# Avaliando a Qualidade de Conjuntos de Teste de Software de Código Aberto por meio de Critérios de Teste Estruturais*

**Adriana Rocha[1], André Mesquita Rincon[1, 2, 3], Márcio Eduardo Delamaro[4], José Carlos Maldonado[4] e Auri Marcelo Rizzo Vincenzi[1]**

[1]Instituto de Informática – INF/UFG – Goiânia, GO
*{adriana,andrerincon,auri}@inf.ufg.br*

[1]Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas
Universidade do Tocantins – UNITINS – Palmas, TO

[3]Instituto Federal de Educação, Ciência e Tecnologia do Tocantins – IFTO –
Paraíso do Tocantins, TO

[4]Instituto de Ciências Matemáticas e de Computação – ICMC/USP – São Carlos, SP –
*{delamaro, jcmaldon}@icmc.usp.br*

***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.*

## 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 2008]. No entanto, ainda há uma relutância quanto à adoção massiva de FLOSS devido, principalmente, à falta de "confiança" nesses produtos [QualiPSo 2008]. Segundo Meirelles [Meirelles 2003], em termos quantitativos, a utilização de FLOSS ainda é expressivamente inferior à utilização de alternativas proprietárias.

A falta de "confiança" está relacionada a aspectos jurídicos, de qualidade (por exemplo: ciclo de desenvolvimento, suporte, confiabilidade e desempenho) e também por um modelo de negócio que possa garantir sustentabilidade. Neste contexto, o QualiPSo [QualiPSo 2008] é considerado um projeto integrado que visa consolidar a aliança natural da indústria e do gover-

* *Este trabalho conta com o apoio do CNPq (Processo 570917/2008-5), CAPES e Projeto QualiPSo -Quality Platform for Open Source Software (Grant Number IST-FP6-IP-034763).*

no com FLOSS provendo o nível de qualidade que a indústria necessita relacionado a aspectos legais e de modelos de negócio.

Este trabalho está inserido no contexto deste projeto e atua em uma das atividades do projeto que visa obter informações sobre FLOSS de modo a "encontrar" requisitos de qualidade que possam ser importantes no estabelecimento de confiança (*trustworthness*) a esses produtos. Neste contexto, o trabalho está voltado para avaliação da qualidade dos conjuntos de testes criados pelas comunidades de desenvolvimento de FLOSS utilizando, para isso, critérios de testes estruturais. O objetivo é identificar o estado da prática da comunidade na criação de conjuntos de teste e, posteriormente, contribuir no estabelecimento de uma estratégia de teste que contribua para a evolução dos conjuntos de teste existentes.

O texto está organizado da seguinte forma: a Seção 2 dá uma visão geral sobre os processos de teste e desenvolvimento utilizados pelas comunidades que desenvolvem FLOSS; a Seção 3 mostra os resultados parciais obtidos na avaliação de projetos FLOSS; e, por fim, a Seção 4 traz as considerações finais e trabalhos futuros.

## 2. PROCESSO DE TESTE EM FLOSS

Embora ainda seja pequena a utilização de produtos FLOSS pela indústria, existe uma tendência de crescimento na adoção desse tipo de software uma vez que um número cada vez maior de indústrias opta por tais soluções. Um dos fatores que dificulta a adoção é a incerteza nos processos de garantia de qualidade adotados em tais produtos. Pode-se dizer que os processos de teste empregados pelas comunidades que desenvolvem FLOSS ainda é uma incógnita. Um estudo realizado por Zhao e Elbaum (2003) mostrou que 80% dos desenvolvedores de FLOSS não criavam planos de teste.

Um dos agravantes para isso 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].

Mockus *et al.* (2000) afirmam que o desenvolvimento de FLOSS é radicalmente diferente da indústria de *Closed Source Software* (CSS) pois, no desenvolvimento de FLOSS, encontram-se as seguintes características: o trabalho não é atribuído (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. No entanto, apesar de não seguir um processo formal de desenvolvimento, os resultados obtidos pelas comunidades que desenvolvem FLOSS são considerados equivalentes ou até superiores os das organizações que desenvolvem CSS. Alega-se, por exemplo, que os defeitos são encontrados e resolvidos mais rapidamente, pois há "muitos olhos" voltados para os problemas. 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, além da exposição de suas habilidades de programação [Raymond 2000] *apud* [Mockus *et al*. 2000].

