CAROLINA RIBEIRO DA COSTA

UMA ANÁLISE EXPLORATÓRIA DE ALGUNS PLUGINS DE TESTE DE SOFTWARE DISPONÍVEIS PARA AS IDEs ECLIPSE E NETBEANS

LAVRAS - MG
2012

CAROLINA RIBEIRO DA COSTA

UMA ANÁLISE EXPLORATÓRIA DE ALGUNS PLUGINS DE TESTE DE SOFTWARE DISPONÍVEIS PARA AS IDEs ECLIPSE E NETBEANS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Dr. Antônio Maria Pereira de Resende

LAVRAS - MG
2012

CAROLINA RIBEIRO DA COSTA

UMA ANÁLISE EXPLORATÓRIA DE ALGUNS PLUGINS DE TESTE DE SOFTWARE DISPONÍVEIS PARA AS IDEs ECLIPSE E NETBEANS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 07 De Novembro de 2012

DR. ANDRÉ LUIZ ZAMBALDE

MSc. ANDRÉ PIMENTA FREIRE

Dr. ANTÔNIO MARIA PEREIRA DE RESENDE

Orientador

LAVRAS – MG

2012

AGRADECIMENTOS

Primeiramente, à Deus pela força, pela saúde, pelo aprendizado, pelas amizades, por toda a felicidade que me foi proporcionada durante esses anos.

À meus pais, Almir Antônio da Costa e Maria Natividade Ribeiro, por me darem a oportunidade de chegar até aqui, pelo amor e carinho dedicados, e pela força e sabedoria transmitidas.

Ao Leandro pelo apoio, dedicação, esforço e por acreditar na realização desse sonho.

À Karen, pela amizade, pela disposição em ajudar, e pelos conhecimentos compartilhados.

Às colegas de república, Ariana, Isabella e Laís, pelo companheirismo, pelos momentos e alegrias divididas.

Ao professor Antônio Maria, pelas orientações e ajuda oferecida para a conclusão deste trabalho.

A todos os amigos que estiveram presentes nos momentos de conquista e nos momentos difíceis, e a todos que contribuíram de alguma forma para que este dia chegasse.

RESUMO

As ferramentas de teste auxiliam os engenheiros de software na tarefa de aplicação de testes, como por exemplo, por meio dos *plugins*. Neste trabalho, abordam-se técnicas e ferramentas de teste de software, tendo como objetivo final relacionar *plugins* e analisar suas principais características e abordagens. Foram selecionados 4 *plugins,*sendo 2 de teste de unidade e 2 de teste de cobertura, e submetidos a testes simples, a fim de observar o desempenho. Nesta monografia, apresenta-se um guia simplificado sobre algumas ferramentas de teste, correlacionando-as com técnicas de testes, dando enfoque nos testes de unidade, facilitando o trabalho de engenheiros de software na seleção de *plugins* aplicáveis em projetos.

Palavras-chave: *Plugins* de Teste de Software. Engenharia de Software. Seleção de Plugins.

ABSTRACT

Test tools assist software engineers in performing testing, for example, the plugins. This dissertation addresses techniques and tools for software testing, with the ultimate goal of relating plugins and analyzing their main features and approaches. We selected 4 plugins, 2 unit test and 2 test coverage, they were subjected to simple tests in order to observe their performance. This dissertation presents a simplified guide on some test tools, correlating them with testing techniques, focusing on unit tests, facilitating the work of software engineers in the selection of plugins applicable to their projects.

Keywords: Plugins Software Testing. Software Engineering. Selecting Plugins.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A atividade de evolução de software trata do aspecto de que um software possa se adaptar bem às mudanças em seu ambiente e às necessidades impostas pelo cliente. O engenheiro deve estar preparado para fazer essas mudanças a qualquer momento, seja durante ou depois do desenvolvimento do *software*. Inúmeras vezes a necessidade dessas mudanças surgem durante a realização dos testes (SOMMERVILLE, 2003).

Os testes, portanto, são utilizados como uma ferramenta que pode avaliar a qualidade de um software. Antigamente os testes eram realizados, somente ao final do processo de codificação do *software*, e tinham como único intuito a detecção de falhas. Atualmente é visto que a atitude correta é a prevenção de erros, desta maneira os testes são meios para verificar não somente se a prevenção tem sido eficaz, mas também para a identificação de falhas nos casos em que não foi eficaz.

