, tem-se um desvio do nó 1 para o nó 3. A Figura 2.3 ilustra o GFC que correspondente à função principal do programa *identifier*.

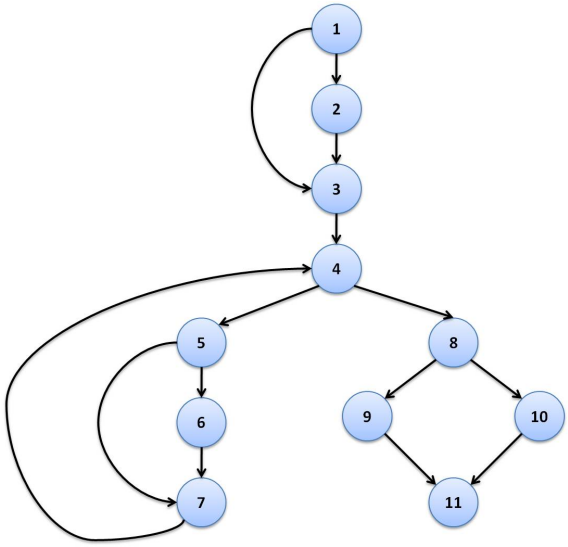


Figura 2.3: *GFC correspondente à função principal do programa identifier*

Na Figura 2.3, cada círculo representa um bloco de código e cada bloco individual é denominado de nó (exemplo: 1, 2, 3, etc.). Cada par de nós executados em sequência são chamados de arcos, por exemplo: (2,3), (5,6) e (8,10). Além disso, outro conceito que pode ser visualizado em um GFC é o caminho. Por exemplo: (2,3,4,5,6,7) é considerado um caminho simples e livre de laços; já o caminho (1,2,3,4,5,7,4,8,9,11) é chamado de caminho completo pois começa no nó inicial e termina no nó final; e (6,7,4,5,7,4,8,9) é considerado não executável e qualquer caminho completo que o inclua é também não executável, ou seja, não existe um dado de entrada que leve à execução desse caminho (BARBOSA et al., 2007).

Como citado anteriormente, as técnicas de teste possuem diferentes critérios. O GFC pode ser considerado como ponto de partida para o entendimento dos critérios de teste estruturais que são classificados com base na complexidade, no fluxo de controle e no fluxo de dados (PRESSMAN, 2006). Os critérios de teste estruturais serão detalhados na seção a seguir.

2.2.2 Critérios de teste estruturais

Os critérios de teste estruturais baseiam-se em diferentes tipos de conceitos e componentes de programas para determinar os requisitos de teste (BARBOSA et al., 2007). Na Tabela 2.1 são ilustrados alguns elementos requeridos da função principal do programa *identifier* (Código 2.1) em função da aplicação de critérios de teste estrutural.

Tabela 2.1: *Elementos e critérios associados em relação à função principal do programa identifier*

Elemento	**Exemplo (*identifier*)**	**Critério**
Nó	6	Todos-Nós
Arco	(5,6)	Todas-Arestas
Caminho	(1,2,3,4,8,9,11)	Todos-Caminhos
Definição de variável	length = 0	Todas-Defs
Uso predicativo de variável	achar != '\n'	Todos-p-Usos
Uso computacional de variável	length++	Todos-c-Usos

Os elementos das linhas 2, 3 e 4 da Tabela 2.1 estão relacionados à critérios baseados em fluxo de controle. Já os elementos das linhas 5, 6 e 7 são referentes aos critérios de fluxo de dados. Além disso, há também os critérios baseados na complexidade.

Critérios baseados na complexidade

Os Critérios Baseados na Complexidade utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. Um critério bastante conhecido dessa classe é o Critério de McCabe (ou teste do caminho básico), que utiliza a complexidade ciclomática do grafo de programa para derivar os requisitos de teste (MCCABE, 1976). Essencialmente, esse critério requer que um conjunto de caminhos linearmente independentes do grafo de programa seja executado (PRESSMAN, 2006).

Um caminho linearmente independente consiste em qualquer caminho do programa que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição, quando estabelecido em termos de um GFC (como o representado na figura 2.3), um caminho linearmente independente deve incluir pelo menos um arco que não tenha sido exercitado anteriormente (BARBOSA et al., 2007).

O Critério de McCabe estabelece um conjunto básico de caminhos linearmente independentes para o GFC (PRESSMAN, 2006) e, para cada caminho, deve ser criado um caso de teste de maneira a forçar a sua execução visando a garantir que cada desvio de execução do programa tenha sido exercitado pelo menos uma vez (BARBOSA et al., 2007).

Pressman (2006) afirma que, para saber quantos caminhos devem ser procurados, é necessário calcular a complexidade ciclomática do GFC que pode ser de três maneiras:

1. Realizar a subtração do número de arcos pelo número de nós do GFC e somar o valor 2. No exemplo da Figura 2.3 o resultado do cálculo seria 14 (número de arcos) menos 11 (número de nós) mais 2 que é igual a 5; ou

2. Realizar a soma do número de nós predicativos (aqueles que possuem estruturas *if* ou *while*) mais o valor 1. No exemplo da função principal do programa *identifier* (Código 2.1) o resultado do cálculo seria 4 (linhas 10, 15, 17 e 24) mais 1 que é

igual a 5; ou

3. Realizar a contagem do número de regiões de um GFC em que, cada região, pode ser informalmente descrita como uma área incluída no plano do grafo. Dessa maneira, o número de regiões é computado contando-se todas as áreas delimitadas e a área não delimitada fora do grafo. A Figura 2.4 ilustra a contagem de regiões para o GFC da função principal do programa *identifier*.

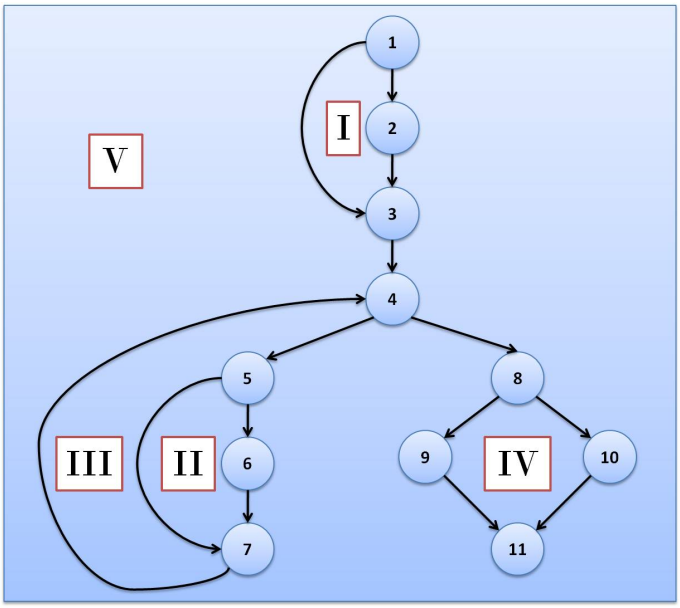


Figura 2.4: *GFC correspondente à função principal do programa identifier com as regiões que representam a complexidade ciclomática*

Como pode ser verificado na Figura 2.4, nos cálculos realizados com a quantidade geral de arestas e nós ou somente com os nós predicativos a complexidade ciclomática do GFC da função principal do programa *identifier* é 5. Esse valor corresponde ao número de caminhos linearmente independentes do GFC e, consequentemente, um limite mínimo do número de casos de teste que deve ser projetado e executado para garantir a cobertura de todas os desvios condicionais do programa (PRESSMAN, 2006).

No caso da função principal do programa *identifier* pode-se ter o seguinte conjunto básico de caminhos linearmente independentes: (1,2,3,4,8,9,11), (1,2,3,4,8,10,11), (1,2,3,4,5,7,4, ...), (1,2,3,4,5,6,7,4, ...) e (1,3,4, ...). As reticências significam que qualquer caminho a partir do último nó é aceitável. Neste caso, seria necessário apenas 5 casos de teste para exercitar todos os desvios condicionais do programa.

Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias (BARBOSA et al., 2007). Os critérios mais conhecidos dessa classe são (PRESSMAN, 2006):

- Todos-Nós: exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo de fluxo de controle, ou seja, que cada comando do programa seja executado pelo menos uma vez;

- Todas-Arestas: requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez;

- Todos-Caminhos: exige que todos os caminhos possíveis do programa sejam executados.

A cobertura do critério Todos-Nós é o mínimo esperado de uma "boa" atividade de teste, pois, dessa maneira, pode-se garantir que cada instrução do programa foi exercitada ao menos uma vez (BARBOSA et al., 2007). Pelo exemplo do GFC (Figura 2.3) da função principal do programa *identifier*, os elementos requeridos para este critério são: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 e 11.

Ainda pelo exemplo do GFC da Figura 2.3, os elementos requeridos pelo critério Todas-Arestas são: (1,2), (1,3), (5,6), (5,7), (8,9) e (8,10). É possível notar que alguns nós não estão aparecendo na formação dos elementos das arestas, como o 2, o 4 e o 11, isso ocorre devido ao fato de apenas os arcos primitivos serem considerados na composição dos elementos deste critério. Arcos primitivos são aqueles que quando exercitados garantem a execução de todos os demais arcos ditos não essenciais (BARBOSA et al., 2007).

Já o critério Todos-Caminhos, que exige que todos os caminhos possíveis do programa sejam executados, embora desejável, pode ser uma tarefa impraticável, pois à medida que são acrescentados laços de repetição na lógica do programa, o número de caminhos a ser percorrido pode ser infactível (BARBOSA et al., 2007). Assim, uma alternativa viável para este critério é o teste do caminho básico que utiliza a complexidade ciclomática para representar o número mínimo de caminhos independentes sem laços.

Além disso, um problema relacionado ao teste estrutural é a impossibilidade, em geral, de se determinar automaticamente se um caminho é ou não executável. Em outras palavras, em geral, não existe um algoritmo que, dado um caminho completo qualquer, decida se o caminho é executável e forneça o conjunto de valores que causa a execução

desse caminho. Desse modo, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para o programa sendo testado (BARBOSA et al., 2007).

A seguir será apresentado um exemplo do processo para determinar os dados de teste que exercitam os caminhos do Código 2.3.

Código 2.3 `evaluate.java`

```
1  boolean evaluate(Ticker Symbol ts){
2      s1; s2; s3;
3      if(c1){
4          s4; s5; s6;}
5      else{
6          s7; s8;}
7      while(c2){
8          s9; s10;
9          switch(c3){
10             case-A:
11                 s20; s21; s22; break;
12             case-B:
13                 s30; s31;
14                 if(c4){
15                     s32; s33; s34;}
16                 else{
17                     s35;} break;
18             case-C:
19                 s40; s41; break;
20             case-D:
21                 s50;break;
22         }
23         s60; s61; s62;
24         if(c5){
25             s70;s71;}
26         s80; s81;
27     }
28     s90; s91; s92;
29     return result;
30 }
```

O Código 2.3 consiste em um exemplo escrito na linguagem JAVA, mas que não realiza nenhum processamento especifico. Ele foi escrito apenas para ilustrar os desvios que as diferentes estruturas lógicas do programa podem causar na representação do GFC.

Por exemplo, os comandos do tipo *if* nas linhas 3, 14 e 24 geram dois caminhos diferentes na sequência do GFC; o comando do tipo *while* na linha 7 gera uma aresta de retorno ao nó que representa o início do laço de repetição; e o comando do tipo *switch* na linha 9 gera um novo caminho para cada comando *case* que há no programa (neste caso são 4 que estão nas linhas 10, 12, 18 e 20). Os elementos presentes nas linhas 2, 4, 6, 8, 11, 13, 15, 17, 19, 21, 23, 25, 26 e 28 (definidos como s1, s2, s3, etc.) são apenas ilustrativos e significam quaisquer trechos de códigos que seriam executados em sequência caso o fluxo fosse direcionado a eles. A Figura 2.5 corresponde ao GFC do Código 2.3.

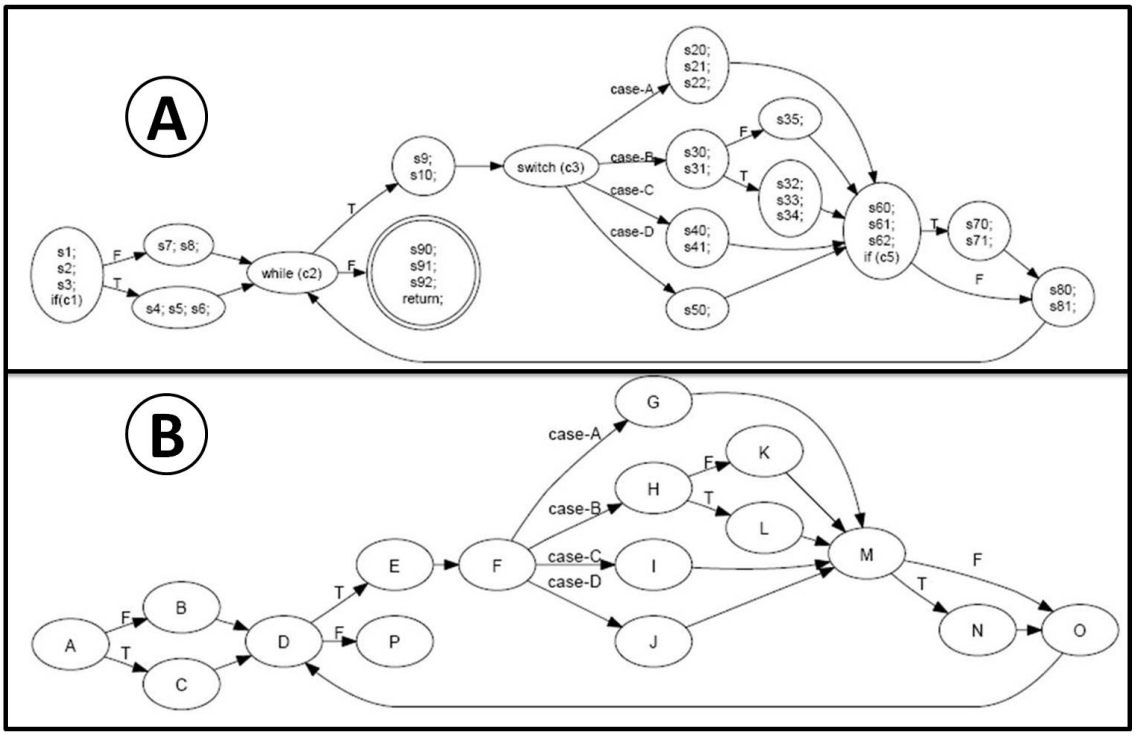


Figura 2.5: *GFC correspondente ao Código evaluate.java*

Na Figura 2.5 é possível verificar duas formas de representação do GFC do programa *evaluate*. A primeira, na parte superior da figura (A), apresenta os trechos de código que cada nó contém. Já a segunda, na parte inferior, consiste somente na representação dos nós. Nesta segunda representação (B), há também elementos que auxiliam na compreensão do fluxo que será seguido no GFC, por exemplo, entre as arestas (A,B) e (A,C) que possui os valores T e F para representar o resultado do teste condicional do *if* que está no nó A. Outro exemplo de representação do direcionamento de fluxo pode ser encontrado nas arestas (F,G), (F,H), (F,I) e (F,J) em que são demarcados os caminhos que serão seguidos para cada comando do tipo *case*.

O primeiro passo para definição dos casos de teste que serão necessários para exercitar os caminhos básicos de um GFC é calcular a complexidade ciclomática (conforme explicado anteriormente nesta seção):

- Baseado na quantidade de arestas e nós (Quantidade de Arestas - Quantidade de Nós + 2): 22 - 16 + 2 = 8; ou

- Baseado na quantidade de nós predicativos (Quantidade de Nós Predicativos + 1): 7 + 1 = 8; ou

- Basedo nas regiões do grafo conforme representado na Figura 2.6;

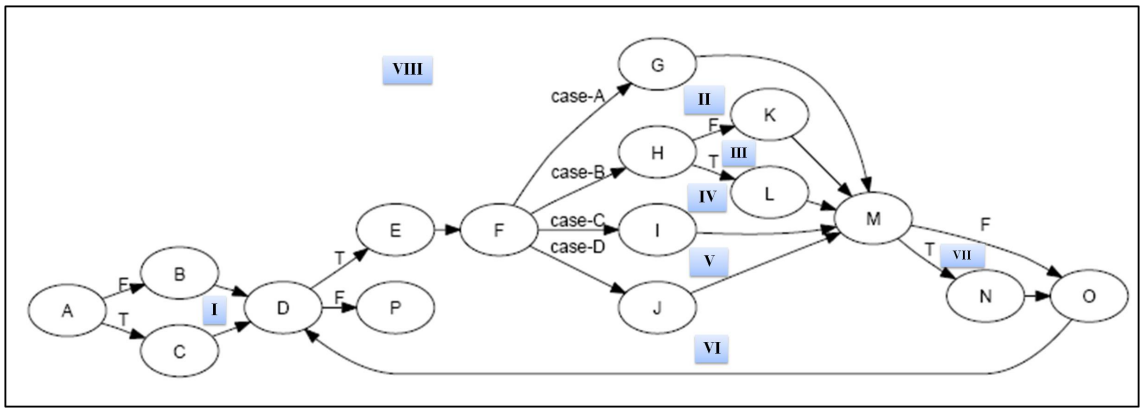


Figura 2.6: *GFC correspondente ao Código evaluate.java com as regiões que representam a complexidade ciclomática*

Como foi apresentado anteriormente, o número de regiões é computado contando-se todas as áreas delimitadas e a área não delimitada fora do grafo conforme representado na Figura 2.6. Sendo assim, após a realização dos cálculos ou da análise da figura, conclui-se o primeiro passo em que foi verificado que a complexidade ciclomática do GFC do programa *evaluate* é 8. Ou seja, serão necessários 8 casos de teste para executar todos os caminhos básicos.

Calculada a complexidade ciclomática, o segundo passo consiste em definir os caminhos linearmente independentes que existem no GFC que, neste exemplo, serão 8. Para iniciar, deve-se escolher um caminho básico que pode ser: caminho mais comum, caminho mais crítico ou caminho mais importante do ponto de vista do teste.

Pelo critério de caminho mais comum, o **primeiro** será (A,B,D,P). Dado que existe uma estrutura de decisão no nó A, o **segundo** caminho será aquele que exercita a outra aresta correspondente à estrutura de decisão, ou seja, (A,C,D,P). Baseado no conhecimento da estrutura do programa *evaluate*, sabe-se que o nó D contém um comando *while* cuja condição de parada determina que o fluxo siga para P, sendo assim, todos os possíveis fluxos que podem existir e que retornam a D devem ser executados. Desse modo, para determinar os próximos caminhos, deve-se escolher as diferentes possibilidades de fluxo que passam pelo nó E (que corresponde ao primeiro nó do *while*). Então, o **terceiro** caminho será (A,B,D,E,F,G,M,O,D,P). O nó F possui um comando *switch*,

assim o **quarto** caminho deve exercitar o *case-B* e será (A,B,D,E,F,H,K,M,O,D,P). No *case-B* ainda há uma estrutura de decisão, desse modo, ela deverá ser exercitada pelo **quinto** caminho (A,B,D,E,F,H,L,M,O,D,P). O **sexto** caminho deverá passar pelo *case-C* formando (A,B,D,E,F,I,M,O,D,P). O **sétimo** exercitará o *case-D* pelo caminho (A,B,D,E,F,J,M,O,D,P). E, por fim, tem-se o **oitavo** caminho, que deverá passar pelo nó N, já que ele é o único que ainda não foi exercitado por nenhum dos caminhos acima, para isso, define-se o caminho (A,B,D,E,F,J,M,N,O,D,P).

Agora que os caminhos já estão definidos, o terceiro passo para determinar os dados de teste que exercitam os caminhos do Código 2.3 consiste na identificação dos valores de entrada que garantam a execução de tais caminhos. A Tabela 2.2 apresenta os valores que devem ser atribuídos em cada estrutura de decisão para que os caminhos básicos sejam executados.

Tabela 2.2: *Casos de teste com valores que devem ser atribuídos em cada estrutura de decisão programa evaluate*

Caminhos	C1	C2	C3	C4	C5
1º	F	F	-	-	-
2º	T	F	-	-	-
3º	F	T	A	-	F
4º	F	T	B	F	F
5º	F	T	B	T	F
6º	F	T	C	-	F
7º	F	T	D	-	F
8º	F	T	D	-	T

Para cada um dos caminhos escolhidos foi definido um caso de teste com seus respectivos valores. Por exemplo, para o dado de teste que está na quinta linha da Tabela 2.2 (F,T,B,F,F) tem-se como resultado esperado que o 4º caminho seja executado.