Decorrente dessas características do modelo de desenvolvimento de FLOSS, como 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.

Segundo Zaidman *et al.* (2008) acredita-se que códigos de teste e códigos de produção devam 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; 2) Quando mudanças são aplicadas, a preservação de comportamento do software precisa ser verificada; e 3) Mesmo quando as alterações preservam o comportamento, os testes podem ser invalidados, 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].

Sendo assim, considera-se que o processo de construção de conjuntos de testes empregados pelas comunidades, deve ser analisado na busca por requisitos que possam auxiliar no estabelecimento de confiabilidade a produtos FLOSS e até mesmo na melhoria do processo de geração de casos de teste.

## 3. EXPERIMENTOS REALIZADOS

A grande maioria dos produtos FLOSS investigados possuem um conjunto de teste funcional (caixa-preta) criado pela comunidade de desenvolvimento visando a verificação constante do FLOSS desenvolvido. 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 investigar qual a porcentagem de cobertura de código de produção que é efetivamente executada pelos testes funcionais disponibilizados juntamente com o código fonte de produtos FLOSS.

Tal avaliação foi conduzida com a ferramenta de teste JaBUTi[1], que apóia a aplicação de oito critérios estruturais, sendo que seis deles foram utilizados: $All\text{-}Nodes_{ei}$, $All\text{-}Nodes_{ed}$, $All\text{-}Edges_{ei}$, $All\text{-}Edges_{ed}$, $All\text{-}Uses_{ei}$ e $All\text{-}Uses_{ed}$. Para mais informações sobre a ferramenta, o leitor interessado pode consultar a página do projeto. A diferença entre os critérios *ei* (*exception independent*) e *ed* (*exception dependent*) é decorrente da presença de tratamento de exceção da linguagem Java. Ou seja, os critérios seguidos de *ei* produzem requisitos de teste que podem ser executados sem a necessidade de ocorrência de exceção, ao passo que aqueles seguidos de *ed* geram requisitos que, necessariamente, exigem que uma exceção seja gerada para serem cobertos.

Até o momento, cinco projetos foram analisados (HSQLDB, JMeter, JUnit, Log4J e PMD). O processo de avaliação foi conduzido da seguinte forma: 1) download da última versão disponível do software; 2) configuração do ambiente de teste (variáveis, diretórios, bibliotecas); 3) preparação das aplicações (instrumentação do código) para que as informações de cobertura de código possam ser coletadas durante a execução dos testes; 4) execução dos testes unitários do JUnit com a ferramenta JaBUTi; 5) coleta e análise dos resultados; e 6) Geração de relatórios de cobertura por método, por classe e por projeto.

Com intuito de apresentar o processo de coleta e análise de dados que foi realizado, considere o programa Log4J que é composto por 276 classes, totalizando 8.543 LOC (*Lines of Code*). Parte das classes desse programa e a cobertura do seu conjunto de teste funcional são apresentadas na Tabela 1. Por exemplo, para a classe `SMTPAppender`, o critério $All\text{-}Nodes_{ei}$ gerou 144 requisitos de teste (#REQ) dos quais 49 foram cobertos (#COV); uma cobertura de 34% (%COV). Para a classe `SocketAppender`(terceira classe), considerando o mesmo critério, foram gerados 49 requisitos mas nenhum deles foi executado pelos casos de teste. Esses dados foram coletados para todas as classes de determinado programa e a média de cobertura em relação a cada critério foi calculada. Os resultados são apresentados nas Tabelas 2 e 3.