Nem sempre um software pode ser avaliado por meio de testes. De acordo com Bach (1994), um *software* pode ser definido como testável se possuir as seguintes características: operabilidade, observabilidade, controlabilidade, decomponibilidade, simplicidade, estabilidade e compreensibilidade.

Para o engenheiro de software é importante, além da avaliação do software que se deseja testar, acerca da sua testabilidade ou não, um conhecimento sólido acerca do Teste de Software, garantindo a sua correta aplicação e aproveitamento dos seus resultados.

Segundo SWEBOK (2004), podem ser citadas como palavras-chave sobre o conhecimento de Teste de Software: Dinâmico, Finito, Selecionado e Esperado. Dinâmico significa que nem sempre o valor da entrada sozinho é eficiente para determinar um teste. O teste deve ser finito, pois mesmo nos

programas mais simples muitos casos de testes podem ser exaustivos e levar meses ou anos para serem executados. Identificar o critério de seleção do conjunto de teste pode ser uma tarefa muito complexa, por isso são aplicadas técnicas de análise de risco e conhecimentos de engenharia de teste; é importante manter testadores cientes dos diferentes resultados dependendo do critério de seleção. Devem ser observados os resultados da execução do programa, e decidido se são esperados, avaliando a expectativa do usuário e a especificação.

O principal objetivo do Teste de software é avaliar a qualidade de software, e melhorá-la, por meio da identificação de defeitos e problemas. Dentre as técnicas existentes, pode-se citar a análise do comportamento dinâmico de um programa em um conjunto finito de casos de teste, devidamente selecionados a partir do domínio de execuções, geralmente infinito, contra o comportamento esperado.

Para acelerar a atividade de testes, existem ferramentas que aplicam de maneira automatizada testes de software. Estas ferramentas proporcionam benefícios como uma aceleração do processo de teste, redução da incidência de erro durante esse processo, e como consequência uma maior confiabilidade dos testes. Selecionar quais delas aplicar tornou-se uma tarefa difícil considerando as inúmeras ferramentas existentes.

Falta uma análise comparativa de plugins, que possa servir como instrumento de consulta, que forneça informações sobre a aplicação destes, sobre o seu modo de funcionamento, características positivas e negativas acerca de plugins que realizam o mesmo tipo de teste, facilitando o trabalho do engenheiro de software quando na escolha entre um ou outro.

1.1 Objetivo Geral

Realizar uma análise exploratória de alguns *plugins* de testes do Eclipse e Netbeans, a fim de facilitar a seleção destes pelos Engenheiros de Software.

1.2 Objetivo Específico

O trabalho seguiu as seguintes etapas:

- Fazer a Revisão bibliográfica, abordando as técnicas de testes e plugins, contextualizando neste trabalho;
- Identificar trabalhos relacionados, para identificar as principais lacunas existentes atualmente na área de análise de plugins de teste;
- Relacionar os *plugins* de testes do Eclipse e Netbeans;
- Selecionar os *plugins* para realizar a análise comparativa;
- Descrever e aplicar cada *plugin*;
- Construir uma tabela exploratória das principais características dos *plugins*.

1.3 Estrutura do Documento

O trabalho apresenta-se na seguinte estrutura:

- O Capítulo 2 expõe a revisão bibliográfica, onde é conceituado teste de software, e demais termos para o trabalho;
- O Capítulo 3 detalha alguns trabalhos relacionados, mostrando suas estruturas e conteúdos;

- O Capítulo 4 exibe a metodologia utilizada nesse trabalho, descrevendo os passos substanciais para se alcançar o objetivo inicialmente esperado;
- O Capítulo 5 descreve cada *plugin* selecionado na prática, ilustrando o seu funcionamento na prática;
- O Capítulo 6 expõe os resultados, exibindo uma avaliação geral e um quadro-resumo que visa facilitar a escolha de *plugins*;
- O Capítulo 7 é a conclusão deste trabalho;
- Por fim o Capítulo 8 lista as referências bibliográficas essenciais para desenvolver-se o trabalho.

2 REFERENCIAL TEÓRICO

2.1 Teste de *software*

De acordo com Presman (2000), mesmo com métodos, técnicas e ferramentas, usados durante o desenvolvimento, alguns erros ainda permanecem. Para que sejam minimizados, necessita-se de atividades que possam garantir a qualidade. Podem ser citadas ,como exemplos destas atividades: a verificação, a validação e o teste.