O exemplo apresentado para o programa *evaluate* ilustra como a complexidade ciclomática pode auxiliar na identificação dos esforços de teste e o atendimento dos critérios de fluxo de controle. Mesmo considerando as desvantagens do teste estrutural e a impossibilidade de satisfação do critério Todos-Caminhos para a maioria dos programas, considera-se esse tipo de teste de fundamental importância para garantir que partes essenciais ou críticas do software tenham sido exercitadas durante os testes.

Critérios baseados em fluxo de dados

Os critérios baseados neste tipo de fluxo utilizam a análise do fluxo de dados como fonte de informação para derivar os requisitos de teste. Tais critérios baseiam-se nas definições, nas associações entre a definição de uma variável e seus possíveis usos

subsequentes dentro do programa (BARBOSA et al., 2007). No contexto de fluxo de dados, a ocorrência de variáveis em um programa pode ser classificada em:

- **Definição** (*def*): ocorre quando uma variável recebe um valor como, por exemplo: length = 0 (linha 5 do Código 2.1);

- **Uso Computacional** (*c-uso*): ocorre quando a variável é utilizada em uma computação como, por exemplo: length++ (linha 21 do Código 2.1);

- **Uso Predicativo** (*p-uso*): ocorre quando a variável é utilizada em uma condição como, por exemplo: achar != '\n' (linha 15 do Código 2.1).

A partir das definições e usos de variáveis de um determinado código é possível criar um Grafo Def-Uso para ele. O Grafo Def-Uso é uma extensão do GFC. Nele são adicionadas informações a respeito do fluxo de dados, caracterizando associações entre pontos do programa nos quais é atribuído um valor a uma variável e pontos nos quais esse valor é utilizado. A partir de tais informações, os requisitos de teste são determinados (BARBOSA et al., 2007). A Figura 2.7 representa o Grafo Def-Uso correspondente à função principal do programa *identifier* (Código 2.1).

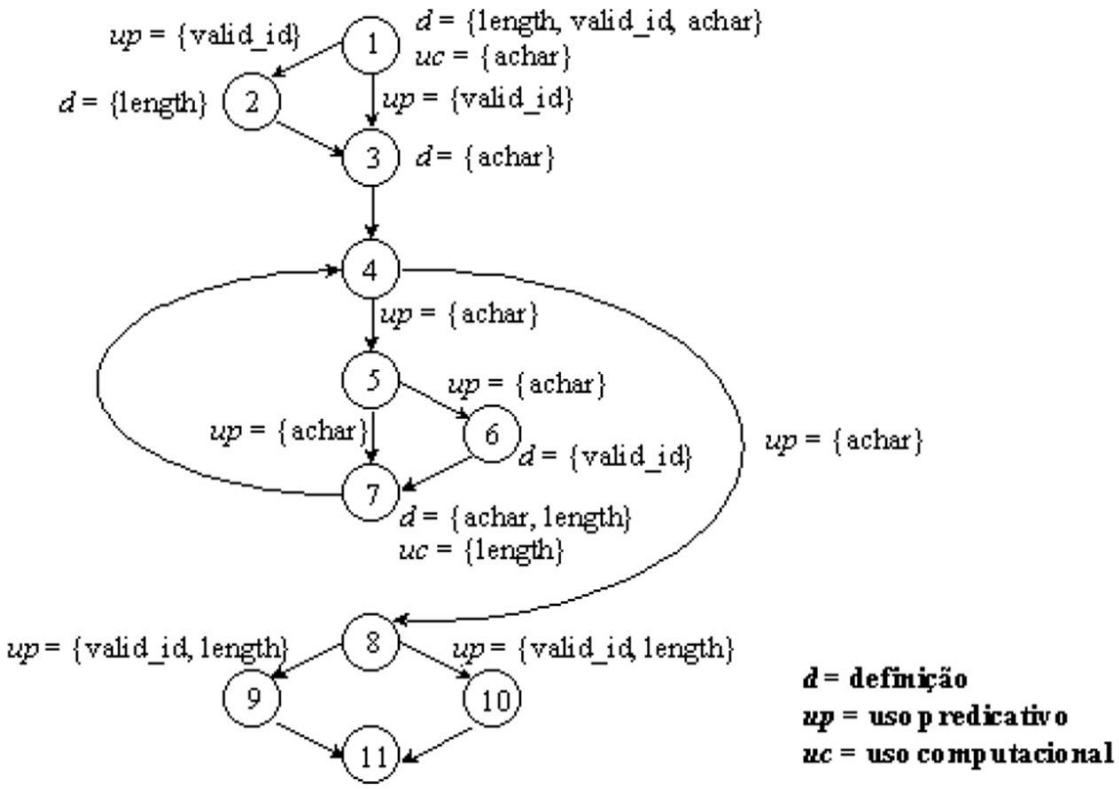


Figura 2.7: *Grafo Def-Uso correspondente à função principal do programa identifier (extraído de Barbosa et al. (2007))*

Na Figura 2.7 em que é apresentado o Grafo Def-Uso da função principal do programa *identifier* pode-se notar que a estrutura dos nós e arestas é a mesma do GFC (Figura 2.3), no entanto, há outras informações no grafo que denotam a definição e o uso das variáveis. Conforme apresentado na legenda que está no canto inferior direito da Figura 2.7, as variáveis marcadas por 'd' são ocorrências de definição, aquelas marcadas com 'up' são referentes ao uso predicativo e as marcadas com 'uc' referem-se ao uso computacional.

Outro conceito que pode ser analisado a partir de um Grafo Def-Uso é o caminho livre de definição. Ele consiste em um caminho que, dada uma definição de variável em um nó A, não pode existir nenhuma outra definição dela entre A e B (BARBOSA et al., 2007). No exemplo da Figura 2.7, pode-se dizer que há um caminho livre de definição para variável 'achar' entre os nós 3 e 6, pois ela é definida no nó 3 e não há outra definição para ela até o nó 6.

A partir dos conceitos apresentados, pode-se definir critérios baseados em fluxo de controle. Dois deles são (RAPPS; WEYUKER, 1982):

- **Todas-Definições** (*all-defs*): é conhecido como o critério mais básico dentre os baseados em análise de fluxo de dados. Ele requer que cada definição de variável seja exercitada pelo menos uma vez, não importando se por um c-uso ou por um p-uso (RAPPS; WEYUKER, 1982);

- **Todos-Usos** (*all-uses*): é o critério que tem sido um dos mais utilizados e investigados dentre os baseados em fluxo de dados. Ele requer que todas as associações entre uma definição de variável e seus subsequentes usos (c-usos e p-usos) sejam exercitadas pelos casos de teste por meio de pelo menos um caminho livre de definição (RAPPS; WEYUKER, 1982).

Partindo do exemplo do Grafo-Def Uso da função principal do programa *identifier* (Código 2.1) seria possível definir caminhos do grafo para atender os critérios Todas-Definições e associações que atenderiam o critério Todos-Usos.

Os sub-caminhos do critério Todas-Definições são identificados da seguinte forma: para cada definição da variável, deve-se listar todos os sub-caminhos livres de definição que podem existir a partir dela, por exemplo, para variável *length* que é definida no nó 1, tem-se (1,3,4,5,7), (1,3,4,5,6,7), (1,3,4,8,9) e (1,3,4,8,10). Após esta identificação, basta exercitar um dos sub-caminhos (que seja executável) de cada definição de variável para que o critério seja satisfeito. A lista dos elementos requeridos pelo critério Todas-Definições para o Grafo Def-Uso da Figura 2.7 pode ser encontrado na Tabela 2.3.

Já para atender ao critério Todos-Usos são formadas associações conforme o seguinte: [i, j, variável] e [i, (j,k), {variável}] que indicam que variável é definida no

Tabela 2.3: *Elementos requiridos pelo critério Todas-Definições para o Grafo Def-Uso da Figura 2.7 (exemplo adaptado de (BARBOSA et al., 2007))*

Nó	Variável	Possíveis Caminhos
1	length	(1,3,4,5,7)
		(1,3,4,5,6,7)
		(1,3,4,8,9)
		(1,3,4,8,10)
1	valid_id	(1,2,3,4,8,9)
		(1,3,4,8,10)
1	achar	(1,3,4,5,7)
		(1,3,4,5,6)
		(1,3,4,8)
2	length	(2,3,4,5,7)
		(2,3,4,5,6,7)
		(2,3,4,8,9)
		(2,3,4,8,10)
3	achar	(3,4,5,7)
		(3,4,5,6)
		(3,4,8)
6	valid_id	(6,7,4,8,9)
		(6,7,4,8,10)
7	length	(7,4,5,7)
		(7,4,5,6,7)
		(7,4,8,9)
		(7,4,8,10)
7	achar	(7,4,5,7)
		(7,4,5,6)
		(7,4,8)

nó *i* e existe um uso computacional dela no nó *j* ou um uso predicativo no arco (*j,k*), respectivamente, bem como pelo menos um caminho livre de definição do nó *i* ao nó *j* ou arco (*j,k*) (BARBOSA et al., 2007). Tomando novamente como exemplo a variável *length* que é definida no nó 1, serão requeridas as seguintes associações: [1, 7, {length}], [1, (8,9), {length}] e [1, (8,10), {length}]. A lista dos elementos requeridos pelo critério Todos-Usos para o Grafo Def-Uso da Figura 2.7 pode ser encontrado na Tabela 2.4.

A partir dos dados apresentados nas Tabelas 2.3 e 2.4 o testador poderá definir os casos de teste para satisfazer cada um dos critérios. No caso do critério Todas-Definições, a seleção de um sub-caminho (dentre os apresentados na Tabela 2.3) para cada definição caracteriza o conjunto de elementos requeridos pelo critério. Já para o critério Todos-Usos deve ser realizada uma análise para escolher qualquer sub-caminho que satisfaça cada uma das associações da Tabela 2.4 como, por exemplo, para [1, 7, {length}] o testador pode escolher qualquer caminho completo que inclua um dos sub-caminhos (1,3,4,5,6,7) ou

Tabela 2.4: *Elementos requiridos pelo critério Todos-Usos para o Grafo Def-Uso da Figura 2.7 (exemplo adaptado de (BARBOSA et al., 2007))*

Associações requeridas		Associações requeridas
[1, 7, {length}]		[3, (4,5), {achar}]
[1, (8,9), {length, valid_id}]		[3, (4,8), {achar}]
[1, (8,10), {length, valid_id}]		[3, (5,6), {achar}]
[1, (1,2), {valid_id}]		[3, (5,7), {achar}]
[1, (1,3), {valid_id}]		[6, (8,9), {valid_id}]
[1, 1, {achar}]		[6, (8,10), {valid_id}]
[1, (4,5), {achar}]		[7, 7, {length}]
[1, (4,8), {achar}]		[7, (8,9), {length}]
[1, (5,6), {achar}]		[7, (8,10), {length}]
[1, (5,7), {achar}]		[7, (4,5), {achar}]
[2, 7, {length}]		[7, (4,8), {achar}]
[2, (8,9), {length}]		[7, (5,6), {achar}]
[2, (8,10), {length}]		[7, (5,7), {achar}]

(1,3,4,5,7) (BARBOSA et al., 2007).

Nas análises para definição dos casos de teste, o testador deve verificar aqueles caminhos que não são executáveis de acordo com a lógica do programa, pois não seria possível construir casos de testes que executem caminhos completos que possuam esses sub-caminhos não executáveis.

Por exemplo, o caminho (1,3,4,8,9) é dito não executável, pois analisando o Código 2.1 nota-se que, para que haja um desvio de fluxo do nó 1 para o 3, a variável *valid_id* tem que ser avaliada como '0' no comando *if(valid_id)* do nó 1 (linha 10) e a variável *length* permanecer com seu valor inicial '0'. Além disso, para que haja um desvio de execução do nó 4 para o nó 8, os comandos internos à estrutura *while (achar != '\n')* que está no nó 4 (linha 15) não podem ser executados. Nesse caso, a variável *achar* deve receber um *'\n'* no nó 3 (linha 14), no entanto, como os valores de *valid_id* e *length* são iguais a '0', o comando condicional no nó 8 (linha 24) é avaliado como falso e, consequentemente, não ocorre desvio do nó 8 para o nó 9.

Assim, qualquer caminho completo que inclua o sub-caminho não executável também será não executável e não será possível construir um caso de teste que o exercite (BARBOSA et al., 2007).

2.2.3 Níveis de cobertura de Código

Cobertura de código consiste na porcentagem dos requisitos que foram testados 'versus' o total de requisitos gerados (COPELAND, 2004). A partir deste conceito, Copeland (2004) definiu 8 níveis de cobertura em função dos elementos do GFC de modo que,

quanto maior o nível, maior o rigor do critério de teste. A Figura 2.8 ilustra a relação dos níveis e uma breve descrição de cada um deles.

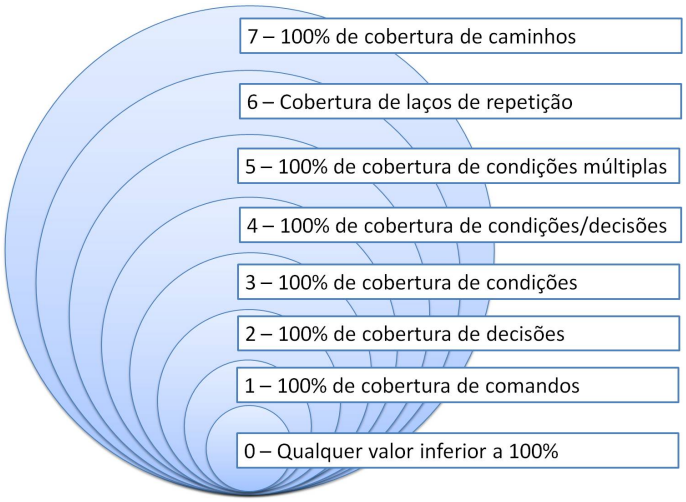


Figura 2.8: *Níveis de cobertura segundo Copeland (COPELAND, 2004)*

Na Figura 2.8 pode-se notar que o **Nível 0** não representa nenhum critério de teste, pois ele representa qualquer valor de cobertura de comandos inferior a 100%, ou seja, qualquer caso de teste que exercite ao menos o primeiro nó de um GFC será considerado como Nível 0.

Já o **Nível 1** está relacionado ao critério Todos-Nós, pois exige a cobertura de todos os comandos (conforme Figura 2.8). O trecho de Código 2.4 ilustra um exemplo para avaliação dos níveis 1 e 2 de cobertura.

Código 2.4 Código para exemplos dos níveis 1 e 2 de cobertura

```
1  if(a > 0){
2        x = x + 1;
3  }
4  if(b == 3){
5        y = 0;
6  }
```

O Código 2.4 possui dois condicionais do tipo *if* (linhas 1 e 4), um comando de computação na linha 2 e um comando de atribuição na linha 5. O GFC correspondente a esse código está representado na Figura 2.9.

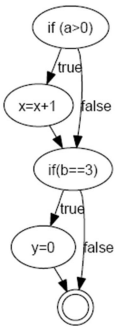


Figura 2.9: *GFC que representa o Código 2.4*

O GFC da Figura 2.9 possui 4 nós que devem ser cobertos para se atingir o Nível 1 de cobertura. Com apenas um caso de teste seria possível atingir este Nível. Por exemplo, para os valores $a = 6$ e $a = 3$ todos os comandos serão cobertos.

O **Nível 2**, conforme apresentado na Figura 2.8 está relacionado à 100% de cobertura das decisões, ou seja, o critério Todas-Arestas. A 2.10 representa os diferentes caminhos que podem existir para o GFC que representa o Código 2.4.

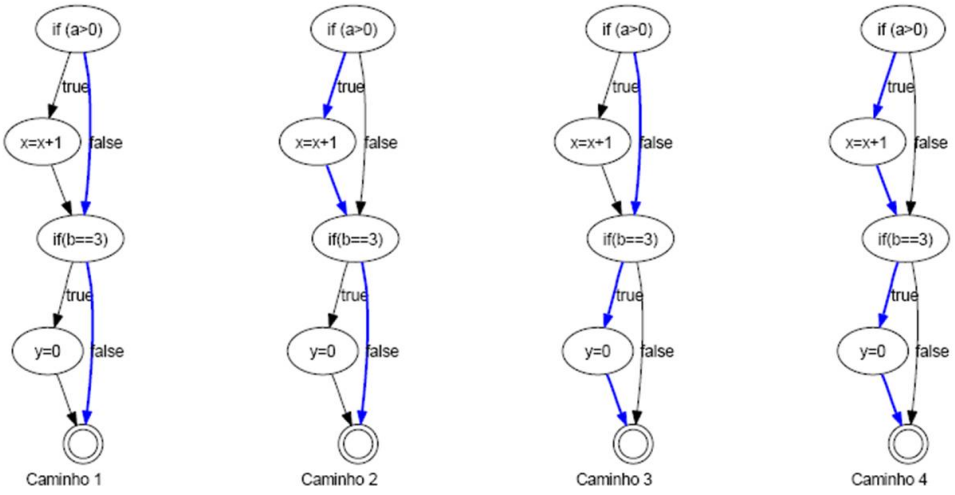


Figura 2.10: *Diferentes caminhos possíveis para o GFC que representa o Código 2.4*

Para atingir o Nível 2 de cobertura, o objetivo dos casos de teste é fazer com que cada comando de decisão assuma os valores *true* e *false*. A Figura 2.10 apresenta 4 caminhos possíveis que são criados alternando os valores desses comandos de decisão, no entanto, o Nível 2 exige somente o critério Todas-Arestas (ou 100% da cobertura de decisões) e, para isso, não é necessário exercitar todos os 4 caminhos apresentados. Por exemplo, com apenas 2 casos de teste com valores ($a = -2$, $b = 2$) e ($a = 4$, $b = 3$), os caminhos 1 e 4 da Figura 2.10 serão exercitados e o Nível 2 será atingido.

No **Nível 3** a exigência está em atender 100% das condições (conforme apresentado na Figura 2.8). O trecho de Código 2.5 ilustra um exemplo para avaliação do Nível 3 de cobertura.

Código 2.5 Código para exemplos dos Nível 3 de cobertura

```
1 if(a > 0 & c == 1){
2       x = x + 1;
3 }
4 if(b == 3 | d < 0){
5       y = 0;
6 }
```

No exemplo do Código 2.5 os comandos de decisão (linhas 1 e 4) possuem uma condição a mais para cada um deles em relação ao Código 2.4. Neste caso, para que o comando da linha 2 seja executado é necessário que as duas condições da linha 1 ($a > 0$ e $c == 1$) tenham valor 'true'. Já o comando da linha 5, para ser executado, necessita somente que uma ou outra condição da linha 4 ($b == 3$ ou $d < 0$) tenha valor 'true'. Para atender a este Nível, ou seja, exercitar todas as condições, dois casos de teste seriam suficientes, por exemplo: ($a = 2$, $c = 1$, $b = 3$, $d = -1$) e ($a = 0$, $c = 2$, $b = 4$, $d = 1$). Ressalta-se que para ilustrar o atendimento ao Nível 3 de Copeland (2004) foram utilizadas condições que não utilizam avaliação de curto-circuito "forçando" a execução de cada um dos condicionais das linhas 1 e 4 do Código 2.5.

O **Nível 4**, conforme apresentado na Figura 2.8, consiste na cobertura de 100% das decisões e condições. O Nível 4 requer que todas as combinações de uma decisão sejam testadas. O trecho de código 2.6 apresenta um exemplo para análise do Nível 4 de cobertura.

Código 2.6 Código para análise do Nível 4 de cobertura

```
1 if(x && y){
2       facaAlgo;
3 }
```

A partir do exemplo do código 2.6 pode-se verificar que com dois casos de teste ($x = true$, $y = false$) e ($x = false$, $y = true$) é possível atingir o Nível 3 (100% de cobertura das condições), no entanto, nenhum desses casos de teste iriam executar o código da linha 2 (*facaAlgo;*), por isso, o Nível 4 requer que todas as combinações de uma decisão sejam testadas para ser atingido.

O **Nível 5** da Figura 2.8 (100% de cobertura de condições múltiplas) consiste em utilizar o conhecimento de como o compilador avalia as condições múltiplas de

determinado comando de decisão e utilizar essa informação na geração de casos de testes. A Figura 2.11 ilustra um exemplo de decisões múltiplas de um determinado compilador.