[1] *Ferramenta de teste JaBUTi. Disponível em: http://ccsl.ime.usp.br/en/project/jabuti/.*

**Tabela 1. Número de requisitos gerados e cobertura dos testes para cada um dos critérios da ferramenta JaBUTi.**

| | All-Nodes | | | | | | All-Edges | | | | | | All-Uses | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ei | | | Fd | | | Ei | | | Fd | | | Ei | | | Fd | | |
| Classes | #Cov | %Cov | #Req | #Cov | %Cov | #Req | #Cov | %Cov | #Req | #Cov | %Cov | #Req | #Cov | %Cov | #Req | #Cov | %Cov | #Req |
| org.apache.log4j.DailyRollingFileAppender | 43 | 95 | 45 | 0 | 0 | 4 | 45 | 81 | 55 | 0 | 0 | 2 | 115 | 83 | 138 | 0 | 0 | 6 |
| org.apache.log4j.net.SMTPAppender | 49 | 34 | 144 | 0 | 0 | 14 | 33 | 23 | 142 | 0 | 0 | 43 | 60 | 20 | 298 | 0 | 0 | 26 |
| org.apache.log4j.net.JMSAppender | 0 | 0 | 68 | 0 | 0 | 8 | 0 | 0 | 57 | 0 | 0 | 63 | 0 | 0 | 119 | 0 | 0 | 20 |
| org.apache.log4j.net.SocketAppender | 30 | 61 | 49 | 4 | 22 | 18 | 24 | 41 | 58 | 1 | 11 | 9 | 38 | 54 | 70 | 9 | 24 | 37 |
| org.apache.log4j.net.JMSSink | 0 | 0 | 27 | 0 | 0 | 6 | 0 | 0 | 30 | 0 | 0 | 8 | 0 | 0 | 41 | 0 | 0 | 6 |
| org.apache.log4j.net.DefaultEvaluator | 2 | 66 | 3 | 0 | 0 | 0 | 1 | 100 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| org.apache.log4j.net.SocketServer | 0 | 0 | 33 | 0 | 0 | 3 | 0 | 0 | 34 | 0 | 0 | 6 | 0 | 0 | 102 | 0 | 0 | 1 |
| org.apache.log4j.net.SimpleSocketServer | 0 | 0 | 22 | 0 | 0 | 3 | 0 | 0 | 22 | 0 | 0 | 4 | 0 | 0 | 23 | 0 | 0 | 1 |

## 3.1. Resultados parciais

A Tabela 2 apresenta as médias de cobertura obtidas para cada um dos cinco produtos avaliados. Cada coluna mostra o número de requisitos cobertos (COV Req) pelo número total de requisitos gerados (Total Req) para os critérios de testes independentes de exceção e a porcentagem de cobertura obtida para cada projeto. Por exemplo, para o Log4J, considerando o critério *All-Nodesei* foram cobertos 2.173 de um total de 5.280 requisitos, ou seja, uma cobertura de 41,08%. Analogamente, a Tabela 3 mostra os resultados de cobertura para os critérios de testes dependentes de exceção. De modo geral, observa-se que os resultados gerais apresentados nas Tabelas 2 e 3 deixam claro que a qualidade dos conjuntos de testes desenvolvidos é questionável do ponto de vista da cobertura obtida. A maior cobertura obtida foi em relação ao critério *All-Nodes-ei* para o Log4J (41,08%). Em muitos casos a cobertura foi inferior a 5% dos requisitos e em outros não chegou a atingir nem 1% (como no caso do critério todos os arcos do subconjunto dependente de exceção do JMeter – Tabela 3). Tal situação demonstra a necessidade de definição de uma estratégia de testes incremental para a melhoria da qualidade desses conjuntos de teste.