Segundo Bach (1994), testabilidade de um software representa o quão fácil um programa de computador pode ser testado. Para avaliar se um software é testável, as principais características a se considerar são operabilidade, observabilidade, controlabilidade, decomponibilidade, simplicidade, estabilidade e compreensibilidade.

O engenheiro de *software* deve trabalhar sempre com o conceito de "testabilidade" na hora de projetar e implementar um sistema. Os testes devem conter um conjunto de características com o objetivo de encontrar o maior número de erros com um mínimo de esforço (PRESSMAN, 2005).

O total cumprimento da qualidade e confiabilidade dos requisitos de sistemas de *software* complexos exige procedimentos específicos de verificação e validação. (SOMMERVILLE, 2003).

Para Sommerville (2003), validação e verificação são os nomes dados aos processos de confirmação da análise, para garantir que o *software* atenda as suas especificações. Na validação e verificação as principais atividades a serem contempladas são: revisão dos requisitos, revisões do projeto, inspeções do código e testes do sistema.

Validação e verificação são comumente confundidas. Sucintamente pode-se expressar suas diferenças com as questões de Boehm (1981) apud Sommerville (2003):

Validação: Avalia se o sistema atende às expectativas do cliente. Os testes unitários, de integração, de sistema e de aceitação podem ser classificados como testes de validação. A validação intenciona responder se o produto certo está sendo construído.

Verificação: Realizar inspeções/revisões para conferir se o *software* cumpre com suas especificações, garantido que o sistema atenda os requisitos funcionais e não funcionais especificados. Alguns autores costumam chamar esta atividade de 'Teste de Verificação'. A verificação objetiva responder se a construção do produto é feita da maneira correta.

De acordo com Pressman (2000), uma taxa recomendada do esforço de testes é de 30 a 40% do esforço total do projeto, como ilustrado na Figura 1. Estimar o esforço de testes é uma tarefa árdua e imprecisa em virtude de que existem muitos fatores externos e internos que podem afetar e influenciar o resultado da estimativa. Frequentemente, esta estimativa é realizada de maneira informal. No entanto, é possível estimar com um maior grau de precisão por meio da utilização de abordagens formais de estimativas. Portanto, não existe uma solução que sirva para todos os casos, cada abordagem deve ser avaliada criteriosamente antes de ser utilizada.

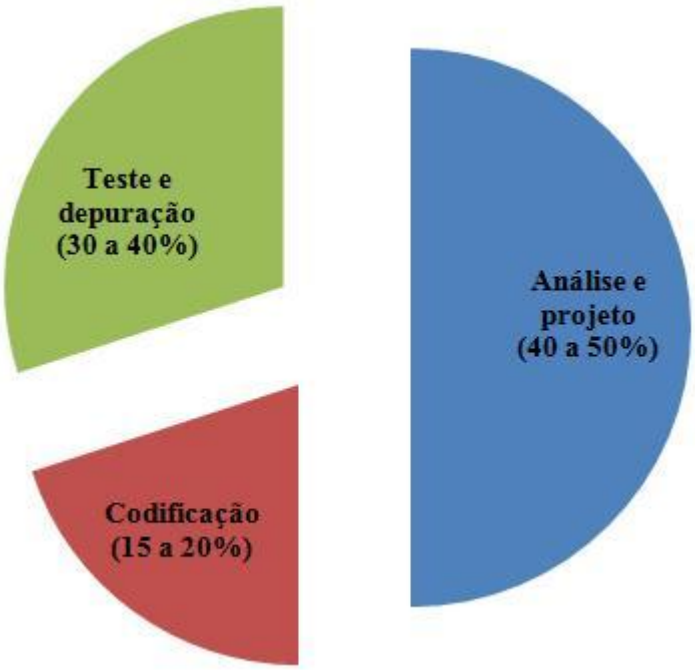


Figura 1 Gráfico de proporção de esforço em um projeto
Fonte: Adaptado de Pressman (2000)

Em suma, nos fundamentos de teste de software, são levantadas algumas questões básicas, como os critérios de seleção de teste, que podem ser usados para selecionar os casos de teste ou para verificar se um pacote de teste selecionado é adequado; o teste de eficácia, que é a observação de uma amostra de execução do programa; teste da identificação de defeitos, onde o teste bem sucedido será aquele que faça com que o sistema falhe; testabilidade, o termo tem dois significados relacionados, um refere-se ao grau em que é fácil para o *software* atender um determinado critério de cobertura de teste, o outro é definido como a probabilidade de que o *software* irá expor uma falha no teste (SWEBOK, 2004).