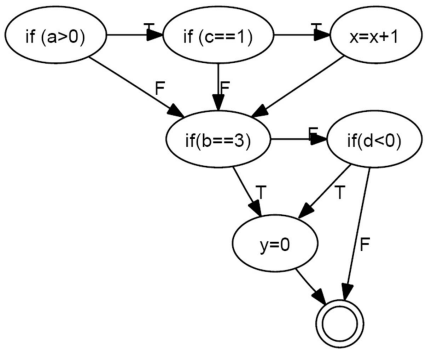


Figura 2.11: *Grafo das condições múltiplas do código 2.5*

Para composição do grafo das condições múltiplas do código 2.5 (Figura 2.11), cada condição das estruturas de decisão compõe um nó independente. Assim, são criadas arestas individuais para valores 'true' e 'false' de cada uma dessas condições. A Figura 2.12 apresenta 4 casos de teste com seus respectivos caminhos para ilustrar como Nível 5 pode ser alcançado.

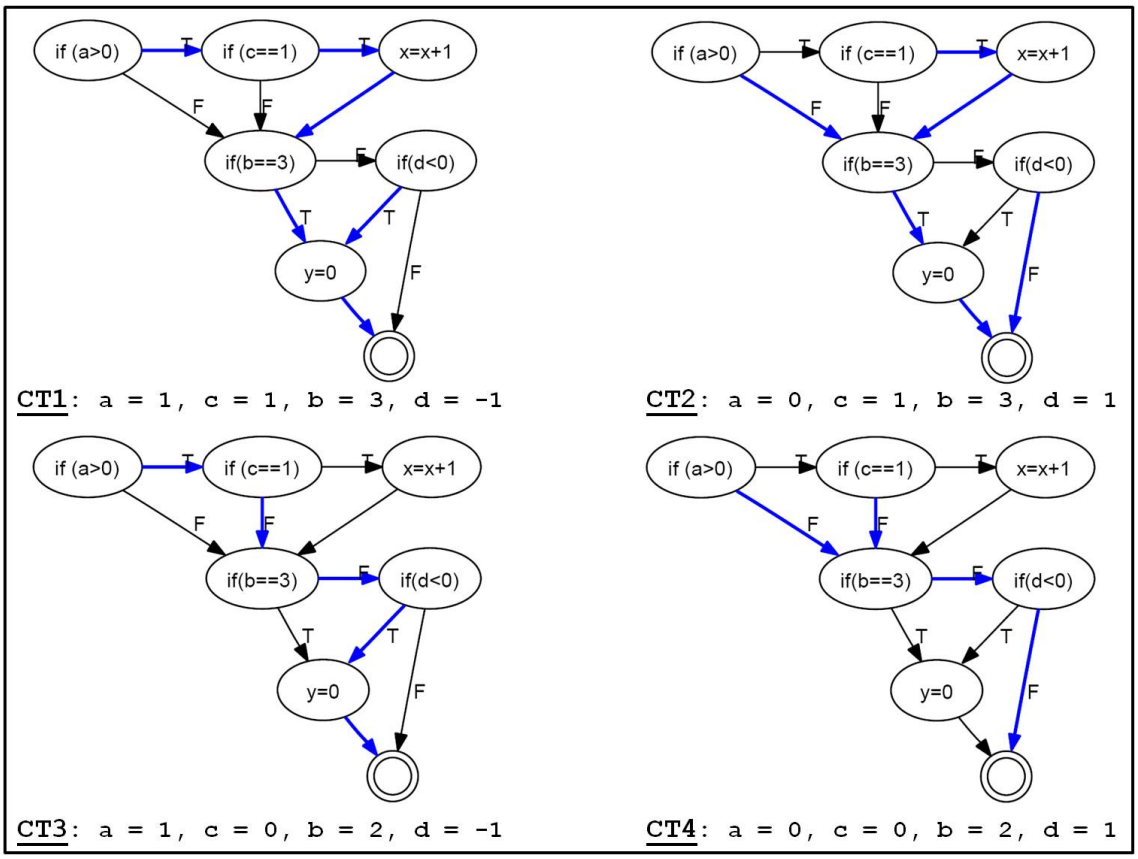


Figura 2.12: *Casos de teste e caminhos do grafo das condições múltiplas do código 2.5*

Os 4 casos de teste apresentados na Figura 2.12 tem o intuito de executar todas as combinações das decisões para alcançar o Nível 5. Pode-se considerar que o Nível 5 tem a mesma exigência do Nível 4. A diferença entre os dois esta no fato de que, para alcançar o Nível 5, utiliza-se informações sobre como o compilador avalia as condições múltiplas. Na Figura 2.12, o caso de teste 1 (CT1) apresenta o caminho em que as combinações $a > 0$, $c == 1$, $b == 3$ e $d < 0$ são iguais a 'true'. Já o caso de teste 2 (CT2) representa o caminho em que $a > 0$ e $d < 0$ são 'false' e $c == 1$, $b == 3$ são 'true'. No caso de teste 3 (CT3) esta o caminho para $a > 0$ e $d < 0$ igual a 'true' e $c == 1$, $b == 3$ igual a 'false'. E, por fim, o caso de teste 4 (CT4) em que $a > 0$, $c == 1$, $b == 3$ e $d < 0$ são 'false'. Ressalta-se que obter 100% da cobertura de condições múltiplas implica cobrir 100% dos critérios anteriores, mas não garante cobertura de Todos-Caminhos.

O **Nível 6** (Figura 2.8) consiste na cobertura de laços de repetição. Para alcançá-lo, deve-se testar os laços de repetição do programa. No entanto, quando programas possuem laços de repetição, o número de caminhos a ser percorrido pode ser infactível. O trecho de código 2.7 ilustra como os laços de repetição podem aumentar a quantidade de caminhos.

Código 2.7 Aumento da quantidade de caminhos

```
1  for(i = 1; i <= 1000; i++){
2          for(j = 1; j <= 1000; j++){
3                  for(k = 1; k <= 1000; k++){
4                          facaAlgoCom(i, j, k);
5                  }
6          }
7  }
```

O exemplo do código 2.7 ilustra como o teste exaustivo de todos os caminhos para laços de repetição pode não ser uma alternativa viável. No exemplo apresentado, como há 3 laços de repetição aninhados, o comando *facaAlgoCom(i, j, k)* (linha 4) seria executado 1 bilhão de vezes (1000 x 1000 x 1000), pois cada laço multiplica o número de caminhos do fluxo de controle. Neste caso, sugere-se uma redução do número de caminhos limitando a execução do laço a, por exemplo: 0 vezes, 1 vez, 2 vezes, 'n' vezes (sendo n um número de vezes padrão que o laço é executado) ou 'm' vezes (sendo m o número máximo de vezes que o laço pode ser executado).

O **Nível 7**, descrito na Figura 2.8 como 100% de cobertura de caminhos, corresponde ao critério Todos-Caminhos. Neste caso, para programas sem laços de repetição o número de caminhos pode ser pequeno o suficiente e casos de testes podem ser construídos para cobri-los. No entanto, para programas com muitos laços ou com laços aninhados (como no código 2.7) o número de caminhos pode ser muito grande ou infinito,

tornando-se impossível cobrir todos os caminhos de execução (conforme apresentado na Seção 2.2.2).

A Seção 2.2.4 apresenta ferramentas que podem auxiliar na execução dos testes estruturais, bem como ferramentas de apoio ao processo de análise de cobertura.

2.2.4 Ferramentas automatizadas para apoio ao teste estrutural

A análise de cobertura de critérios para programas com estruturas lógicas consideradas complexas pode não ser uma atividade trivial (conforme discutido na Seção 2.2.2). Sendo assim, a aplicação de critérios de teste estrutural sem o apoio de ferramentas automatizadas tende a ser uma atividade propensa a erros e limitada a programas muito simples (BARBOSA et al., 2007).

As ferramentas de teste de cobertura de código apresentam ao testador detalhes do código que está sendo exercitado (ou mais importante, códigos que não estão sendo exercitados), durante a execução da aplicação (BERG, 2007). Em Barbosa et al. (2007) podem ser encontradas algumas ferramentas de apoio ao teste estrutural, dentre elas:

- RXVP: ferramenta comercial que realiza basicamente a análise de cobertura de teste de arestas em programas escritos na linguagem Fortran;

- TCAT (*Test-Coverage Analysis Tool*) (Software Research, Inc., 2010): ferramenta comercial que realiza teste de unidade segundo o critério Teste de Ramos Lógicos (arestas) e está disponível para análise de códicos nas linguagens C, C++ e Java. Teste de ramos lógicos divide os predicados encontrados em uma condição em vários *"if's"*, no qual cada *if* só contém um dos predicados (BARBOSA et al., 2007);

- SCORE: ferramenta de código proprietário e que apóia o teste de arestas de programas escritos na linguagem Pascal;

- Asset (*A System to Select and Evaluate Tests*) (FRANKL; WEYUKER, 1985): desenvolvida na Universidade de Nova Iorque e que apóia os critérios de fluxo de dados no teste de programas escritos na linguagem Pascal;

- POKE-TOOL (*Potential Uses Criteria Tool for Program Testing*) (CHAIM, 1991): apóia a aplicação dos critérios Potenciais-Usos (MALDONADO, 1991) e também de outros critérios estruturais baseados em fluxo de controle e fluxo de dados e pode ser utilizada para testes em programas escritos na linguagem C e Cobol;

- JaBUTi (*Java Bytecode Understanding and Testing*) (VINCENZI et al., 2010): ferramenta de código aberto que fornece suporte aos diferentes critérios de teste estruturais apresentados na Seção 2.2, além de um conjunto de métricas estáticas para avaliar classes que compõem programas escritos na linguagem Java.

O primeiro passo para realização deste trabalho foi a escolha de uma ferramenta automatizada para auxiliar no processo de análise dos experimentos. Na Seção 3.1 pode-se encontrar detalhes sobre a condução das análises, bem como explicações sobre a utilização de uma ferramenta desse gênero dentro do processo de experimentação.

Para escolha da ferramenta a ser adotada neste trabalho, o critério levado em consideração foi o suporte à linguagem Java decorrente das necessidades apresentadas na Seção 2.4.3. No início do trabalho optou-se pela ferramenta JaBUTi. No entanto, durante a realização dos primeiros experimentos foi detectado que a ferramenta não apresentava um desempenho satisfatório para o processo de instrumentação de código em projetos com "grande" quantidade de classes. Processo este, que, conforme descrito na Seção 3.1, consiste em uma parte fundamental para realização da coleta dos dados de cobertura.

Desse modo, concluiu-se que, para cada projeto Java que deveria ser testado, uma série de adequações na ferramenta JaBUTi deveria ser realizada. Além disso, verificou-se que tal processo de adequação estava sendo dispendioso em relação ao tempo disponível para realização deste trabalho. Sendo assim, optou-se por pesquisar a possibilidade de utilização de outra ferramenta para condução dos experimentos. Neste contexto, optou-se pela ferramenta Emma (ROUBTSOV, 2010).

EMMA

A Emma consiste em uma ferramenta de código aberto que possibilita medir e analisar a cobertura de códigos escritos na linguagem Java. Ela auxilia o desenvolvedor ou o testador na atividade de avaliação de quais partes do código estão sendo exercitadas pela atividade de teste por meio de relatórios que indicam quais áreas do código que foram cobertas (ROUBTSOV, 2010).

A utilização da ferramenta Emma possibilita cobertura de classes, métodos, linhas e blocos básicos (ou simplesmente blocos). A unidade básica da Emma é o bloco, pois, a partir desta, todas as outras são derivadas. A utilização de blocos de *bytecode* possibilita a obtenção dos dados de cobertura sem a necessidade do código fonte (ROUBTSOV, 2010).

Um bloco consiste em uma sequência de instruções de *bytecode* sem *jumps* que é executado como uma unidade atômica (na ausência de exceções). Quando existe uma exceção que não foi tratada pelo desenvolvedor dentro de um bloco *try/catch*, a Emma "marca" a cobertura até a última instrução lida. Quando a exceção é tratada em *try/catch*,

a Emma cria os blocos para cada estrutura e verifica a cobertura normalmente. Assim, um bloco pode ser considerado coberto quando o controle alcança sua última instrução, ou seja, um bloco coberto é considerado aquele que foi executado sem falhas ao menos uma vez (ROUBTSOV, 2010).

As medidas de cobertura de linhas, métodos e classes, conforme apresentado acima, são derivadas dos blocos. No caso das linhas, a Emma verifica como cada linha é mapeada para a estrutura de blocos, assim se todos os blocos da linha forem cobertos, considera-se que a linha foi 100% coberta, caso contrário, tem-se uma fração desse valor de acordo com a quantidade de blocos cobertos (ROUBTSOV, 2010). Observa-se que, no caso dos experimentos conduzidos durante este trabalho, vide Seção 3.1, o mapeamento de cobertura de blocos para linhas não foi realizado adequadamente pela ferramenta EMMA em alguns projetos. Desse modo, nos dados apresentados nas Seções 3.1.3 e 3.2, a informação sobre a cobertura de linhas foi omitida dos relatórios.

Ainda segundo Roubtsov (2010), uma classe executável é indicada como tendo sido coberta se tiver sido carregada e inicializada pela *Java Virtual Machine* (JVM) e um método será considerado coberto quando o processamento tiver entrado nele, ou seja, se seu primeiro bloco tiver sido coberto. A Figura 2.13 ilustra um exemplo de como a Emma computa a cobertura de um código.

```
 1 public class MyClass
 2 {
 3     public static void main (final String [] args)
 4     {
 5         int vi = 1;
 6         int vj = vi > 0 ? -1 : 1;
 7
 8         for (int vk = 0; vk < vj; ++ vk)
 9         {
10             System.out.println ("vk = " + vk);
11         }
12     }
13
14     public MyClass () {}
15 }
```

Figura 2.13: *Exemplo de cobertura de código utilizando a Emma (ROUBTSOV, 2010)*

A Figura 2.13 consiste na representação de uma classe Java denominada *MyClass* (linha 1). Ela é composta por um método denominado *main* (linha 3) e um construtor que está declarado na linha 14. Algumas linhas são marcadas por cores diferentes que representam a cobertura obtida pela Emma. A linha 5 é marcada na cor verde indicando cobertura de 100% dela. As linhas 6 e 8 estão em amarelo, pois apenas parte delas são cobertas. No caso da linha 6, apenas um ramo do condicional é executado. E na linha 8, a variável 'vk' nunca é incrementada, já que, segundo a lógica do programa, a condição '$vk < vj$' não é satisfeita nem no primeiro teste do laço de repetição. Por fim, as linhas 10

e 14 são destacadas em vermelho indicando que não há nenhuma cobertura. No caso da linha 10, ela não é executada decorrente do critério da linha 8 não ter sido satisfeito e a linha 14 não é executada, pois o corpo do método está vazio.

Segundo as definições de Roubtsov (2010), pode-se concluir que o método *main* (declarado na linha 3) foi considerado coberto, pois o processamento "entrou" nele e a classe *MyClass* (declarada na linha 1) também foi coberta, pois foi carregada pela JVM. A Emma pode ser utilizada basicamente de duas formas (ROUBTSOV, 2010):

- **Instrumentação *on-the-fly*** que é utilizada para coletar dados de cobertura enquanto a aplicação está sendo utilizada. Este modo de instrumentação pode ser utilizado por testadores que queiram, por exemplo, avaliar a cobertura de um conjunto de testes funcionais criando-se um programa executável que coleta dados de cobertura à medida que ele é utilizado;

- **Instrumentação *offline*** que possibilita a instrumentação, execução dos casos de teste e geração de relatórios em fases separadamente.

A utilização da Emma no modo de instrumentação *offline* foi o escolhido para realização dos experimentos deste trabalho, pois os conjuntos de teste analisados foram testes automatizados desenvolvidos utilizando o JUnit (JUNIT, 2010) e não foi necessário utilizar o modo de instrumentação *on-the-fly*. Na Seção 3.1.1 é descrito o processo geral para realização dos experimentos com a Emma e na Seção 3.1.3 podem ser encontrados os comandos da Emma que foram utilizados no processo de instrumentação, execução dos casos de teste e geração dos relatórios.

Para as duas formas de instrumentação da Emma é possível utilizá-la em conjunto com o Ant (Apache Foundation, 2010e). O Ant consiste em uma ferramenta que auxilia na "construção" de projetos Java. Ela utiliza um arquivo em formato XML (W3C, 2010) para descrever o processo de construção e suas dependências (Apache Foundation, 2010e).

Os projetos Java que são construídos utilizando o Ant possuem um arquivo denominado *build.xml*, localizado geralmente na pasta raiz do projeto, com comandos que possibilitam, por exemplo, compilar os códigos fonte, compilar os códigos de teste e executar o conjunto de testes. Na Seção 3.1.3 são apresentadas as alterações realizadas no arquivo *build.xml* dos projetos para coletar os dados de cobertura, bem como os comandos do Ant que foram utilizados na condução de cada um dos experimentos.

A Seção 2.3 apresenta algumas considerações sobre o processo de teste adotado em comunidades FLOSS.

2.3 Teste de Software FLOSS

Esta seção apresenta o significado do conceito de *Free / Libre / Open Source Software* (FLOSS) e suas características e exemplifica o processo de desenvolvimento e de testes utilizado por comunidades de FLOSS visando a ilustrar os impactos das características desse tipo de produto no ciclo de vida do software.

2.3.1 FLOSS

O conceito de Software Livre começou a ser utilizado em 1985 com a criação da *Free Software Foundation* (FSF) (FALCãO et al., 2005). A FSF é uma organização sem fins lucrativos com uma missão global de promover a liberdade de usuários de computadores e de defender os direitos de todos os usuários de software livre (FSF, 2010). O movimento iniciado pela FSF ganhou mais força em 1991 com o desenvolvimento do GNU/Linux e o lançamento da GNU GPL (GNU *General Public License* ou Licença Pública do GNU), instrumento jurídico pelo qual o software pode ser considerado livre (FALCãO et al., 2005).

A GPL entende por software livre aquele em que o autor permite aos seus usuários quatro direitos ou liberdades: (a) a liberdade de executar o programa a qualquer propósito; (b) a liberdade para estudar o programa e adaptá-lo às suas necessidades; (c) a liberdade de distribuir cópias de modo que auxilie a terceiros; e (d) a liberdade de aperfeiçoar o programa e divulgar para o público. Tais propósitos ajudam a evitar a propagação de programas ditos "fechados", pois quando o código fonte é suprimido de um programa de computador, são suprimidos também, além do código, dois outros importantes elementos: o conhecimento em torno do programa; e a possibilidade de inovação a partir daquele programa (FALCãO et al., 2005).

Já o conceito de *Open Source Software* (OSS) ou Software de Código Aberto (que pode ser encontrada na íntegra em (OSI, 2010)), segundo (STALLMAN, 2010) foi derivado indiretamente dos critérios de software livre e ambas definições estão de acordo na maioria das características.

Além desses termos, outros tem surgido gerando uma série de interpretações e definições diferentes para um conceito bastante semelhante, dentre eles: "software de domínio público" (*public domain software*), "*Copylefted software*", "*semi-free software*" e "*non-copylefted software*" (FALCãO et al., 2005). Sendo assim, ressalta-se que neste trabalho, os termos "Software de Código Aberto", "Software Livre", *Free / Libre / Open Source Software* são utilizados indistintamente, embora a expressão "Open Source" possa ter significado diferente em outro contexto.

A Seção 2.3.2 apresenta algumas características do processo de desenvolvimento de Software utilizado por comunidades que desenvolvem FLOSS com intuito de contextualizar os impactos desse modelo no processo de teste.

2.3.2 Desenvolvimento de Software em comunidades FLOSS

Na Seção 2.1 foi apresentado que o teste de software pode ser considerado como umas das atividades de garantia da qualidade de software. Além disso, citou-se que o processo de teste deve ser considerado como parte integrante do ciclo de vida do desenvolvimento de software e deve seguir algumas atividades básicas, como: planejamento e organização do ambiente de teste; geração de casos de teste; execução dos testes; avaliação dos resultados; comunicação e armazenamento de problemas encontrados; e rastreamento de defeitos (*defect tracking*). No entanto, um estudo realizado por Zhao e Elbaum (2003), mostrou que 80% dos desenvolvedores de FLOSS entrevistados responderam que seus projetos não tem plano de teste.