**Tabela 2. Cobertura de código para critérios independentes de exceção.**

| Critério/OSS | All-Nodes-ei Cov Reg/Total Reg(%) | All-Edges-ei Cov Reg/Total Reg(%) | All-Uses-ei Cov Reg/Total Reg(%) | All-Pot-Uses-ei Cov Reg/Total Reg(%) |
|---|---|---|---|---|
| HSQLBD | 8.029 / 40.703 (19,73%) | 7.476 / 45.098 (16,58%) | 19.720 / 126.246 (15,62%) | 67.847 / 458.843 (14,79%) |
| JMeter | 7.845 / 20.462 (38,34%) | 5.461 / 19.317 (28,27%) | 10.935 / 41.180 (26,55%) | 33.615 / 130.547 (25,75%) |
| JUnit | 608 / 1.951 (31,16%) | 380 / 1.436 (26,46%) | 631 / 2.624 (24,05%) | 1.475 / 6.243 (23,63%) |
| Log4J | 2.173 / 5.290 (41,08%) | 1.956 / 5.173 (37,81%) | 4.243 / 10.416 (40,74%) | 10.286 / 26.881 (38,26%) |
| PMD | 7.938 / 21.184 (37,47%) | 6.858 / 23.249 (29,50%) | 13.331 / 57.552 (23,16%) | 38.404 / 252.261 (15,22%) |

**Tabela 3. Cobertura de código para critérios Dependentes de Exceção**

| Critério/OSS | All-Nodes-ei Cov Reg/ Total Reg(%) | All-Edges-ei Cov Reg/ Total Reg(%) | All-Uses-ei Cov Reg/ Total Reg(%) | All-Pot-Uses-ei Cov Reg/Total Reg(%) |
|---|---|---|---|---|
| HSQLBD | 141 / 1.942 (7,26%) | 49 / 6.513 (0,75%) | 256 / 2.750 (9,31%) | 3.591 / 38.032 (9,44%) |
| JMeter | 51 / 1.541 (3,31%) | 39 / 4.863 (0,80%) | 52 / 2.093 (2,48%) | 276 / 15.301 (1,80%) |
| JUnit | 12 / 156 (7,69%) | 9 / 184 (4,89%) | 13 / 183 (7,10%) | 29 / 632 (4,59%) |
| Log4J | 30 / 544 (5,51%) | 10 / 973 (1,03%) | 41 / 604 (6,79%) | 186 / 3.773 (4,93%) |
| PMD | 325 / 2.039 (15,94) | 121 / 3.814 (3,17%) | 388 / 3.590 (10,81%) | 1.689 / 20.285 (8,33%) |

Além dos dados de cobertura, a ferramenta JaBUTi também computa, para cada classe, uma série 27 métricas orientadas a objeto, dentre elas, a complexidade ciclomática de McCabe, o acoplamento entre objetos, falta de coesão entre métodos, dentre outras. Um exemplo de parte dessas métricas computadas para o Log4J é mostrado na Tabela 4.

**Tabela 4. Exemplos de métricas Orientadas a Objetos computadas pela JaBUTi**

| Classes | nmos | ncv | cc_max | niv | amm | amz_size | ncm_2 | ncm | nmas | lcom | nii | lcom_3 | amz_lcm | lcom_2 | noc | dit |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| org.apache.log4j.DailyRollingFileAppender | 2 | 8 | 7 | 7 | 0,6 | 43,5 | 0 | 1 | 5 | 0 | 0 | 0 | 9,4 | 0 | 0 | 4 |
| org.apache.log4j.net.SMTPAppender | 4 | 1 | 10 | 18 | 0,53 | 16,08 | 0 | 3 | 45 | 934 | 1 | 937 | 0 | 3 | 0 | 2 |
| org.apache.log4j.net.JMSAppender | 4 | 0 | 7 | 13 | 0,43 | 15,8 | 0 | 0 | 25 | 341 | 0 | 341 | 0 | 0 | 0 | 2 |
| org.apache.log4j.net.SocketAppender | 4 | 3 | 9 | 9 | 0,68 | 17,77 | 0 | 2 | 15 | 74 | 0 | 75 | 5,55 | 1 | 0 | 2 |
| org.apache.log4j.net.JMSSink | 0 | 2 | 4 | 0 | 1,43 | 32 | 1 | 5 | 0 | 1 | 1 | 11 | 0 | 10 | 0 | 1 |
| org.apache.log4j.net.DefaultEvaluator | 0 | 0 | 1 | 0 | 0,5 | 4 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| org.apache.log4j.net.SocketServer | 0 | 6 | 3 | 3 | 1 | 42,38 | 1 | 5 | 0 | 0 | 0 | 10 | 8 | 10 | 0 | 1 |
| org.apache.log4j.net.SimpleSocketServer | 0 | 3 | 2 | 0 | 0,83 | 24,83 | 1 | 5 | 0 | 0 | 0 | 10 | 5 | 10 | 0 | 1 |