2.2 Níveis de teste e algumas técnicas relevantes para este trabalho

O planejamento dos testes deve ocorrer em diferentes níveis e em paralelo ao desenvolvimento do software. Segundo Rocha (2001), os principais níveis de teste de *software* são:

- **Teste de Unidade:** também conhecido como testes unitários. Tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código. É um teste feito pelo desenvolvedor. As classes de teste devem ser projetadas e desenvolvidas antes do próprio projeto. É sugerido também que os testes unitários sejam realizados diariamente para garantir a correção de pequenos erros e evitar problemas maiores ao final.

- **Teste de Integração:** visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do *software* que foi estabelecida na fase de projeto.

- **Teste de Sistema:** avalia o *software* em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do software. Verifica se o produto satisfaz seus requisitos.

- **Teste de Aceitação:** são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.

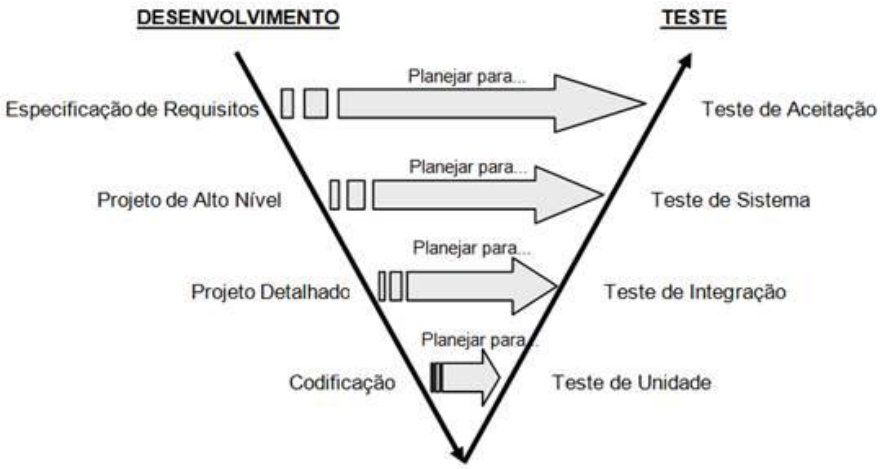


Figura 2 Descrição do paralelismo entre as atividades de desenvolvimento e teste de *software.*
Fonte: Devmedia (2012).

Como pode ser visualizado na Figura 2, o planejamento e projeto dos testes devem ocorrer de cima para baixo, ou seja, em primeiro lugar o planejamento do teste de aceitação, a partir dos requisitos; em segundo o planejamento do teste de sistema, a partir do projeto de alto nível; em seguida o planejamento dos testes de integração, a partir do projeto detalhado e por último o planejamento dos testes de codificação.

As técnicas de teste que foram relevantes para abordagem neste trabalho foram:

- **Teste de Regressão:** Teste de regressão não corresponde a um nível de teste, mas é uma estratégia importante para redução de "efeitos colaterais". Consiste em se aplicar, a cada nova versão do *software* ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema. Pode ser aplicado em qualquer nível de teste.

- **Teste de Cobertura:** também ligado à qualidade do software, avalia a quantidade do código que foi executada em um determinado teste. Se a cobertura não é 100%, significa que mais testes podem ser escritos para os itens que foram perdidos e dessa maneira melhorar a cobertura.

3 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados trabalhos com conteúdo relacionado, ou seja, que abordem testes de *software* e ferramentas que realizam esses testes.

3.1 Um Estudo Sobre os Aspectos Teóricos e Práticos de Teste de Software (Bandera, Oliveira e Silva, 2008)

O objetivo do trabalho foi criar um processo de teste automatizado com o auxílio de ferramentas da IDE Eclipse.

No primeiro estudo de caso foi utilizada primeiramente a ferramenta Metrics, que aborda as classes a serem testadas, analisando e verificando suas medidas, tendo como principais métricas: complexidade ciclomática; quantidade de parâmetros; número de subclasses; números de linhas de código total; e outros.

Em seguida, construção de casos de teste com base na análise realizada, levando em conta a complexidade ciclomática, com o objetivo de varrer os métodos da classe que será testada.