Um dos agravantes para este fato é que as atividades de análise e projeto de produtos FLOSS não são bem planejadas pelas comunidades de desenvolvimento, seja devido à visão de curto prazo e não comercial que caracteriza muitos projetos FLOSS, ou pelo fato de muitos terem sido iniciados para resolver problemas particulares de um usuário sem uma visão de longo prazo e uma percepção real da inovação e grau de evolução que o projeto poderia ter no futuro (MORASCA et al., 2009).

A "marca registrada" do desenvolvimento de FLOSS está na sua forma colaborativa e distribuída em que o Software é construído por diferentes equipes compostas por desenvolvedores apaixonados que trabalham em uma grande comunidade virtual (MORASCA et al., 2009).

O desenvolvimento de forma colaborativa e distribuída, que é uma das características marcantes das comunidades que desenvolvem FLOSS, também pode ser encontrado nos mais variados tipos de organizações (como: multinacionais, pequenas empresas, órgãos governamentais, etc.) que desenvolvem software. Em ambos os casos, equipes geograficamente separadas com centenas (ou milhares) de pessoas interagem para construção de um produto. No entanto, um fato que surpreende nas comunidades FLOSS está na não existência de pessoas ou atividades exclusivas para a coordenação do desenvolvimento de seus produtos (MOCKUS et al., 2000).

Mockus et al. (2000) afirmam que o desenvolvimento de FLOSS é radicalmente diferente da indústria de código "fechado", pois, no desenvolvimento de FLOSS, o trabalho é caracterizado pelo seguinte: as atividades não são atribuídas e as pessoas se comprometem ao trabalho que optaram por realizar; não existe um sistema explícito e nem um plano de projeto; e não há um cronograma ou uma lista de produtos a serem construídos.

Geralmente o controle que existe é apenas sobre o produto, em que uma pessoa ou Organização seleciona um subconjunto de funcionalidades e códigos para disponibilizar em versões oficiais (MOCKUS et al., 2000). Mas pode-se considerar que essa atividade é mínima se comparada aos processos de gerenciamento empregados pela indústria de

código “fechado”.

A descrição do processo de desenvolvimento do Servidor Apache (Apache Foundation, 2010a), conforme descrito por Mockus et al. (2000), pode ser considerado um exemplo clássico das características do desenvolvimento realizado por comunidades FLOSS. Segundo os autores, no início do projeto os desenvolvedores eram apenas voluntários separados geograficamente e que não podiam dedicar grande parte do seu tempo ao desenvolvimento do projeto. Para que fosse possível realizar o desenvolvimento, eles optaram por um processo de desenvolvimento e de tomada de decisão que enfatizava espaços descentralizados e de comunicação assíncrona (como listas de discussão e um sistema de votação para a resolução de conflitos).

O projeto Apache era composto por um núcleo de desenvolvedores que realizava votações quando um novo membro se interessava em entrar neste núcleo. Segundo Mockus et al. (2000), mesmo que não houvesse um processo formalizado de desenvolvimento, os membros do núcleo executavam iterações realizando as seguinte atividades:

1. **Identificação de um problema**: os problemas eram relatados na lista de discussão dos desenvolvedores, em um sistema de comunicação de problemas (BugDB (PEPPLER, 2010)) ou por meio do *newsgroup* do Apache. A prioridade era dada à lista de discussão, pois problemas repassados por membros do núcleo de desenvolvimento possuíam mais informações sobre o problema encontrado. Para acompanhar o andamento do projeto, era criada uma agenda com uma lista de prioridade de desenvolvimento para cada produto do repositório (relatório de problemas);

2. **Escolha de um voluntário**: os relatórios de problemas eram criados por voluntários que analisavam as dificuldades apresentadas nas listas de discussão, no BugDB e no *newsgroup*. Os relatórios eram disponibilizados para que qualquer voluntário se candidatasse para correção. A prioridade era dada à quem já estava mais familiarizados com a funcionalidade;

3. **Identificação da solução**: a principal dificuldade nesta fase não era encontrar uma solução, mas sim decidir qual das várias possibilidades seria a solução mais adequada, neste caso, o desenvolvedor geralmente encaminhava as alternativas para a lista de discussão a fim de obter a opinião do restante do grupo antes de desenvolver uma solução;

4. **Codificação, teste e revisão**: depois que uma solução era definida, o desenvolvedor codificava e testava em seu servidor local. Em seguida ele submetia à revisão pelos membros do núcleo que iriam definir se o código estava pronto para ser incorporado

à versão oficial;

- Cada funcionalidade que era finalizada e incorporada ao produto oficial era automaticamente enviada para uma lista de discussão. Todos os desenvolvedores do núcleo eram responsáveis por rever as alterações para garantir que elas eram adequadas. Além disso, qualquer pessoa, mesmo de fora da comunidade, tinha a possibilidade de se inscrever na lista e fazer as revisões, fato o que acrescentava informações úteis antes do software ser disponibilizado;

- Quando o projeto se aproximava de uma versão "completa" do produto, um dos principais desenvolvedores era definido voluntariamente para ser o gerente de lançamento. Ele deveria identificar problemas críticos que impedissem a liberação, determinava prazo para que esses problemas fossem reparados e definia um "ponto de corte"a partir do qual o software era considerado como uma versão estável e pronta para o lançamento.

Conforme apresentado acima, nota-se que a realização dos testes em projetos FLOSS é uma atividade dos desenvolvedores e geralmente não há um processo coordenado ou planejado a ser seguido. No caso dos projetos FLOSS analisados neste trabalho (listados na Seção 3.1.2), são disponibilizados os conjuntos de casos de teste automatizados construídos no JUnit (JUNIT, 2010) que foram utilizados na realização dos testes pelos desenvolvedores.

Outro fato que pode ser considerado uma das características do desenvolvimento de FLOSS está na consolidação dos testes dos produtos depois da disponibilização de versões *beta*. Exemplo disso está no estudo realizado Mockus et al. (2000) em que eles constataram que, dentre os 15 principais problemas do servidor Apache, apenas três foram identificados e reportados pelo núcleo de desenvolvimento, ou seja, verificou-se que, na sua grande maioria, o teste do sistema ficou a cargo dos usuários e da comunidade "externa".

Apesar de não seguir um processo formalizado de desenvolvimento, considera-se que os resultados obtidos pelas comunidades que desenvolvem FLOSS são equivalentes ou até superiores às organizações que desenvolvem software mais tradicionalmente. Alega-se que os defeitos são encontrados e resolvidos mais rapidamente, pois há "muitos olhos" voltados para os problemas e, além disso, acredita-se que o código é escrito com mais cuidado e criatividade, porque os desenvolvedores estão trabalhando apenas em coisas para as quais eles tem uma verdadeira paixão (RAYMOND, 2000).

A Seção 2.4 detalha o Projeto QualiPSo que consiste no contexto ao qual este projeto está inserido.

2.4 Contexto do trabalho: o projeto QualiPSo

O projeto integrado QualiPSo (*Quality Plataform for Open Source*) propõe-se a definir e implementar tecnologias, procedimentos, leis e políticas com o objetivo de potencializar as práticas de desenvolvimento de software livre, tornando-as confiáveis, reconhecidas e estabelecidas na indústria. Para viabilizar o projeto e a sustentação do software livre como uma solução confiável para a indústria, foi criado um consórcio formado por indústrias, academia e governo (QUALIPSO, 2010).

O projeto é composto por 18 membros fundadores na Europa, Brasil e China, dentre eles (QUALIPSO, 2010):

- Industria: *Atos Origin*, *Bull*, *Engineering Ingegneria Informatica*, *Siemens*, *Telefonica*, *Thales*;

- Governo: Departamento de Inovação e Tecnologias da Informação da Itália;

- Academia: *Centro Ricerche Matematica Pura e Applicata*, *Fraunhofer Institute for Open Communication Systems*, INRIA, *Poznan Supercomputing and Networking Center*, Universidade de São Paulo, *South China University of Technology/Guangzhou Middleware Research Center*, *University of Bozen*, *University of Insubria e University Rey Juan Carlos*.

Além desses membros fundadores, o projeto também está fortemente relacionado e está sendo acompanhado por importantes comunidades de Software de Código Aberto, como a *ObjectWeb* e *Morfeo* (QUALIPSO, 2010). Os objetivos do projeto são listados na Seção 2.4.1.

2.4.1 Objetivos do projeto QualiPSo

Para atingir a meta principal do projeto conforme citado na Seção 2.4, os seguintes objetivos foram traçados (QUALIPSO, 2010):

- Definir métodos, processos de desenvolvimento e modelos de negócio para OSS que estejam de acordo com normas requeridas na indústria de Software;

- Projetar e implementar um ambiente específico (*QualiPSo Factory*) no qual diferentes ferramentas estejam integradas para facilitar e apoiar o desenvolvimento de sistemas OSS viáveis para indústria - garantir uma colaboração segura;

- Implementar ferramentas específicas para checar e garantir a qualidade de OSS provando a existência de propriedades como robustez e escalabilidade para suportar aplicações mais críticas;

- Implementar e apoiar melhores práticas relacionadas à gerência da informação (incluindo código fonte, documentação, etc) para demonstrar a produtividade do desenvolvimento e a evolução de sistemas OSS;

- Demonstrar interoperabilidade que geralmente é o centro da implementação nos padrões abertos em OSS para prover conjuntos de teste e qualidade na integração;

- Compreender as condições legais em que os produtos OSS são protegidos e reconhecidos, sem violar o espírito OSS;

- Desenvolver uma grande rede de profissionais interessados na qualidade do OSS.

A Seção 2.4.2 apresenta as cinco classes de atividades que foram definidas para realização do projeto.

2.4.2 Divisão do projeto QualiPSo em atividades

Para atingir os objetivos do projeto (listados na Seção 2.4.1), o projeto foi estruturado em cinco classes de atividades (QUALIPSO, 2010) que proporcionam a fundamentação e o conteúdo tecnológico em que o projeto está sendo construído. As cinco classes de atividades são:

- Questões legais: está preocupada com a necessidade de um contexto jurídico claro em que FLOSS será capaz de evoluir;

- Modelos de negócios: aborda a necessidade de incorporar novos modelos de desenvolvimento de software que possam lidar com as peculiaridades de FLOSS;

- Interoperabilidade: preocupa-se com as necessidades da indústria por normas baseadas em interoperabilidade de software;

- Resultados confiáveis (RC): aborda a necessidade da definição de fatores de qualidade claramente testados e identificados em produtos FLOSS;

- Processos confiáveis: preocupa-se com a necessidade da definição de uma metodologia para desenvolvimento de FLOSS.

Para execução do projeto QualiPSo, as atividades foram divididas entre os participantes e membros fundadores. Esta dissertação está relacionada à atividade "Resultados Confiáveis" e foi desenvolvida no contexto do projeto QualiPSo em uma parceria da Universidade Federal de Goiás (UFG) em conjunto com a Universidade de São Paulo (USP). A Seção 2.4.3 apresenta os pacotes de trabalhos em que essa atividade foi dividida.

2.4.3 Avaliação de conjuntos de teste de software baseada em critérios estruturais

A meta da atividade Resultados Confiáveis é a identificação, quantificação e avaliação de fatores relacionados à qualidade dos produtos de software (bem como para artefatos produzidos durante o seu desenvolvimento) que possam afetar a confiança dos produtos FLOSS com ênfase em fatores funcionais e não funcionais (QUALIPSO, 2010). Para esta atividade foram definidos os seguintes pacotes de trabalho (*workpackages*):

- WP5.1: Elucidação das metas empresariais da indústria de Software européia para confiabilidade do código aberto;
- WP5.2: Análise dos artefatos e projetos de código aberto relevantes;
- WP5.3: Definição de fatores relevantes para confiabilidade;
- WP5.4: Definição de um padrão de métodos de teste, conjunto de teste e *benchmarks* para OSS;
- WP5.5: Definição e construção de ferramentas para plataforma de colaboração;
- WP5.6: Experimentação e construção do modelo.

O trabalho relatado nesta dissertação consiste em uma parte das atividades do pacote de trabalho WP5.4. O objetivo principal deste trabalho é avaliar a qualidade dos conjuntos de teste disponibilizados pelas comunidades FLOSS, para produtos desenvolvidos na linguagem Java, em relação a critérios de teste estruturais visando a investigar qual a porcentagem de cobertura de código de produção que é efetivamente executada pelos testes disponibilizados juntamente com o código fonte de produtos FLOSS.

CAPÍTULO 3

Discussão e Análise

Este capítulo apresenta os resultados obtidos após a realização dos experimentos por meio da exposição e análise dos resultados, destacando o processo de execução dos testes, as dificuldades encontradas e os materiais utilizados. A Seção 3.1 traz as informações sobre a realização dos experimentos e está organizada da seguinte forma: na Subseção 3.1.1 é apresentado o processo geral definido para execução dos experimentos com a ferramenta EMMA, na Subseção 3.1.2 são listados os materiais (hardware e software) utilizados no trabalho e na Subseção 3.1.3 o processo geral é detalhado de acordo com as especificidades de cada projeto analisado. Na Seção 3.2, têm-se as considerações sobre os resultados dos experimentos. Por fim, na Seção 3.3 é apresentada uma proposta de estratégia de testes incremental.

3.1 Experimentos Realizados

Os produtos FLOSS investigados são implementados em Java e possuem um conjunto de teste criado pela sua comunidade de desenvolvimento visando a verificação constante do FLOSS desenvolvido. Conforme comentado no Capítulo 1, a característica principal de tais conjuntos é que se tratam de conjuntos de testes funcionais, muitos deles criados de forma *ad-hoc*. Nesse sentido, uma pergunta natural seria: "Qual a qualidade desses conjuntos de testes funcionais?". Nos experimentos realizados, a qualidade de tais conjuntos foi avaliada em relação a critérios de teste estruturais visando a investigar qual a porcentagem de cobertura de código de produção que é efetivamente executada pelos testes funcionais disponíveis. Tal avaliação foi conduzida com a ferramenta de teste EMMA (ROUBTSOV, 2010), que apresenta relatórios de cobertura para classes, métodos, blocos e linhas (ver Seção 2.2.4). A seção a seguir descreve o processo geral definido para execução dos experimentos com a ferramenta EMMA.

3.1.1 Processo geral dos experimentos com a EMMA

O processo geral para realização dos experimentos com a ferramenta EMMA, ilustrado na Figura 3.1, segue cinco passos básicos que são: 1) efetuar o download da versão mais recente disponível do código fonte (*source*) do projeto FLOSS; 2) executar os procedimentos para compilar o código fonte de produção e de teste; 3) executar os testes disponibilizados pela comunidade que desenvolveu o projeto; 4) instrumentar o código fonte com a ferramenta EMMA para que seja possível coletar as informações de cobertura; e 5) executar os testes novamente para que os dados de cobertura sejam obtidos e os relatórios sejam gerados pela EMMA.

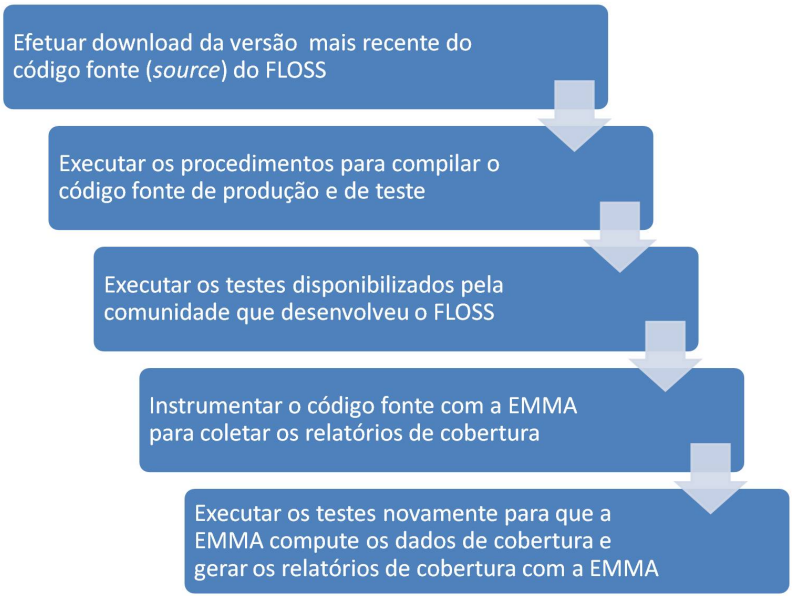
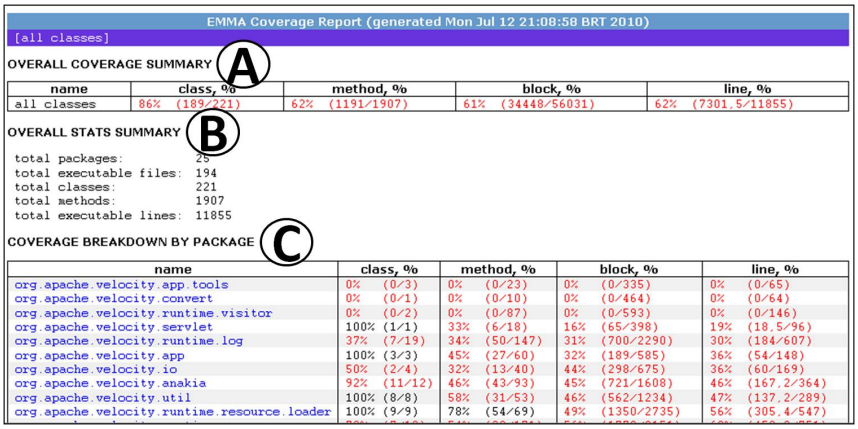


Figura 3.1: *Processo geral para realização dos experimentos com a EMMA*

Para realização do procedimento descrito no 1º Passo, efetuar o download da versão mais recente disponível do código fonte, utilizou-se repositórios do Subversion (SVN *repository*) (SUBVERSION, 2010) ou links disponíveis diretamente da página oficial dos projetos. Os procedimentos para compilar os códigos fonte de produção e de teste, 2º Passo, foram realizados com o Ant conforme descrito na Seção 2.2.4. Nesse procedimento são identificadas as necessidades de configuração do ambiente de teste, instalação de ferramentas auxiliares e resolução de dependências. Após isso, tem-se a execução dos testes disponibilizados pela comunidade juntamente com o projeto, 3º Passo, que corresponde à execução dos códigos de teste unitários desenvolvidos, em geral formatados de acordo com o arcabouço do o JUnit (JUNIT, 2010). Em seguida, tem-se a instrumentação do código fonte, 4º Passo, que corresponde a uma alteração dos códigos de produção por meio da ferramenta EMMA para que seja possível a coleta dos dados de cobertura. Por fim, no 5º Passo, ocorre uma nova execução dos testes, agora com o código fonte de produção já

instrumentado, para que a ferramenta EMMA possa coletar os dados de cobertura e gerar os relatórios para análise.

O objetivo final do processo de realização dos testes é o relatório de cobertura que possibilita analisar qual a porcentagem de código de produção está sendo coberta pelo conjunto de testes disponibilizado para cada projeto. A Figura 3.2 consiste em um exemplo de parte do relatório em formato de página da Web com os dados de cobertura gerados pela ferramenta EMMA para o projeto Velocity (Apache Foundation, 2010c).