Com base nos dados computados, os estudos atuais buscam o estabelecimento de uma estratégia de teste incremental. Por exemplo, dado que se tenha um conjunto de classes que não tiveram boa cobertura, as métricas de software podem ser utilizadas para ordenar as classes de modo a priorizar para quais delas serão desenvolvidos novos casos de teste. Isto é, após a execução dos testes (por exemplo, do JUnit), é possível ordenar todas as classes que tiveram cobertura de requisitos igual ou inferior a determinada cobertura mínima e estabelecer uma ordem entre elas, seja pelo maior tamanho (LOC), pela complexidade ciclomática (CC) ou pelas que possuem maior acoplamento. O objetivo é priorizar os testes daquelas classes que irão contribuir de forma mais significativa para a melhoria da qualidade do produto de software em teste.

## 4. CONSIDERAÇÕES FINAIS

Os dados obtidos até o momento e as análises preliminares demonstram que os conjuntos de teste disponibilizados junto aos produtos FLOSS apresentam coberturas de código inferior a 42% considerando todo o código do produto avaliado. Sendo esse o melhor caso, isso significa que os produtos são liberados para uso com mais de 50% do código não sendo executado por nenhum caso de teste.

A partir desses resultados iniciais, verificou-se a necessidade de uma avaliação mais detalhadas desses projetos FLOSS para que outros elementos importantes possam ser identificados no estabelecimento da confiabilidade a esses produtos como, por exemplo, a real importância de códigos de produção que não são cobertos por nenhum caso de teste, pois eles poderiam ser reavaliados para verificação da sua real necessidade na lógica do programa possibilitando uma otimização do código de produção e até a diminuição de defeitos causada por códigos que “nunca” são executados.

Como foi dito anteriormente, este trabalho está inserido em um contexto maior que é o Projeto QualiPSo e, para dar continuidade à execução do projeto, tem-se os seguintes trabalhos futuros:

- Continuar o processo de análise da cobertura de código em projetos FLOSS desenvolvidos em Java (ao todo serão 27);
- Realizar uma avaliação comparativa entre as diversas versões disponíveis nos projetos FLOSS para verificar se os conjuntos de teste evoluem na mesma proporção que o código do produto;

- Concluir a estratégia de testes incremental visando aumentar a qualidade dos conjuntos de testes empregados pelas comunidades de desenvolvimento de FLOSS.

## REFERÊNCIAS

Elbaum, S., Gable, D., and Rothermel., G. (2001). The impact of software evolution on code coverage information. In ICSM ’01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM’01), pages 170–179, Washington, DC, USA. IEEE Computer Society.

Meirelles, F. S. (2003). Pesquisa Anual - Administração de Recursos de Informática - CIA - Centro de Informática Aplicada. FGV/EAESP.

Mockus, A., Fielding, R. T., and Herbsleb, J. (2000). A case study of open source software development: the apache server. In ICSE ’00: Proceedings of the 22nd international conference on Software engineering, pages 263–272, New York, NY, USA. ACM.

Morasca, S., Taibi, D., and Tosi, D. (2009). 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, pages 25–30.

QualiPSo (2008). Quality Platform for Open Source Software: Trust and Quality in Open Source Systems. Página Web do Projeto QualiPSo. Disponível em: HTTP://www.qualipso.org/. Acesso em 20/01/2010.

Raymond, E. S. (2000). The cathedral and the bazaar. O’Reilly & Associates, Inc., Sebastopol, CA, USA.

Zaidman, A., Rompaey, B. V., Demeyer, S., and Deursen, A. v. (2008). 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, pages 220–229, Washington, DC, USA. IEEE Computer Society.

Zhao, L. and Elbaum, S. (2003). Quality assurance under the open source development model. J. Syst. Softw., 66(1):65–75.