EMMA Coverage Report (generated Mon Jul 12 21:08:58 BRT 2010)
[all classes]
OVERALL COVERAGE SUMMARY A

name	class, %	method, %	block, %	line, %
all classes	86% (189/221)	62% (1191/1907)	61% (34448/56031)	62% (7301.5/11855)

OVERALL STATS SUMMARY B

total packages:	25
total executable files:	194
total classes:	221
total methods:	1907
total executable lines:	11855

COVERAGE BREAKDOWN BY PACKAGE C

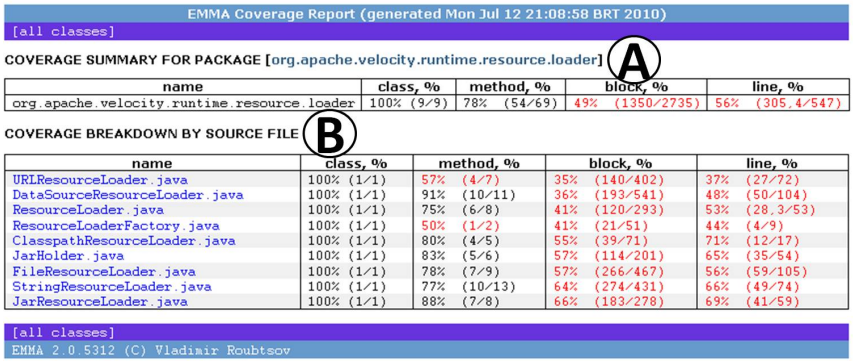
name	class, %	method, %	block, %	line, %
org.apache.velocity.app.tools	0% (0/3)	0% (0/23)	0% (0/335)	0% (0/65)
org.apache.velocity.convert	0% (0/1)	0% (0/10)	0% (0/464)	0% (0/64)
org.apache.velocity.runtime.visitor	0% (0/2)	0% (0/87)	0% (0/593)	0% (0/146)
org.apache.velocity.servlet	100% (1/1)	33% (6/18)	16% (65/398)	19% (18.5/96)
org.apache.velocity.runtime.log	37% (7/19)	34% (50/147)	31% (700/2290)	30% (184/607)
org.apache.velocity.app	100% (3/3)	45% (27/60)	32% (189/585)	36% (54/148)
org.apache.velocity.io	50% (2/4)	32% (13/40)	44% (298/675)	36% (60/169)
org.apache.velocity.anakia	92% (11/12)	46% (43/93)	45% (721/1608)	46% (167.2/364)
org.apache.velocity.util	100% (8/8)	58% (31/53)	46% (562/1234)	47% (137.2/289)
org.apache.velocity.runtime.resource.loader	100% (9/9)	78% (54/69)	49% (1350/2735)	56% (305.4/547)

Figura 3.2: *Exemplo de relatório gerado pela ferramenta EMMA em nível de projeto*

Os dados representados no cabeçalho do relatório, Figura 3.2(A), correspondem ao resumo de cobertura global (*OVERALL COVERAGE SUMMARY*) em que são apresentados os seguintes dados: porcentagem de classes cobertas, número de classes cobertas e total de classes do projeto; porcentagem de métodos cobertos, número de métodos cobertos e total de métodos existentes nas classes do projeto; porcentagem de blocos cobertos, número de blocos cobertos e total de blocos do projeto; e porcentagem de linhas de código cobertas, número de linhas de código cobertos e total de linhas de código existentes no projeto[1]. O segundo conjunto de informações do relatório, Figura 3.2(B), consiste em um resumo das estatísticas do projeto (*OVERALL STATS SUMMARY*) em que os seguintes dados são listados: total de pacotes do projeto, total de arquivos executáveis, total de classes, total de métodos e total de linhas executáveis. Por fim, tem-se na Figura 3.2(C) o detalhamento dos dados de cobertura para cada pacote do projeto (*COVERAGE BREAKDOWN BY PACKAGE*) em que, o mesmo conjunto de dados apresentados no cabeçalho do relatório (ou seja, classes, métodos, blocos e linhas cobertos), também é apresentado detalhadamente para cada pacote.

[1]Observa-se que, conforme comentado na Seção 2.2.4, devido a problemas que a Emma apresentou no cálculo da cobertura de linhas em alguns dos projetos avaliados, tais dados de cobertura foram omitidos dos relatórios apresentados neste capítulo. No DVD tais dados de cobertura são apresentados para aqueles projetos nos quais a ferramenta foi capaz de calcular.

Nesse relatório em formato de página da Web, cada pacote é um link para o detalhamento dos dados de cobertura das classes que o compõe. A Figura 3.3 ilustra um relatório em formato de página da Web com os dados de cobertura gerados pela ferramenta EMMA para o pacote *org.apache.velocity.runtime.resource.loader* do projeto Velocity.



EMMA Coverage Report (generated Mon Jul 12 21:08:58 BRT 2010)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [org.apache.velocity.runtime.resource.loader] (A)

name	class, %	method, %	block, %	line, %
org.apache.velocity.runtime.resource.loader	100% (9/9)	78% (54/69)	49% (1350/2735)	56% (305.4/547)

COVERAGE BREAKDOWN BY SOURCE FILE (B)

name	class, %	method, %	block, %	line, %
URLResourceLoader.java	100% (1/1)	57% (4/7)	35% (140/402)	37% (27/72)
DataSourceResourceLoader.java	100% (1/1)	91% (10/11)	36% (193/541)	48% (50/104)
ResourceLoader.java	100% (1/1)	75% (6/8)	41% (120/293)	53% (28.3/53)
ResourceLoaderFactory.java	100% (1/1)	50% (1/2)	41% (21/51)	44% (4/9)
ClasspathResourceLoader.java	100% (1/1)	80% (4/5)	55% (39/71)	71% (12/17)
JarHolder.java	100% (1/1)	83% (5/6)	57% (114/201)	65% (35/54)
FileResourceLoader.java	100% (1/1)	78% (7/9)	57% (266/467)	56% (59/105)
StringResourceLoader.java	100% (1/1)	77% (10/13)	64% (274/431)	66% (49/74)
JarResourceLoader.java	100% (1/1)	88% (7/8)	66% (183/278)	69% (41/59)

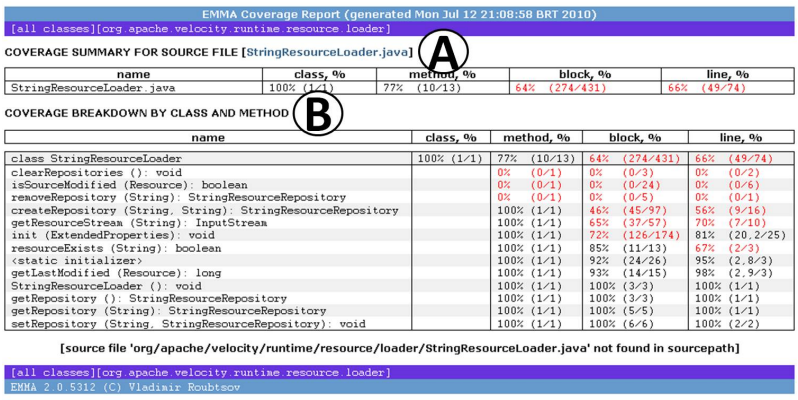
[all classes]

EMMA 2.0.5312 (C) Vladimir Roubtsov

Figura 3.3: *Exemplo de relatório gerado pela ferramenta EMMA em nível de pacote*

Analogamente à Figura 3.2, o cabeçalho do relatório em nível de pacote, Figura 3.3(A), apresenta o resumo da cobertura obtida (*COVERAGE SUMMARY FOR PACKAGE*) para o pacote que está sendo visualizado. Já a Figura 3.3(B) traz o detalhamento dos dados de cobertura para cada arquivo “.java"que existe no pacote (*COVERAGE BREAKDOWN BY SOURCE FILE*).

Assim como no relatório completo do projeto, Figura 3.2, em que o nome de cada pacote é um link para o detalhamento dos dados de cobertura das suas classes, no relatório em nível de pacote, Figura 3.3, cada arquivo “.java” também é um link que detalha as informações de cobertura da classe e métodos que o compõe. A Figura 3.4 representa um relatório em formato de página da Web com os dados de cobertura gerados pela ferramenta EMMA para o arquivo *StringResourceLoader.java* que pertence ao pacote *org.apache.velocity.runtime.resource.loader* do projeto Velocity.



EMMA Coverage Report (generated Mon Jul 12 21:08:58 BRT 2010)

[all classes][org.apache.velocity.runtime.resource.loader]

COVERAGE SUMMARY FOR SOURCE FILE [StringResourceLoader.java] (A)

name	class, %	method, %	block, %	line, %
StringResourceLoader.java	100% (1/1)	77% (10/13)	64% (274/431)	66% (49/74)

COVERAGE BREAKDOWN BY CLASS AND METHOD (B)

name	class, %	method, %	block, %	line, %
class StringResourceLoader	100% (1/1)	77% (10/13)	64% (274/431)	66% (49/74)
clearRepositories (): void		0% (0/1)	0% (0/3)	0% (0/2)
isSourceModified (Resource): boolean		0% (0/1)	0% (0/24)	0% (0/6)
removeRepository (String): StringResourceRepository		0% (0/1)	0% (0/5)	0% (0/1)
createRepository (String, String): StringResourceRepository		100% (1/1)	46% (45/97)	56% (9/16)
getResourceStream (String): InputStream		100% (1/1)	65% (37/57)	70% (7/10)
init (ExtendedProperties): void		100% (1/1)	72% (126/174)	81% (20.2/25)
resourceExists (String): boolean		100% (1/1)	85% (11/13)	67% (2/3)
<static initializer>		100% (1/1)	92% (24/26)	95% (2.8/3)
getLastModified (Resource): long		100% (1/1)	93% (14/15)	98% (2.9/3)
StringResourceLoader (): void		100% (1/1)	100% (3/3)	100% (1/1)
getRepository (): StringResourceRepository		100% (1/1)	100% (3/3)	100% (1/1)
getRepository (String): StringResourceRepository		100% (1/1)	100% (5/5)	100% (1/1)
setRepository (String, StringResourceRepository): void		100% (1/1)	100% (6/6)	100% (2/2)

[source file 'org/apache/velocity/runtime/resource/loader/StringResourceLoader.java' not found in sourcepath]

[all classes][org.apache.velocity.runtime.resource.loader]

EMMA 2.0.5312 (C) Vladimir Roubtsov

Figura 3.4: *Exemplo de relatório gerado pela ferramenta EMMA em nível de classe*

De forma semelhante à Figura 3.3, o cabeçalho do relatório em nível de classe, Figura 3.4(A), apresenta o resumo da cobertura obtida (*COVERAGE SUMMARY FOR SOURCE FILE*) para o arquivo ".java" que está sendo visualizado. E a Figura 3.4(B) consiste no detalhamento dos dados de cobertura para cada classe e os métodos que existem neste arquivo (*COVERAGE BREAKDOWN BY CLASS AND METHOD*).

Para cada um dos projetos analisados neste trabalho, foram gerados esses três tipos de relatório: em formato de página Web, em formato de arquivo XML e formato de texto. Os relatórios completos e nos três formatos podem ser encontrados no DVD que acompanha a dissertação, no qual estão todos os códigos dos projetos analisados. A próxima seção apresenta a lista de materiais (hardware e software) utilizados no desenvolvimento deste trabalho.

3.1.2 Materiais

Nesta seção serão apresentados os recursos de hardware e software utilizados durante o desenvolvimento do trabalho.

Hardware

1. Notebook Processador Intel Celeron M 410 e 1 Gb de Memória Ram;

Software

1. Sistema operacional: Open Suse 11.1 (Novell, Inc., 2010);

2. Ambiente de desenvolvimento: Kad 1.0 (LUCENA, 2010).

Projetos Java analisados

Os seguintes projetos Java foram analisados conforme uma lista definida no projeto QualiPso: Canoo WebTest (Canoo Web Test, 2010), HttpUnit (GOLD, 2010), JFreeChart (JFree Community, 2010), JMeter (Apache Foundation, 2010b), Log4j (Apache Foundation, 2010d), Mondrian (Pentaho Corporation, 2010), Poi (Apache Foundation, 2010f), Velocity (Apache Foundation, 2010c), Weka (WEKA, 2010) e Xerces2 (Apache Foundation, 2010g). A próxima seção apresenta, em ordem alfabética de acordo com o nome do projeto, detalhes específicos relacionados ao processo de análise de cada um dos projetos Java citados.

3.1.3 Detalhes específicos de cada projeto

Nesta seção serão apresentados detalhes relacionados às especificidades de cada projeto encontrados durante o processo de realização dos experimentos, bem como um resumo da avaliação de cobertura obtida durante as análises (os relatórios completos de cada projeto podem ser encontrados no DVD que acompanha a dissertação). Para cada um desses projetos, serão descritos os seguintes dados: 1) breve descrição; 2) versão analisada; 3) endereço do repositório onde o projeto está disponível; 4) procedimentos realizados na execução dos experimentos; e 5) outras informações e problemas encontrados.

O objetivo principal da inclusão de tais detalhes no texto da dissertação é de propiciar que o experimento aqui descrito possa ser reproduzido por outros grupos de pesquisa, além de contribuir para a definição de um mecanismo que permita generalizar a forma de condução de experimentos desse tipo para uma grande quantidade de projetos.

Canoo WebTest

Canoo WebTest é uma ferramenta para automatização de testes de aplicações web (Canoo Web Test, 2010).

Versão analisada:
3.0
Endereço do repositório SVN:
https://svn.canoo.com/trunk/webtest/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Canoo;

2) Compilar o código fonte e os códigos de teste utilizando o comando: *ant*;

3) Executar os casos de teste utilizando o comando: *ant test*;

4) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Canoo/lib*;

5) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp ../lib/emma.jar emma instr -d src/main/java/com/canoo/webtest -ip src/main/java/com/canoo/webtest*;

6) Executar novamente os casos de teste utilizando o seguinte comando para que sejam coletados os dados de cobertura: *ant test*;

7) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp lib/emma.jar emma merge -in coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp lib/emma.jar emma report -r txt,html,xml -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Canoo WebTest:

Tabela 3.1: *Canoo WebTest: resumo da cobertura obtida*

Classes % (Cobertas / Total)	85% (243 / 286)
Métodos % (Cobertos / Total)	73% (1769 / 2418)
Blocos % (Cobertos / Total)	67% (27332 / 40927)

Outras informações e problemas encontrados:
Neste projeto, não foram encontrados problemas na execução dos testes.

HttpUnit

O HttpUnit possibilita a criação de testes para aplicações Web. Ele emula comportamentos importantes de um navegador, incluindo submissão de formulário, JavaScript, autenticação http básica, cookies e redirecionamento automático de páginas, além de permitir códigos de teste em Java para examinar páginas retornadas em texto, como XML DOM, ou contêiner de formulários, tabelas e links. Pode ser combinado com arcabouços como o JUnit para a verificação do funcionamento de um web site (GOLD, 2010).

Versão analisada:
1.7

Endereço do repositório SVN:
https://httpunit.svn.sourceforge.net/svnroot/httpunit/trunk/

Procedimentos realizados na execução do experimento:
1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada HttpUnit;

2) Compilar o código fonte utilizando o comando: *ant compile*;

3) Compilar os códigos de teste utilizando o comando: *ant testcompile*;

4) Executar os casos de teste utilizando o comando: *ant test*;

5) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *HttpUnit/httpunit/build*;

6) Fazer uma cópia dos bytecodes originais das classes que estão na pasta *HttpUnit/httpunit/build/Classes* em uma nova pasta *HttpUnit/httpunit/build/Classes.orig* de maneira que as duas fiquem com o mesmo conteúdo;

7) Criar uma pasta chamada *HttpUnit/httpunit/build/Classes.inst* para guardar os bytecodes instrumentados;

8) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp build/emma.jar emma instr -d classes.inst -ip classes.orig*;

9) Copiar os bytecodes da pasta *HttpUnit/httpunit/build/Classes.inst* para pasta *HttpUnit/httpunit/build/Classes*;

10) Adicionar a seguinte referência à ferramenta Emma no *classpath* do arquivo *build.xml* para que os dados de cobertura de teste possam ser coletados: *<property name = "emma.jar" value = "${build.dir}/emma.jar" />*;

11) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test*;

12) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp build/emma.jar emma merge -in build/coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp build/emma.jar emma report -r txt,xml,html -in coverage.es*.

Resumo dos dados de cobertura obtidos para o HttpUnit:

Tabela 3.2: *HttpUnit: resumo da cobertura obtida*

Classes % (Cobertas / Total)	96% (317 / 330)
Métodos % (Cobertos / Total)	81% (2837 / 3491)
Blocos % (Cobertos / Total)	83% (36090 / 43333)

Outras informações e problemas encontrados:

As atividades descritas nos Passos 6, 7, 8 e 9 são necessárias, pois o comando "instr" da ferramenta Emma não copia os itens não-executáveis, como interfaces e pacotes de recursos que são necessários para que não haja erros de dependência na execução dos testes, então deve-se ter o conteúdo original da pasta com as classes (código objeto) que serve como um backup para que não haja erros de dependência durante a execução dos testes no programa. Sendo assim, ao realizar esse procedimento e as classes instrumentadas forem copiadas novamente para a pasta de classes, os itens não executáveis e demais elementos necessários para correto funcionamento do programa já estarão na pasta original e possibilitarão a correta execução dos casos de teste.

JFreeChart

O JFreeChart consiste em uma biblioteca livre para criação de gráficos desenvolvida em Java. Ela pode ser utilizada por desenvolvedores para facilitar a utilização de gráficos em suas aplicações (JFree Community, 2010).

Versão analisada:
1.0.13

Endereço do repositório SVN:
https://jfreechart.svn.sourceforge.net/svnroot/jfreechart/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada JFreeChart;

2) Compilar o código fonte utilizando o comando: *ant compile*;

3) Alterar o arquivo *build.xml* para corrigir o endereço dos arquivos que serão necessários para compilar os códigos de teste:

a. Na linha 342, o endereço correto será: *"${basedir}/lib/${jfreechart.name}-${jfreechart.version}.jar"*;

b. Na linha 343, o endereço correto será: *"${basedir}/lib/${jfreechart.name}-${jfreechart.version}-experimental.jar"*;

4) Compilar os códigos de teste utilizando o comando: *ant compile-tests*;

5) Alterar o arquivo *build.xml* para corrigir o endereço dos arquivos que serão necessários para executar os testes:

a. Na linha 368, o endereço correto será: *"${basedir}/lib/${jfreechart.name}-${jfreechart.version}.jar"*;

b. Na linha 369, o endereço correto será: *"${basedir}/lib/${jfreechart.name}-${jfreechart.version}-experimental.jar"*;

6) Executar os casos de teste utilizando o comando: *ant test*;

7) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *JFreeChart/lib*;

8) Instrumentar os bytecodes com a ferramenta Emma utilizando os comandos:

a. *java -cp ../lib/emma.jar emma instr -m overwrite -cp ../lib/jfreechart-1.0.13.jar*;

b. *java -cp ../lib/emma.jar emma instr -m overwrite -cp ../lib/jfreechart-1.0.13-experimental.jar*;

9) Renomear os seguintes arquivos que estão na pasta *JFreeChart/lib* para que eles não sejam sobrescritos no processo de execução dos testes:

a. *jfreechart-1.0.13.jar* deverá ser renomeado para *jfreechart-1.0.13-instr.jar*;

b. *jfreechart-1.0.13-experimental.jar* deverá ser renomeado para *jfreechart-1.0.13-experimental-instr.jar*;

10) Adicionar a seguinte referência à ferramenta Emma no *classpath* do arquivo *build.xml* para que os dados de cobertura de teste possam ser coletados: *<property name = "emma.jar" value = "${basedir}/lib/emma.jar" />*;

11) Alterar o arquivo *build.xml* para fazer a correta referência aos nomes dos arquivos alterados no passo 9:

a. Na linha 368, o novo endereço será: *"${basedir}/lib/${jfreechart.name}-${jfreechart.version}-instr.jar"*;

b. Na linha 369, o novo endereço será: *"${basedir}/lib/${jfreechart.name}-${jfreechart.version}-experimental-instr.jar"*;

12) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test*;

13) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp ../lib/emma.jar emma merge -in coverage.em -in ../coverage.ec -out coverage.es*;

b. *java -cp ../lib/emma.jar emma report -r txt,html,xml -in coverage.em,../coverage.ec*.

Resumo dos dados de cobertura obtido para o JFreeChart:

Tabela 3.3: *JFreeChart: resumo da cobertura obtida*

Classes % (Cobertas / Total)	72% (369 / 514)
Métodos % (Cobertos / Total)	52% (4131 / 8017)
Blocos % (Cobertos / Total)	47% (97529 / 209564)

Outras informações e problemas encontrados:

A correção dos endereços dos caminhos dos arquivos apresentados nos Passos 3 e 5 foram necessárias, pois o endereço que está no arquivo *build.xml* original não possui a pasta *lib* no caminho. Além disso, vale comentar que no processo de execução dos testes, mesmo antes da instrumentação dos arquivo para análise com a ferramenta Emma, o caso de teste *RelativeDateFormatTests.java* apresentava uma falha e interrompia a execução dos testes, por isso ele foi excluído do experimento e não teve sua cobertura avaliada.

JMeter

O Apache JMeter é um aplicativo projetado para analisar o comportamento de testes funcionais e para mensurar desempenho. Ele foi originalmente projetado para

testes de aplicações Web, mas tem sido expandido para outras funções de teste. Pode ser usado para testar o desempenho tanto de recursos estáticos e dinâmicos (como: imagens, *Servlets*, *scripts Perl*, *Java Objects*, bases de dados e consultas e servidores FTP) (Apache Foundation, 2010b).

Versão analisada:
2.3.4
Endereço do repositório SVN:
http://svn.apache.org/repos/asf/jakarta/jmeter/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada JMeter;

2) Compilar o código fonte utilizando o comando: *ant*;

3) Compilar os códigos de teste utilizando o comando: *ant compile-tests*;

4) Alterar o arquivo *JMeter/build.xml* para excluir os casos de teste *batchtest* e *batchtestserver*:

a. A alteração deverá ocorrer no *target test* que, após as alterações, deverá ficar da seguinte forma: *<target name ="test" depends ="compile-tests,_test" description = "Run tests (use -Djava.awt.headless=true on systems without graphic displays)" />*;

5) Executar os casos de teste utilizando o comando: *ant test*;

6) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *JMeter/lib/ext*;

7) Instrumentar os bytecodes com a ferramenta Emma utilizando os comandos:

a. Na pasta *JMeter/lib/ext*, deve-se instrumentar todos os arquivos com a extensão *.jar* (um arquivo por vez), utilizando o comando: *java -cp lib/ext/emma.jar emma instr -m overwrite -cp lib/ext/{NomeDoArquivo}.jar*;

b. Na pasta *JMeter/bin*, deve-se instrumentar o arquivo *ApacheJMeter.jar*, utilizando o comando: *java -cp lib/ext/emma.jar emma instr -m overwrite -cp bin/ApacheJMeter.jar*;

8) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test*;

9) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp lib/ext/emma.jar emma merge -in coverage.em -in bin/coverage.ec -out coverage.es*;

b. *java -cp lib/ext/emma.jar emma report -r txt,html,xml -in coverage.es*.

Resumo dos dados de cobertura obtidos para o JMeter:

Tabela 3.4: *JMeter: resumo da cobertura obtida*

Classes % (Cobertas / Total)	80% (667 / 836)
Métodos % (Cobertos / Total)	53% (4016 / 7646)
Blocos % (Cobertos / Total)	46% (80546 / 174011)

Outras informações e problemas encontrados:

No processo de execução dos testes, alguns casos de teste foram excluídos conforme apresentado no Passo 4, pois não foram encontradas informações sobre como configurá-los.

Log4j

O Log4j consiste em uma ferramenta de *logging*. *Logging* de uma aplicação pode ser considerada como uma das maneiras de *debug* (i.e.: registro em algum meio, como arquivo ou em bancos de dados, de eventos ocorridos na aplicação, sejam eles simples informações de uso ou indicações de erros no processamento). Neste contexto, o arcabouço de *logging* Log4j fornece ao desenvolvedor a possibilidade de trabalhar com controle do *log* de dados de uma aplicação (Apache Foundation, 2010d).

Versão analisada:

1.2.16

Endereço do repositório SVN:

http://svn.apache.org/repos/asf/logging/log4j/trunk/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Log4j;

2) Compilar o código fonte do projeto de acordo com as seguintes orientações:

a. Executar o comando *ant dist* na pasta raiz do projeto Log4j, pois neste projeto há dois arquivos denominados *build.xml* em que, um consiste no arquivo de configuração do projeto completo e outro consiste no arquivo de configuração do projeto de testes;

b. A estação de trabalho onde estão sendo realizados os experimentos deve estar conectada à Internet, pois este projeto utiliza o Maven (Apache Foundation, 2010h) para resolução de dependências;

3) Compilar os códigos de teste utilizando o comando *ant build* dentro da pasta *Log4j/tests* em que está localizado o arquivo de configuração do projeto de testes;

4) Executar os casos de teste utilizando o comando *ant runAll* dentro da pasta *Log4j/tests* em que está localizado o arquivo de configuração do projeto de testes;

5) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Log4j/dist/lib*;

6) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp ../dist/lib/emma.jar emma instr -m overwrite -cp ../dist/lib/log4j-1.2.16.jar*;

7) Adicionar a seguinte referência à ferramenta Emma no *classpath* do arquivo *Log4j/tests/build.xml* para que os dados de cobertura de teste possam ser coletados: *<property name = "emma.jar" value = "../dist/lib/emma.jar"/>*;

8) No arquivo *Log4j/tests/build.xml*, localizar a *tag <target name = "build" depends = "parentBuild, prepare" >* e alterar o valor da propriedade *haltonfailure* para *no* de forma que o conteúdo fique da seguinte maneira: *<property name = "haltonfailure" value = "no" />*;

9) Executar novamente os casos de teste para que sejam coletados os dados de cobertura, no entanto agora deve-se utilizar o comando *ant -Dlog4j.jar = ../dist/lib/log4j-1.2.16_instr.jar runAll* dentro da pasta *Log4j/tests* em que esta localizado o arquivo de configuração do projeto de testes;

10) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp ../dist/lib/emma.jar emma merge -in coverage.ec -in coverage.em -out coverage.es*;

b. *java -cp ../dist/lib/emma.jar emma report -r txt,xml,html -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Log4j:

Tabela 3.5: *Log4j: resumo da cobertura obtida*

Classes % (Cobertas / Total)	36% (88 / 245)
Métodos % (Cobertos / Total)	37% (715 / 1955)
Blocos % (Cobertos / Total)	38% (14971 / 39216)

Outras informações e problemas encontrados:

No Passo 9, o comando para execução dos testes possui um parâmetro diferente em relação ao Passo 4, isso se dá, pois no Passo 9 é informado o caminho onde estão os arquivos que foram instrumentados e que devem ser utilizados na coleta dos dados de cobertura. Já a alteração do arquivo *Log4j/tests/build.xml*, descrita no Passo 8, foi necessária, pois um dos casos de teste, denominado *Test org.apache.log4j.helpers.OptionConverterTestCase* apresentou uma falha que não permitia que os outros casos de testes fossem executados, sendo assim, com a alteração da propriedade conforme descrito no Passo 8, a execução não é interrompida.

Mondrian

O Mondrian é um servidor OLAP que compõe a plataforma de BI do *Pentaho* que permite ter acesso a grandes quantidades de dados em tempo real. Ele funciona como um conector entre um servidor OLAP e um banco de dados relacional possibilitando a geração de instruções SQL para o banco de dados e o processamento dos dados resultantes (Pentaho Corporation, 2010).

Versão analisada:
3.0.2

Endereço do repositório SVN:
http://source.pentaho.org/svnmondrianroot/trunk/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Mondrian;

2) Compilar o código fonte utilizando o comando: *ant compile*;

3) Compilar os códigos de teste e executar os casos de teste utilizando o comando: *ant test*;

4) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Mondrian/lib*;

5) Alterar o arquivo *Mondrian/build.xml* para acrescentar referência à ferramenta Emma no *classpath*. Para isso, a linha <*property name='jar.emma' value='emma.jar'/*> deve ser acrescentada como elemento filho do nó <*path id="project.classpath"*>;

6) Fazer uma cópia dos bytecodes originais das classes que estão na pasta *Mondrian/classes* em uma nova pasta chamada *Mondrian/classes.orig*;

7) Criar uma pasta chamada *Mondrian/classes.inst* para guardar os bytecodes instrumentados;

8) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp lib/emma.jar emma instr -d classes.inst -ip classes.orig*;

9) Copiar os bytecodes instrumentados da pasta *Mondrian/classes.inst* para pasta *Mondrian/classes*;

10) Alterar o arquivo *Mondrian/build.xml* para excluir as comandos que compilam os códigos fonte e evitar que os bytecodes instrumentados sejam sobrescritos na execução dos teste. Para isso, deve-se excluir do *target test* a dependência do *target compile*;

11) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test*;

12) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp lib/emma.jar emma merge -in coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp lib/emma.jar emma report -r txt,xml,html -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Mondrian:

Tabela 3.6: *Mondrian: resumo da cobertura obtida*

Classes % (Cobertas / Total)	39% (632 / 1605)
Métodos % (Cobertos / Total)	21% (2310 / 11042)
Blocos % (Cobertos / Total)	23% (64941 / 283245)

Outras informações e problemas encontrados:

Analogamente ao que foi apresentado para o projeto HttpUnit, a execução dos Passos 6, 7, 8 e 9 visam a evitar erros de dependência no processo de execução dos casos de teste do Mondrian.

Poi

A missão do Apache POI é criar e manter API's Java para manipular vários formatos de arquivo com base nas normas *Office Open XML* (OOXML) e *Compound Document Format* (OLE2) da Microsoft. Por meio das API's desenvolvidas neste projeto, pode-se ler e gravar arquivos do Excel usando Java, além de ser possível executar operações de leitura e escrita em arquivos do Word e PowerPoint (Apache Foundation, 2010f).

Versão analisada:

3.6

Endereço do repositório SVN:

http://svn.apache.org/repos/asf/poi/trunk/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Poi;

2) Compilar o código fonte utilizando o comando: *ant compile-all*;

3) Alterar o arquivo *Poi/build.xml* para excluir os casos de teste *TestDate.java*, *TestHSSFDataFormatter.java* e *TestPOIXMLProperties.java*, para isso, a alteração deverá ocorrer:

a. No *target test-main* em que as seguintes linhas devem ser inseridas como elemento filhos do nó *fileset*: <*exclude name="**/formula/functions/TestDate.java"*/> e <*exclude name="**/usermodel/TestHSSFDataFormatter.java"*/>;

b. No *target ooxml-test-runner* em que a seguinte linha deve ser inserida como elemento filho do nó *fileset*: <*exclude name="**/TestPOIXMLProperties.java"*/>;

4) Compilar os códigos de teste e executar os casos de teste utilizando o comando: *ant test-all*;

5) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Poi/lib*;

6) Alterar o arquivo *Poi/build.xml* para acrescentar referência à ferramenta Emma no *classpath*. Para isso, a linha <*property name="main.emma.jar" location="${main.lib}/emma.jar"*/> deve ser acrescentada como elemento filho do nó *projetc*;

7) Fazer uma cópia dos bytecodes originais das classes que estão nas pastas *Poi/build/NOME_PASTA* em novas pastas *Poi/build/NOME_PASTA.orig*. Os nomes das pastas que devem ser utilizadas nesse procedimento são:

a. classes;

b. contrib-classes;

c. examples-classes;

d. ooxml-classes;

e. ooxml-lite-classes; e

f. scratchpad-classes;

8) Criar pastas chamadas *Poi/build/NOME_PASTA.inst* para guardar os bytecodes instrumentados de cada uma das pastas descritas no Passo 6;

9) Instrumentar os bytecodes com a ferramenta Emma utilizando o seguinte comando para cada uma das pastas descritas no Passo 6: *java -cp ../lib/emma.jar emma instr -d NOME_PASTA.inst -ip NOME_PASTA.orig*;

10) Copiar os bytecodes de cada uma das pastas *Poi/build/NOME_PASTA.inst* para as pastas *Poi/build/NOME_PASTA*;

11) Alterar o arquivo *Poi/build.xml* para excluir as comandos que compilam os códigos fonte e evitar que os bytecodes instrumentados sejam sobrescritos na execução dos teste. Para isso, deve-se alterar o seguinte:

a. No *target test*: excluir a dependência do *target compile*;

b. No *target test-main*: excluir a dependência do *target compile-main*;

c. No *target test-contrib*: excluir a dependência do *target compile-main* e do *target compile-contrib*;

d. No *target test-ooxml*: excluir a dependência do *target compile-main* e do *target compile-ooxml*;

e. No *target test-scratchpad*: excluir a dependência do *target compile-main* e do *target compile-scratchpad*;

12) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test-all*;

13) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp lib/emma.jar emma merge -in build/coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp lib/emma.jar emma report -r txt,xml,html -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Poi:

Tabela 3.7: *Poi: resumo da cobertura obtida*

Classes % (Cobertas / Total)	34% (873 / 2582)
Métodos % (Cobertos / Total)	10% (4182 / 40845)
Blocos % (Cobertos / Total)	10% (84256 / 856179)

Outras informações e problemas encontrados:

Analogamente ao que foi apresentado para o projeto HttpUnit, a execução dos Passos 7, 8, 9 e 10 visam evitar erros de dependência no processo de execução dos casos de teste do Poi.

A exclusão de casos de teste descrita no Passo 3 se fez necessária, pois no processo de execução dos testes, mesmo antes da instrumentação dos arquivos para análise com a ferramenta Emma, os casos de teste *TestDate.java*, *TestHSSFDataFormatter.java* e *TestPOIXMLProperties.java* apresentaram falhas que interrompiam a execução dos testes.

Velocity

O Velocity consiste em um *engine* baseado em Java para a construção de *templates*. Ele permite que se utilize modelos simples, porém poderosos, para referenciar objetos definidos em códigos Java (Apache Foundation, 2010c).

Versão analisada:

1.6.3

Endereço do repositório SVN:

http://svn.apache.org/repos/asf/velocity/engine/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Velocity;

2) Compilar o código fonte utilizando o comando: *ant compile-src*;

3) Compilar os códigos de teste utilizando o comando: *ant compile-test*;

4) Executar os casos de teste utilizando o comando: *ant test*;

5) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Velocity/bin/test-lib*;

6) Fazer uma cópia dos bytecodes originais das classes que estão na pasta *Velocity/bin/Classes* em uma nova pasta *Velocity/bin/Classes.orig* de maneira que as duas fiquem com o mesmo conteúdo;

7) Criar uma pasta chamada *Velocity/bin/Classes.inst* para guardar os bytecodes instrumentados;

8) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp ../bin/test-lib/emma.jar emma instr -d classes.inst -ip classes.orig*;

9) Copiar os bytecodes da pasta *Velocity/bin/Classes.inst* para pasta *Velocity/bin/Classes*;

10) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test*;

11) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp ..bin/test-lib/emma.jar emma merge -in bin/coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp ..bin/test-lib/emma.jar emma report -r txt,html,xml -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Velocity:

Tabela 3.8: *Velocity: resumo da cobertura obtida*

Classes % (Cobertas / Total)	86% (189 / 221)
Métodos % (Cobertos / Total)	62% (1191 / 1907)
Blocos % (Cobertos / Total)	61% (34448 / 56031)

Outras informações e problemas encontrados:

Analogamente ao que foi apresentado para o projeto HttpUnit, a execução dos Passos 6, 7, 8 e 9 visam a evitar erros de dependência no processo de execução dos casos de teste do Velocity.

Weka

Weka é uma coleção de algoritmos de aprendizado de máquina para realização de tarefas de mineração de dados. Esses algoritmos podem ser aplicados diretamente a um conjunto de dados ou utilizados no desenvolvimento de um programa Java qualquer. Weka contém ferramentas para: pré-processamento de dados, classificação, regressão, *clustering*, regras de associação e visualização (WEKA, 2010).

Versão analisada:
3.7.1

Endereço do repositório SVN:
https://svn.scms.waikato.ac.nz/svn/weka/

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Weka;

2) Compilar o código fonte utilizando o comando: *ant*;

3) Compilar os códigos de teste utilizando o comando: *ant compile_tests*;

4) Executar os casos de teste utilizando o comando: *ant run_tests*;

5) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Weka/lib*;

6) Fazer uma cópia dos bytecodes originais das classes que estão na pasta *Weka/Build/Classes* em uma nova pasta *Weka/Build/Classes.orig* de maneira que as duas fiquem com o mesmo conteúdo;

7) Criar uma pasta chamada *Weka/Build/Classes.inst* para guardar os bytecodes instrumentados;

8) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp ../lib/emma.jar emma instr -d classes.inst -ip classes.orig*;

9) Copiar os bytecodes da pasta *Weka/Build/Classes.inst* para pasta *Weka/Build/Classes*;

10) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant run_tests*;

11) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp lib/emma.jar emma merge -in build/coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp lib/emma.jar emma report -r txt,html,xml -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Weka:

Tabela 3.9: *Weka: resumo da cobertura obtida*

Classes % (Cobertas / Total)	49% (1099 / 2244)
Métodos % (Cobertos / Total)	35% (8164 / 23034)
Blocos % (Cobertos / Total)	37% (326494 / 874351)

Outras informações e problemas encontrados:

Analogamente ao que foi apresentado para o projeto HttpUnit, a execução dos Passos 6, 7, 8 e 9 visam a evitar erros de dependência no processo de execução dos casos de teste do Weka. Além disso, vale ressaltar que no processo de execução dos testes, muitos erros e *warnings* são apresentados, mesmo nas classes sem instrumentação, mas não há detalhes suficientes para se evitar tais problemas.

Xerces2

O Xerces2 consiste na segunda geração de parser XML (W3C, 2010) da família Xerces Apache (Apache Foundation, 2010g).

Versão analisada:

2.10

Endereço do repositório SVN:

http://svn.apache.org/repos/asf/xerces/java/trunk

Procedimentos realizados na execução do experimento:

1) Efetuar o download dos arquivos do repositório SVN e salvar em uma pasta chamada Xerces;

2) Compilar o código fonte utilizando o comando: *ant compile*;

3) Compilar os códigos de teste utilizando o comando: *ant tests*;

4) Executar os casos de teste utilizando o comando: *ant test*;

5) Efetuar o download da ferramenta Emma e copiar *Emma.jar* para a pasta *Xerces/tools*;

6) Alterar o arquivo *Xerces/build.xml* para acrescentar referência à ferramenta Emma no *classpath*. Para isso, a linha <*property name='jar.emma' value='emma.jar'/*> deve ser acrescentada como elemento filho do nó <*target name="init"*>;

7) Fazer uma cópia dos bytecodes originais das classes que estão na pasta *Xerces/build/classes* em uma nova pasta chamada *Xerces/build/classes.orig*;

8) Criar uma pasta chamada *Xerces/build/classes.inst* para guardar os bytecodes instrumentados;

9) Instrumentar os bytecodes com a ferramenta Emma utilizando o comando: *java -cp ../tools/emma.jar emma instr -d classes.inst -ip classes.orig*;

10) Copiar os bytecodes instrumentados da pasta *Xerces/build/classes.inst* para pasta *Xerces/build/classes*;

11) Alterar o arquivo *Xerces/build.xml* para excluir as comandos que compilam os códigos fonte e evitar que os bytecodes instrumentados sejam sobrescritos na execução dos teste. Para isso, deve-se alterar a dependência do *target samples* de *compile* para *init*;

12) Alterar o arquivo *Xerces/build.xml* para acrescentar referência à ferramenta Emma na execução dos testes. Para isso, o caminho *{tools.dir}/${jar.emma}${path.separator}* deve ser acrescentado no atributo *value* de todos os elementos *jvmarg* que estão no *target test*;

13) Executar novamente os casos de teste utilizando o seguinte o comando para que sejam coletados os dados de cobertura: *ant test*;

14) Gerar os relatórios de cobertura com a ferramenta Emma utilizando os comandos:

a. *java -cp tools/emma.jar emma merge -in build/coverage.em -in coverage.ec -out coverage.es*;

b. *java -cp tools/emma.jar emma report -r txt,xml,html -in coverage.es*.

Resumo dos dados de cobertura obtidos para o Xerces2:

Tabela 3.10: *Xerces2: resumo da cobertura obtida*

Classes % (Cobertas / Total)	46% (483 / 1050)
Métodos % (Cobertos / Total)	34% (3625 / 10786)
Blocos % (Cobertos / Total)	34% (124071 / 359941)

Outras informações e problemas encontrados:

Analogamente ao que foi apresentado para o projeto HttpUnit, a execução dos Passos 7, 8, 9 e 10 visam a evitar erros de dependência no processo de execução dos casos de teste do Xerces2.

3.2 Considerações sobre os resultados dos experimentos

Nesta seção são apresentados os resultados obtidos no processo de análise com a ferramenta Emma. Além disso, tem-se também outros dados que puderam ser coletados por meio de estudos mais detalhados nos relatórios obtidos em cada experimento. A Tabela 3.11, que esta organizada em ordem alfabética pelo nome dos projetos, apresenta a quantidade de classes e os dados de cobertura para cada projeto analisado.

Tabela 3.11: *Resumo das coberturas e quantidades de classes dos projetos analisados*

Projeto	**Classes**	% **Classes**	% **Métodos**	% **Blocos**
Canoo	286	85%	73%	67%
HttpUnit	330	96%	81%	83%
JFreeChart	514	72%	52%	47%
JMeter	836	80%	53%	46%
Log4j	245	36%	37%	38%
Mondrian	1605	39%	21%	23%
Poi	2582	34%	10%	10%
Velocity	221	86%	62%	61%
Weka	2224	49%	35%	37%
Xerces2	1050	46%	34%	34%

A Tabela 3.11 consiste em uma lista dos resultados de cobertura coletados nos experimentos. Para isso, optou-se pelas seguintes informações: **Classes** representa a quantidade de classes do projeto; **% Classes** representa a porcentagem de classes que foram cobertas pelos testes; **% Métodos** consiste na porcentagem de métodos que foram cobertos pelos testes; e **% Blocos** que representa a porcentagem de blocos (ou instruções de *bytecode*) que foram cobertos pelos testes.

Os resultados apresentados na Tabela 3.11 mostram que, dentre os dez produtos FLOSS analisados, somente um atinge o primeiro nível de cobertura que representa a cobertura de todos os comandos (na prática ele atingiu um percentual mais comumente aceito, conforme Cornett (2007) que estabelece um mínimo variando entre 70 e 80%). Mesmo projetos como JMeter, bastante utilizado pela comunidade, tem aproximadamente 54% de seu código fonte não executado pelos testes.

Os valores de cobertura apresentados na Tabela 3.11 são números gerais, ou seja, pode-se dizer que o número consiste apenas em uma representação das classes, métodos e blocos que tiveram "alguma" cobertura. Além dos dados obtidos com a ferramenta Emma, optou-se por realizar uma análise comparativa cujo resultado está descrito na Tabela 3.12. Tal análise consiste em um detalhamento da cobertura de blocos no nível pacote para cada um dos projetos analisados baseado no valor de 80% que, por sua vez, representa o valor que ferramenta Emma considera como cobertura mínima "aceitável".

A Tabela 3.12 possui as seguintes informações: **PK** consiste no número de pacotes; **NC** representa a quantidade de classes do projeto; **NB** consiste na quantidade de blocos (ou instruções de *bytecode*) computados para o projeto em questão; **CC > 0** representa a quantidade e a porcentagem de classes com alguma cobertura; **BC > 0** consiste na quantidade e na porcentagem de blocos com alguma cobertura; **BC** $\geq$ **th** representa a quantidade e a porcentagem de blocos com cobertura superior ao *threshold* da ferramenta Emma (i.e.: qualquer valor maior ou igual a 80% e menor ou igual a 100%)

Tabela 3.12: *Detalhamento das coberturas dos projetos analisados*

Projeto	PK	NC	NB	CC $\neq$ 0	BC $\neq$ 0	BC $\geq$ th	CC = 0	BC = 0
Canoo	29	286	40927	243 (85%)	27332 (67%)	10131 (24,75%)	43 (15%)	13595 (33%)
HttpUnit	11	330	43333	317 (96%)	36105 (83%)	33151 (76,50%)	13 (4%)	7228 (17%)
JFreeChart	40	514	209564	369 (72%)	97529 (47%)	61 (0,03%)	145 (28%)	112035 (53%)
JMeter	94	836	174011	667 (80%)	80546 (46%)	10080 (5,79%)	169 (20%)	93465 (54%)
Log4j	18	245	39216	88 (36%)	14971 (38%)	0 (0%)	157 (64%)	24245 (62%)
Mondrian	26	1605	283245	632 (39%)	64941 (23%)	2306 (0,81%)	973 (61%)	218304 (77%)
Poi	145	2583	856179	873 (34%)	84256 (10%)	29632 (3,46%)	1709 (66%)	771923 (90%)
Velocity	25	221	56031	189 (86%)	34448 (61%)	3150 (5,62%)	32 (14%)	21583 (39%)
Weka	93	2244	874351	1099 (49%)	326494 (37%)	38544 (4,41%)	1145 (51%)	547857 (63%)
Xerces2	82	1050	359941	483 (46%)	124071 (34%)	11800 (3,28%)	567 (54%)	235870 (66%)

no nível de pacote; **CC = 0** consiste na quantidade e na porcentagem de classes sem nenhuma cobertura; e **BC = 0** representa a quantidade e a porcentagem de blocos sem nenhuma cobertura.

No detalhamento dos resultados da análise, conforme ilustrado na Tabela 3.12, a porcentagem de código que possui cobertura superior ao *threshold* da ferramenta Emma no nível de pacote é "baixa" na maioria dos programas. Tal informação pode ser útil na priorização do desenvolvimento de casos de teste para os pacotes com menor quantidade de código exercitado. O projeto **HttpUnit** apresentou o melhor resultado dentre os projetos analisados, pois, além de apenas 4% de suas classes e 17% dos blocos não apresentarem nenhuma cobertura, 76,50% do código foi coberto com um valor acima do *threshold*.

O projeto **Canoo** apresentou um resultado regular quanto à cobertura de classes (apenas 13% sem cobertura), no entanto, 1/3 do blocos não tiveram nenhuma cobertura e, além disso, dos 67% dos blocos que tiveram alguma cobertura, somente 24,75% foram superior ao valor do *threshold*. O projeto **JMeter**, a exemplo do Canoo, também não teve um resultado ruim quanto à cobertura de classes (apenas 20% não foram cobertas), no entanto, mais da metade dos blocos (54%) tiveram cobertura igual a 0% e, além disso, dos 46% que tiveram alguma cobertura, somente 5,79% foram superior ao valor do *threshold*. O projeto **Velocity**, a exemplo do Canoo e do JMeter, também não teve um resultado ruim quanto à cobertura de classes (apenas 14% não tiveram nenhuma instrução executada), no entanto, 39% dos blocos tiveram cobertura igual a 0% e, além disso, dos 61% que tiveram alguma cobertura, somente 5,62% foram superior ao valor do *threshold*.

O projeto **Weka** teve mais da metade de suas classes (51%) e blocos (63%) sem nenhuma cobertura e, além disso, dos 37% de código que tiveram alguma cobertura, somente 4,41% foram superior ao valor do *threshold*. O projeto **Xerces2**, a exemplo do Weka, também teve mais de metade das suas classes (54%) e blocos (66%) sem nenhuma cobertura e, além disso, dos 46% que tiveram alguma cobertura, somente 3,28% foram superior ao valor do *threshold*.

O projeto **JFreeChart** não foi o que apresentou pior cobertura de classes (28%) e

blocos (53%) sem nenhuma cobertura, no entanto, dos 47% de código que tiveram alguma cobertura, somente 0,03% foram superior ao valor do *threshold*.

O projeto **Mondrian** figura entre os piores resultados no que diz respeito à classes (61%) e blocos (77%) sem nenhuma cobertura e, além disso, apresentou o terceiro pior resultado para blocos que estão em pacotes com valores acima do *threshold*, ou seja, dos 23% que apresentaram alguma cobertura, apenas 0,81% estão em pacotes que apresentam cobertura de blocos superior a 80%.

O projeto **Poi** apresentou o pior resultado no que diz respeito à classes (66%) e blocos (90%) sem nenhuma cobertura e além disso, apresentou o quarto pior resultado para blocos que estão em pacotes com valores acima do *threshold*, ou seja, dos 10% que apresentaram alguma cobertura, apenas 3,46% estão em pacotes que apresentam cobertura de blocos superior a 80%.

O projeto **Log4j** apresentou os piores resultados tanto na cobertura de classes (64%) e blocos (62%) sem nenhuma cobertura e, além disso, dos 38% de código que tiveram alguma cobertura todos ficaram abaixo do *threshold*.

Baseado nos estudos apresentados, nas fundamentações teóricas e nos resultados obtidos nos experimentos, é proposta uma estratégia de teste incremental visando a auxiliar a comunidade de desenvolvimento dos produtos de software livre a melhorarem a qualidade dos conjuntos de teste desenvolvidos, em função das métricas de cobertura de código de critérios estruturais utilizada. Tal estratégia é descrita na seção seguinte.

3.3 Estratégia incremental

Um dos objetivos deste trabalho, conforme descrito na Seção 1.2, consiste em apresentar uma proposta de estratégia para melhoria dos conjuntos de teste das comunidades de desenvolvimento de FLOSS. Conforme foi apresentado neste trabalho, principalmente nas Seções 1.1 e 2.3.2, considera-se que, na maioria dos casos, as comunidades FLOSS realizam um processo de teste de software *ad hoc*.

Neste contexto, com intuito de contribuir para melhoria contínua dos conjuntos de casos de teste (haja visto que o objeto principal deste trabalho são os conjuntos de teste e não o processo de teste como um todo), propõe-se uma estratégia incremental para evolução de tais conjuntos com base nos níveis de cobertura de código proposto por Copeland (2004).

A proposta de estratégia é dita incremental pois sugere que os conjuntos de teste "sofram" um processo de melhoria contínua e possam evoluir nível a nível dentro da escala, conforme descrito na Seção 2.2.3. A Figura 3.5 ilustra como seria a execução da estratégia no processo de teste de software.

Figura 3.5: *Execução da estratégia no processo de teste*

O ciclo visa a representar que o processo deverá ser executado até que determinado objetivo de teste seja atingido. As fases que seriam executadas na estratégia estão divididas em: a) desenvolvimento dos casos de teste; b) avaliação da cobertura e dos critérios de teste obtidos pelo conjunto de teste desenvolvido; c) análise dos resultados obtidos na realização da avaliação da cobertura; e d) determinação do nível de cobertura atingido conforme os níveis definidos por Copeland (2004).

Desse modo, espera-se que a aplicação da estratégia de teste de software pelas comunidades FLOSS possa contribuir para: elaboração de casos de testes; determinar se o produto foi ou não suficientemente testado; e atribuição de níveis de rigor aos conjuntos de teste.

Para contribuir com esta estratégia, tendo em vista que ela está sendo proposta dentro do contexto de projetos FLOSS, optou-se por sugerir um processo de evolução contínua do conjunto de teste que empregue o conceito distribuído adotado para o desenvolvimento do FLOSS também para o registro, coleta e envio de informações de teste realizadas por membros ou usuários de um determinado produto.

Existe a suposição de que os produtos FLOSS são “testados” pelos usuários em ambientes reais. Este ponto é apoiado em parte pelo argumento de Raymond (2000) de que “dados olhos suficientes, todos os defeitos são triviais”. Por isso, seria importante desenvolver mecanismos que permitam o registro da cobertura dos testes à medida que os usuários utilizam um FLOSS no seu dia a dia ou com o objetivo específico de contribuir com o teste do produto. Neste contexto, “testes do usuário” que exercitam partes não cobertas poderiam ser registrados, enviados e, posteriormente, incorporados ao conjunto de testes de determinado FLOSS de forma automática ou semi-automática. A Figura 3.6 ilustra algumas possibilidades de contribuições e uso de FLOSS.

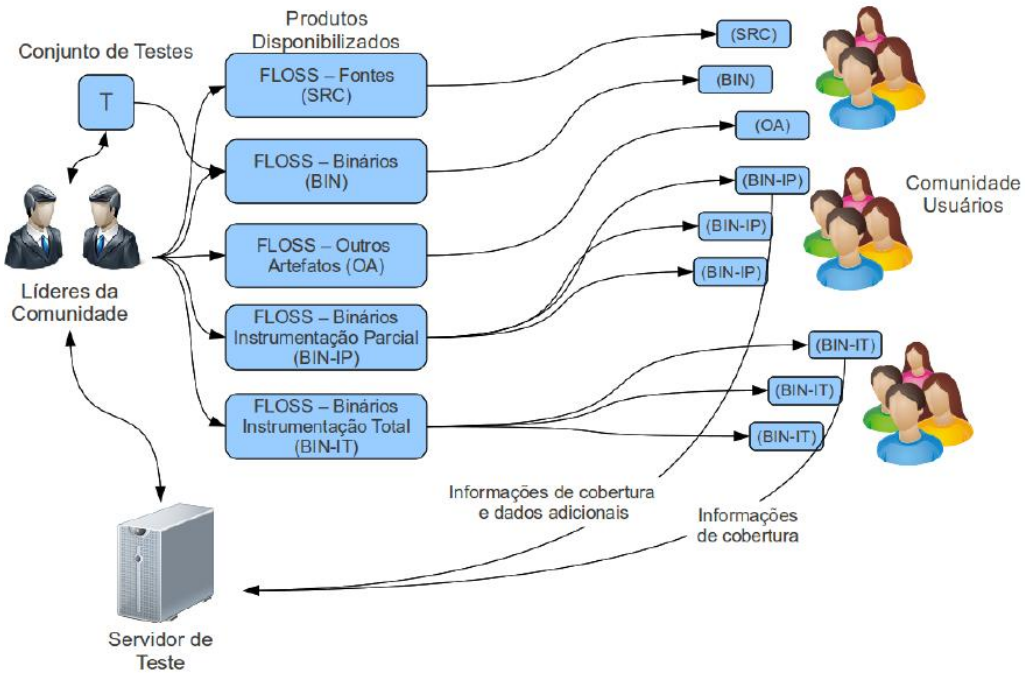


Figura 3.6: *Modelo distribuído para auxiliar na evolução contínua do conjunto de teste*

Como pode ser observado, os líderes da comunidade de desenvolvimento mantém o repositório com o conjunto de testes (**T**), código fonte do produto (**SRC**), código binário (**BIN**) e outros artefatos (**OA**). Para apoiar a estratégia proposta, os líderes do projeto da comunidade FLOSS poderiam disponibilizar uma versão instrumentada do código para aqueles usuários que gostariam de contribuir com a evolução do conjunto de testes de um determinado produto. Nesse sentido, com base nas informações de cobertura dos testes, uma versão dos binários seria disponibilizada com instrumentação parcial, ou seja, a instrumentação realizada apenas nas pastes do binário ainda não executadas pelos casos de testes (**BIN-IP**). O software que estiver sendo utilizando neste contexto, por usuários comuns ou por desenvolvedores, teria seus dados de entrada e sua cobertura de código monitoradas e armazenadas localmente. À medida que partes do software ainda não executadas pelo conjunto de teste original fossem cobertas com base nas entradas fornecidas pelo usuário, este seria alertado sobre o acontecimento e indagado se autoriza o envio dos dados de entrada fornecidos para o servidor de teste da comunidade que gerencia o projeto, contribuindo, dessa forma, para a melhoria e evolução do conjunto de testes disponibilizados e do produto em si. A seguir, essa estratégia é detalhada em quatro passos:

1. Identificar usuários dispostos a utilizar o software com instrumentação residual, isto é, versões do software instrumentada apenas em pontos do programa ainda não cobertos (**BIN-IP**);

2. Armazenar os dados de entrada fornecidos e o caminho dos arquivos de entrada de dados utilizados pelo usuário a cada execução do software;

3. Solicitar autorização do usuário para envio dos dados coletados para o Servidor de Teste da comunidade, bem como o resultado da execução (bem sucedida ou não), toda vez que partes ainda não executadas fossem cobertas. No caso de o software utilizar dados complexos (imagens, arquivos, etc), é necessário solicitar que o usuário autorize e forneça também esses dados;

4. Os novos casos de teste serão analisados pela equipe do projeto para inclusão no conjunto de testes oficial (**T**).

A implementação dessa estratégia depende da solução de problemas técnicos e não-técnicos. O primeiro problema é a criação de uma infraestrutura para coleta dos dados fornecidos por usuários, pois, dependendo da popularidade dos projetos, o acesso a essa infraestrutura pode ser grande.

Outra questão técnica é que a instrumentação residual não pode impactar o uso do sistema sobremaneira. Critérios como os utilizados pela ferramenta EMMA em geral não causam impacto (aumento no uso de memória e no tempo de execução) significativo para aplicações do tipo "desktop". Porém, critérios baseados em fluxo de dados (RAPPS; WEYUKER, 1982), como os implementados nas ferramentas POKETOOL (CHAIM, 1991) e JaBUTi (VINCENZI et al., 2010), podem comprometer o desempenho de produtos FLOSS para programas de longa duração. Nesses casos, à medida que critérios de testes mais rigorosos forem demandados, formas mais eficientes de instrumentação terão que ser desenvolvidas para viabilizar o emprego dessa estratégia.

Do ponto de vista não-técnico, um possível problema seria identificar usuários dispostos a abrir mão da sua privacidade para contribuir para o projeto. Isto porque o nível de monitoramento requerido para a implementação dessa estratégia é bastante detalhado e é preciso avaliar quantos usuários estariam dispostos a abrir mão da sua privacidade.

Outro problema é que a equipe de desenvolvimento poderá ser sobrecarregada uma vez que, ao receber novos possíveis casos de teste, estes devem ser analisados para validar os dados de entrada e de saída para inclusão no conjunto de testes oficial. O problema é que pode ser que uma cobertura já tenha sido obtida, mas usuários com versões anteriores continuem a enviar novos casos de teste para a equipe de desenvolvimento. Nesse caso, um esquema de atualização automática da versão instrumentada pode ser desenvolvido, ou então, casos de testes recebidos e que executem comandos já cobertos por outros casos de testes existentes podem ser tratados com menos prioridade, aliviando o trabalho de análise e melhoria do conjunto de teste oficial.

Uma segunda estratégia seria a instrumentação total do código fonte (**BIN-IT**). Esse caso seria ideal para aqueles usuários da comunidade que não desejam ter suas informações pessoais, dados de entrada ou outros arquivos submetidos para o Servidor de Teste. Nesse modelo de instrumentação, apenas dados de cobertura seriam enviados ao servidor de teste. Nenhuma informação de como o usuário atingiu aquela cobertura seria disponibilizada. Os dados de cobertura, nesse caso, seriam utilizados pelos líderes da comunidade para priorizar o desenvolvimento do conjunto de teste uma vez que, em princípio, o conjunto de teste (**T**) é desenvolvido sem o conhecimento do perfil operacional de seus usuários. Com esse modelo de instrumentação seria possível, de forma rápida e eficiente, conhecer, do ponto de vista do usuário, quais as principais partes do software que são demandadas e, desse modo, os líderes poderiam concentrar o desenvolvimento dos testes priorizando essas partes em detrimento a outras menos utilizadas.

Além de contribuir para elaboração de casos de teste, acredita-se que a utilização da estratégia poderá auxiliar a determinar se o produto foi ou não suficientemente testado, pois à medida que os níveis de cobertura vão sendo "atingidos" pelo conjuntos de teste, critérios de teste cada vez mais exigentes vão sendo exercitados. Além disso, o fato de ser possível classificar os conjuntos de teste por nível pode ser considerado como um fator de determinação da qualidade do conjunto de teste, em relação a métricas de cobertura de código.

Dentro da perspectiva do ciclo de melhoria contínua apresentado na Figura 3.5, mas ainda sem considerar a aplicação da proposta de evolução contínua do conjunto de testes Figura 3.6, pode-se considerar que este trabalho representa a primeira análise dos casos de teste. Sendo assim, decorrente dos resultados apresentados na Seção 3.2, entende-se que a maioria dos conjuntos de teste analisados estão no nível 0 de Copeland (2004), pois sequer atingiram a cobertura de todos os comandos.

Na prática, exigir 100% de cobertura de código é impraticável. Segundo Cornett (2007), um nível de cobertura razoável estaria entre 70 a 80% para a maioria dos produtos de software. Assim, uma alternativa para iniciar a aplicação da estratégia seria definir um nível de cobertura de código que varie dentro desse intervalo.

Para a continuação do processo de melhoria dos conjuntos de casos de teste, ou seja, a partir do nível 1 de (COPELAND, 2004) em que são exigidos 100% de cobertura de critérios estruturais, sugere-se a definição de metas a serem atingidas e que possam ser aumentadas à medida que o conjunto de teste evolui.

A estratégia de teste pode ser aplicada a todas as fases de teste (unidade, integração e sistemas), mas as exigências de cobertura podem ser minimizadas para testes de integração e de sistemas, pois, segundo Cornett (2007), quanto menor o código, mais fácil será obter altos níveis de cobertura. Além disso, é importante frisar que as metas

de cobertura que serão definidas podem variar para cada tipo de projeto, pois há normas como, por exemplo, a DO-178B (RTCA, 1992) e ANSI / IEEE 1008-1997 (IEEE, 1987) que exigem cobertura de 100% de código para softwares de segurança crítica.

Na maioria dos projetos de aplicações não críticas, o nível de exigência de cobertura pode ser menos rigoroso. Desse modo, a comunidade de desenvolvimento pode priorizar atingir níveis de cobertura de 100% apenas para um subconjunto das classes dos projetos, exatamente aquelas que implementam as funcionalidades críticas da aplicação em questão, ou que atendam as funcionalidades do perfil operacional padrão do produto de software. Com o passar do tempo, outras classes podem ser atacadas visando, a longo prazo, a atingir 100% de cobertura de todo o projeto.

Do ponto de vista de teste estrutural mais importante do que saber o que foi executado durante os testes é saber o que não foi executado, pois, sem essa informação, partes essenciais ou críticas da aplicação que foram mal testadas não podem ser identificadas, trazendo transtornos indesejáveis para os usuários da aplicação e colocando em risco a credibilidade do produto de software ou de sua comunidade de desenvolvimento.

CAPÍTULO 4

Conclusões

Estabelecer elementos que possam garantir qualidade aos produtos FLOSS pode ser considerado como tema emergente nas pesquisas da área de software de código aberto. Neste contexto, vários aspectos dos produtos dessa natureza estão sendo investigados, como: aspectos legais, modelo de negócio, processo de desenvolvimento e qualidade dos produtos.

No processo de investigação da qualidade dos produtos, análises estáticas e dinâmicas do código fonte são realizadas com intuito de identificar o estado atual desses códigos disponibilizados pelas comunidades. Especificamente com relação à análise dinâmica, a investigação está voltada para avaliação da qualidade dos conjuntos de testes criados pelas comunidades de desenvolvimento de FLOSS utilizando, para isso, critérios de testes estruturais.

A partir das análises com relação à parte dinâmica em dez produtos FLOSS, deu-se início a identificação do estado da prática da comunidade na criação de conjuntos de teste e pode-se propor uma estratégia de teste visando a contribuir para a evolução dos conjuntos existentes.

Os resultados encontrados na investigação dos conjuntos de teste por meio de critérios estruturais mostraram que, dentre os dez produtos FLOSS analisados, somente um atinge o primeiro nível de cobertura que representa a cobertura de todos os comandos (na prática ele atingiu um percentual mais comumente aceito que estabelece um mínimo variando entre 70 e 80%). Ressalta-se que este resultado não define a qualidade do produto, ou seja, não se pode afirmar que os produtos possuem ou não qualidade apenas avaliando-o do ponto de vista dinâmico, mensurando a cobertura de seus conjuntos de teste. Sabe-se que várias comunidades empregam outras técnicas de V&V, além dos testes, para assegurar a qualidade de seus produtos e, tais aspectos estão fora do escopo desta dissertação. Neste caso, a conclusão obtida neste trabalho é que o conjunto de teste disponibilizado pela comunidade alcança somente o primeiro nível de cobertura e, por isso, propõe-se aplicar uma estratégia para evolução dos conjuntos de teste.

Desse modo, a Seção 4.1 sintetiza as principais contribuições deste trabalho. A Seção 4.2 apresenta os artigos produzidos durante o desenvolvimento desta dissertação.

Finalmente, a Seção 4.3 descreve possíveis desdobramentos e trabalhos futuros.

4.1 Contribuições

Nesta seção, apresenta-se um resumo das principais contribuições identificadas durante o desenvolvimento desta dissertação:

- Descrição de um processo de análise de cobertura de código para produtos FLOSS desenvolvidos em Java com o detalhamento dos procedimentos utilizados na análise de cada um deles. O detalhamento dos procedimentos pode ser útil na repetição dos experimentos, bem como na realização de analises em outras versões do mesmo produto e/ou na criação de *scritps* que generalizam o processo de coleta dos dados;

- Apresentação dos dados de cobertura obtidos na avaliação dos produtos FLOSS. Os resultados podem ser utilizados como informações iniciais para um ciclo de melhoria contínua dos conjuntos de teste;

- Proposta de uma estratégia de teste incremental baseada em critérios estruturais e um processo de evolução contínua que visa a contribuir para a evolução dos conjuntos de teste disponibilizados pelas comunidades FLOSS.

4.2 Produção Bibliográfica

Nesta seção, apresentam-se os artigos publicados e apresentados durante o desenvolvimento desta dissertação. Para cada artigo, tem-se: o título; o resumo (ou *abstract*) extraído do original para facilitar a identificação da relação com esta dissertação); e o evento no qual foi publicado e apresentado.

Artigos Apresentados em evento local:

- **Título**: QualiPSo: Plataforma de Qualidade para Software de Código Aberto (RINCON; VINCENZI, 2009a)
  Resumo: Este artigo apresenta uma breve descrição do projeto QualiPSo, composto por 18 membros fundadores na Europa, Brasil e China. O projeto conta com representantes da indústria, órgãos governamentais, academia e comunidades de Software de Código Aberto. O artigo também discute uma das atividades do projeto, a qual visa obter informações sobre produtos de Software de Código Aberto de modo a definir quais requisitos de qualidade são importantes no estabelecimento de confiança (*trustworthiness*) a esses produtos. Para isso, o artigo mostra como será

a realização dos testes relacionados à essa atividade, quais os Softwares de Código Aberto que serão testados e o método que será seguido na realização dos testes.
Evento: Jornada de Pesquisa do Instituto de Informática, Goiânia, Brasil. Junho, 2009.

- **Título**: Quality evaluation of Open Source Software using Structural Testing Criteria (RINCON; VINCENZI, 2009b)
  Abstract: This paper presents a proposal for quality evaluation of Open Source Software. This evaluation will be analyzed structural testing criteria (e.g. all nodes, all edges, all uses and all potential uses) using the JaBUTi tool. The results of that evaluation will be used in the project QualiPSo and will assist in a definition of a test strategy based on structural criteria to ensure quality in software products open source.
  Evento: Jornada de Pesquisa do Instituto de Informática, Goiânia, Brasil. Outubro, 2009.

Artigo Publicado e Apresentado em evento nacional:

- **Título**: Avaliando a Qualidade de Conjuntos de Teste de Software de Código Aberto por meio de Critérios de Teste Estruturais (ROCHA et al., 2010)
  Abstract: QualiPSo (Quality Platform for Open Source Software) is an international project investigating Free/Libre/Open Source Software (FLOSS) products to define quality requirements that are important in establishing the trustworthiness of these products. One of the supported activities of QualiPSo aims at evaluating the quality of the test sets developed by the FLOSS community. This paper presents the results of employing structural testing criteria as a measure of quality for such a test sets aiming at identifying the state-of-practice played by the FLOSS community and contributing to the establishment of an incremental testing strategy to evolve the test sets.
  Evento: XI Workshop de Software Livre (WSL 2010), Porto Alegre, Brasil. Julho, 2010.

4.3 Trabalhos Futuros

A partir do levantamento bibliográfico, dos experimentos realizados e dos resultados obtidos nas análises descritas nesta dissertação, sugerem-se os seguintes trabalhos futuros:

- Avaliar diferentes versões de um mesmo produto FLOSS para verificar se há uma evolução nos dados de cobertura visando a identificar se o progresso dos conjuntos

de teste ocorre em paralelo à evolução dos códigos fonte;

- Utilizar as informações coletadas no item anterior visando a identificar um padrão de priorização dos testes adotado pela comunidade de código livre e tentar associar esse padrão com métricas de software, como complexidade, acoplamento e coesão;

- Aplicar a estratégia de teste proposta em uma comunidade FLOSS para verificar os benefícios, desvantagens e aspectos práticos para consolidação da estratégia;

- Desenvolvimento de um pacote de experimentação sobre o assunto desta dissertação visando a facilitar a replicação do experimento aqui apresentado, neste caso, se sugere a adaptação dos "builds" dos projetos analisados à estrutura do Mavem para possibilitar a integração com uma ferramenta de apoio à integração contínua como o Hudson (ORACLE, 2010);

- Criar uma infraestrutura baseada em serviços Web para coletar as informações e investigar como realizar a instrumentação com o menor custo possível para o processo de evolução contínua do conjunto de teste. Após a implementação da infraestrutura, esse processo será investigado por meio de experimentação;

- Utilizar ferramentas automatizadas que apoiem outros critérios estruturais, além da cobertura de comandos, para obter um diagnóstico dos conjuntos de teste visando a melhoria continua.

Referências Bibliográficas

ANDERSEN, E. *BusyBox*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.busybox.net/>.

Apache Foundation. *Apache: http server project*. 2010a. Página Web - último acesso em Agosto de 2010. Disponível em: <http://httpd.apache.org/>.

Apache Foundation. *Apache JMeter*. 2010b. Página Web - último acesso em Agosto de 2010. Disponível em: <http://jakarta.apache.org/jmeter/>.

Apache Foundation. *The Apache Velocity Project*. 2010c. Página Web - último acesso em Agosto de 2010. Disponível em: <http://velocity.apache.org/>.

Apache Foundation. *Apache log4j*. 2010d. Página Web - último acesso em Agosto de 2010. Disponível em: <http://logging.apache.org/log4j/>.

Apache Foundation. *The Apache Ant Project*. 2010e. Página Web - último acesso em Setembro de 2010. Disponível em: <http://ant.apache.org/>.

Apache Foundation. *The Apache POI Project - the Java API for Microsoft Documents*. 2010f. Página Web - último acesso em Outubro de 2010. Disponível em: <http://poi-.apache.org/>.

Apache Foundation. *Xerces2 Java Parser*. 2010g. Página Web - último acesso em Outubro de 2010. Disponível em: <http://xerces.apache.org/xerces2-j/>.

Apache Foundation. *Apache Maven Project*. 2010h. Página Web - último acesso em Outubro de 2010. Disponível em: <http://maven.apache.org/>.

BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Teste estrutural. In: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (Ed.). *Introdução ao teste de software*. [S.l.]: Rio de Janeiro: Elsevier, 2007. p. 47–76.

BERG, A. *Validate Your Functional Tests With These Code Coverage Tests For Java*. 2007. Artigo em revista digital - último acesso em Março de 2007. Disponível em: <http://www.stpmag.com/>.

BROOKS JR., F. P. No silver bullet essence and accidents of software engineering. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, p. 10–19, April 1987. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.1987.1663532>.

Canoo Web Test. *Canoo WebTest: free testing for web apps*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://webtest.canoo.com/>.

CHAIM, M. J. *Potential Uses Criteria Tool for Program Testing*. 1991. Dissertação de Mestrado, DCA/FEE/UNICAMP - Campinas, SP, Brasil.

COPELAND, L. *A practitioner's guide to software test design*. [S.l.]: Artech House Publishers, 2004.

CORNETT, S. *Minimum Acceptable Code Coverage*. 2007. Artigo online - último acesso em Outubro de 2010. Disponível em: <http://www.bullseye.com/minimum.html>.

CRESPO, A. N.; SILVA, O. J. da; BORGES, C. A.; SALVIANO, C. F.; JUNIOR, M. de Teive e A.; JINO, M. Uma metodologia para teste de software no contexto da melhoria de processo. In: *Anais do III Simpósio Brasileiro de Qualidade de Software*. [S.l.]: SBC Sociedade Brasileira de Computação, 2004. ISBN 858844276-0.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Conceitos básicos. In: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (Ed.). *Introdução ao teste de software*. [S.l.]: Rio de Janeiro: Elsevier, 2007. p. 1–7.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. *Object Oriented Reengineering Patterns*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. Foreword By-Johnson, Ralph E. ISBN 1558606394.

ECLIPSE. *Eclipse*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.eclipse.org/>.

ELBAUM, S.; GABLE, D.; ROTHERMEL., G. The impact of software evolution on code coverage information. In: *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, 2001. p. 170–179. ISBN 0-7695-1189-9.

FALCãO, J.; JUNIOR, T. S. F.; LEMOS, R.; MARANHãO, J.; SOUZA, C. A. P. de; SENNA, E. *Estudo sobre o Software Livre*. 2005. Estudo da Escola de Direito da Fundação Getúlio Vargas - Rio de Janeiro - comissionado pelo Instituto Nacional de Tecnologia da Informação disponível na Web - último acesso em Setembro de 2010. Disponível em: <http://www.softwarelivre.gov.br/publicacoes/Estudo_FGV.pdf>.

FRANKL, P. G.; WEYUKER, E. J. A data flow testing tool. In: *Proceedings of the second conference on Software development tools, techniques, and alternatives*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985. p. 46–53. ISBN 0-89791-173-3.

FSF. *Site Oficial da Fundação para o Software Livre*. 2010. Página Web - último acesso em Setembro de 2010. Disponível em: <http://www.fsf.org/>.

GOLD, R. *HttpUnit*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://httpunit.sourceforge.net/>.

GRUBER, H.; KöRNER, C.; PLöSCH, R.; POMBERGER, G.; SCHIFFER, S. Benchmarking-oriented analysis of source code quality: experiences with the qbench approach. In: *Proceedings of the IASTED International Conference on Software Engineering*. Anaheim, CA, USA: ACTA Press, 2008. (SE '08), p. 7–13. ISBN 978-0-88986-716-1. Disponível em: <http://portal.acm.org/citation.cfm?id=1722603.1722606>.

HALSTEAD, M. H. *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN 0444002057.

IEEE. *IEEE Standard for Software Unit Testing - ANSI/IEEE Std 1008-1987*. 1987. IEEE Computer Society Press.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. [S.l.], 2002.

IEEE. *Software Engineering Body of Knowledge (SWEBOK)*. [s.n.], 2004. Disponível em: <http://www.swebok.org/>.

Jasper Forge. *Jasper Report: Open Source Java Reporting Library*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://jasperforge.org/>.

JBOSS. *Hibernate: Relational Persistence for Java and .NET*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.hibernate.org/>.

JFree Community. *JFreeChart*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.jfree.org/jfreechart/>.

JUNIT. *JUnit*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.junit.org/>.

LAPRéVOTE, A.; VINCENZI, A. M. R.; CHAIM, M. L.; SILVA, M. A. G.; TOSI, D.; HUI.WU; KRYSZTOFIAK, T. *Working Document 5.4.2 - Test suites and benchmarks for the chosen set of Open Source projects and artifacts. Methodology for creating test suites and benchmarks for arbitrary systems.* [S.l.], 2009.

LUCENA, F. *Kyrios Ambiente de Desenvolvimento (Kad)*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://kyrios.sourceforge.net/kad.html>.

MALDONADO, J. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese (Doutorado) — DCA/FEE/UNICAMP - Campinas, SP, Brasil, 1991.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. R. S.; DELAMARO, M. E. *Aspectos teóricos e empíricos de teste de cobertura de software*. 1998. Relatório Técnico 31, ICMC-USP.

MCCABE, T. J. A complexity measure. In: *Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. (ICSE '76), p. 407–. Disponível em: <http://portal.acm.org/citation.cfm?id=800253-.807712>.

MCCONNELL, S. Open-source methodology: Ready for prime time? *IEEE Software*, v. 16, n. 4, p. 6–8, 1999.

MEIRELLES, F. S. *Pesquisa Anual - Administração de Recursos de Informática - CIA - Centro de Informática Aplicada*. [S.l.]: FGV/EAESP, 2003.

MEIRELLES, P.; JR., C. S.; MIRANDA, J.; KON, F.; TERCEIRO, A.; CHAVEZ, C. A study of the relationships between source code metrics and attractiveness in free software projects. In: *Anais do XXIV Simpósio Brasileiro de Engenharia de Software (SBES 2010)*. [S.l.: s.n.], 2010.

MENS, T.; WERMELINGER, M.; DUCASSE, S.; DEMEYER, S.; HIRSCHFELD, R.; JAZAYERI, M. Challenges in software evolution. In: *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2005. p. 13–22. ISBN 0-7695-2349-8.

MIDHA, V. Does complexity matter? the impact of change in structural complexity on software maintenance and new developers' contributions in open source software. In: *Proceedings of the 15th International Conference on Information Systems*. [s.n.], 2008. (ICIS 2008). Disponível em: <http://aisel.aisnet.org/icis2008/37>.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. A case study of open source software development: the apache server. In: *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM, 2000. p. 263–272. ISBN 1-58113-206-9.

MOONEN, L.; DEURSEN, A. van; ZAIDMAN, A.; BRUNTINK, M. Software evolution. In: MENS, T.; DEMEYER, S. (Ed.). *chapter The interplay between software testing and software evolution*. [S.l.]: Springer, 2008.

MORASCA, S.; TAIBI, D.; TOSI, D. Towards certifying the testing process of open-source software: New challenges or old methodologies? In: *Emerging Trends in Free/Libre/Open Source Software Research and Development, 2009. FLOSS '09. ICSE Workshop on*. [S.l.: s.n.], 2009. p. 25–30.

Novell, Inc. *OpenSuse*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.opensuse.org/>.

ORACLE. *Hudson: Extensible continuous integration server*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://hudson-ci.org/>.

OSI. *The Open Source Definition*. 2010. Página Web - último acesso em Setembro de 2010. Disponível em: <http://opensource.org/docs/osd>.

Pentaho Corporation. *Pentaho Analysis Services (Mondrian)*. 2010. Página Web - último acesso em Outubro de 2010. Disponível em: <http://mondrian.pentaho.com/>.

PEPPLER, M. *BugDB*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.peppler.org/bugdb/>.

PETRINJA, E.; NAMBAKAM, R.; SILLITTI, A. Introducing the opensource maturity model. In: *Emerging Trends in Free/Libre/Open Source Software Research and Development, 2009. FLOSS '09. ICSE Workshop on*. [S.l.: s.n.], 2009. p. 37–41.

PLOESCH, R.; GRUBER, H.; KOERNER, C.; SAFT, M. A method for continuous code quality management using static analysis. *2010 Seventh International Conference on the Quality of Information and Communications Technology*, Ieee, p. 370–375, 2010. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper-.htm?arnumber=5655663>.

PRESSMAN, R. S. *Engenharia de Software*. [S.l.]: McGraw-Hill, 2006.

QBENCH. *QBench-Projekt*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.qbench.de/>.

QUALIPSO. *Relatório técnico com a descrição oficial do projeto QualiPSo: Part B.* [S.l.], 2005.

QUALIPSO. *Quality Platform for Open Source Software: Trust and Quality in Open Source Systems - Página Oficial do Projeto QualiPSo*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.qualipso.org/>.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: *Proceedings of the 6th international conference on Software engineering*. Los Alamitos,

CA, USA: IEEE Computer Society Press, 1982. (ICSE '82), p. 272–278. Disponível em: <http://portal.acm.org/citation.cfm?id=800254.807769>.

RAYMOND, E. S. *The cathedral and the bazaar*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2000. ISBN 0-596-01008-8.

RINCON, A. M.; VINCENZI, A. M. R. Qualipso: Plataforma de qualidade para software de código aberto. I Jornada de Pesquisa do Instituto de Informática, Goiânia, GO, BRA, 2009.

RINCON, A. M.; VINCENZI, A. M. R. Quality evaluation of open source software using structural testing criteria. II Jornada de Pesquisa do Instituto de Informática, Goiânia, GO, BRA, 2009.

ROCHA, A.; RINCON, A. M.; DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Avaliando a qualidade de conjuntos de teste de software de código aberto por meio de critérios de teste estruturais. XI Workshop de Software Livre (WSL 2010), Porto Alegre, RS, BRA, 2010.

ROUBTSOV, V. *EMMA: a free Java code coverage tool*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://emma.sourceforge.net/>.

RTCA. *Software Considerations in Airborne Systems and Equipment Certification Radio Technical Commission for Aeronautics*. 1992. Technical Report, RTCA/DO-178B.

RUNESON, P. A survey of unit testing practices. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 4, p. 22–29, 2006. ISSN 0740-7459.

Software Research, Inc. *Test-Coverage Analysis Tool*. 2010. Página Web - último acesso em Setembro de 2010. Disponível em: <http://www.soft.com/TestWorks/index.html>.

STALLMAN, R. *Why Open Source misses the point of Free Software*. 2010. Página Web - último acesso em Setembro de 2010. Disponível em: <http://www.gnu.org/philosophy-/open-source-misses-the-point.html>.

STAMELOS, I.; ANGELIS, L.; OIKONOMOU, A.; BLERIS, G. L. Code quality analysis in open source software development. *Information Systems Journal*, v. 12, n. 1, p. 43–60, 2002.

SUBVERSION. *Subversion: the world's most popular open source version control system*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://subversion.tigris.org/>.

Sun Microsystems. *Java*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.java.com/>.

VINCENZI, A.; MONACO, F.; DELAMARO, M. *JaBUTi - Java Bytecode Understanding and Testing*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://jabuti.incubadora.fapesp.br/portal>.

VINCENZI, A. M. R. *Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação*. Tese (Doutorado) — USP - São Carlos, SP, Brasil, 2004.

W3C. *Extensible Markup Language (XML)*. 2010. Página Web - último acesso em Setembro de 2010. Disponível em: <http://www.w3.org/XML/>.

WEKA. *Weka: Data Mining Software in Java*. 2010. Página Web - último acesso em Agosto de 2010. Disponível em: <http://www.cs.waikato.ac.nz/ml/weka/>.

ZAIDMAN, A.; ROMPAEY, B. V.; DEMEYER, S.; DEURSEN, A. v. Mining software repositories to study co-evolution of production & test code. In: *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008. p. 220–229. ISBN 978-0-7695-3127-4.

ZHAO, L.; ELBAUM, S. Quality assurance under the open source development model. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 66, n. 1, p. 65–75, 2003. ISSN 0164-1212.