A partir do plano de testes iniciou-se a implementação dos casos de testes utilizando o JUnit.

Com a ferramenta EclEmma foi possível obter uma análise da cobertura dos testes por linha.

Gerou-se uma documentação com os resultados dos testes, com os erros e acertos encontrados utilizando a ferramenta Ant, que realiza esse processo de forma automatizada.

O objetivo final foi mostrar a praticidade da realização de testes unitários com a integração das ferramentas no desenvolvimento de *softwares*.

No segundo estudo de caso, foi utilizado um sistema com a finalidade de gerenciar e controlar cadastros de imóveis e atividades econômicas que servem para o lançamento de tributos. Onde o caso de uso analisado foi o "Manter Distrito", que limita as cidades para fins cartográficos.

Primeiramente foi construído o plano de teste. Com o auxílio da ferramenta Metrics verificou-se os métodos da classe.

Construiu-se casos de teste para cada método. O JUnit executa os casos de teste. O EclEmma analisa os resultados retornando a cobertura do teste. E por fim é gerado um relatório com a ferramenta Ant.

Neste trabalho, foram utilizados dois exemplos de software onde foram aplicados os mesmos *plugins* em cada um, foi uma abordagem interessante, em que foi possível avaliar o comportamento do *plugin* diante de situações distintas. Porém, seria uma abordagem significante a aplicação de *plugins* que têm a proposta de fazer o mesmo tipo de teste e trazer o mesmo resultado, sendo testados no mesmo software para que fosse possível fazer uma comparação entre os resultados de cada um diante da mesma situação.

3.2 *Research on the Definition and Model of Software Testing Quality* (Jin, Zeng, 2011)

Ele descreve a definição de qualidade de teste de *software*, constrói um quadro modelo sobre qualidade de teste de *software* e sobre métricas de qualidade. A qualidade está ligada ao grau de satisfação com o produto, e para medir a qualidade é essencial escolher um indicador dentre as regras centrais, esta escolha é crucial.

Apresenta como conceito de qualidade de teste de *software* "o nível de melhoria do benefício de melhora do serviço de teste que é executado pela organização de teste.". Apontando como pontos principais que o teste de

software é fornecido por organizações individuais ou de terceiros e que o grau de qualidade do teste reflete o benefício do teste.

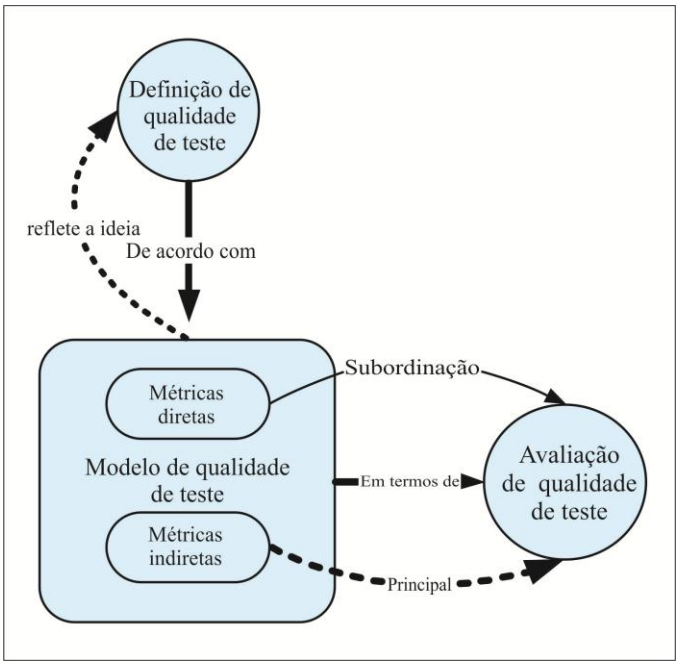


Figura 3Relação entre definição do modelo de qualidade de software, modelo e avaliação

Fonte: Adaptado de Jin e Zeng (2011)

A Figura 3 reflete diretamente o uso do modelo. Os indicadores de métricas diretos e indiretos formam o modelo. Os indiretos são formados a partir do processo de teste e seus resultados e os diretos são suplementos.

A pesquisa traz como retorno a impossibilidade do objeto de avaliação da qualidade ser descrito precisamente, mas ele torna a construção do modelo mais difícil. No método geral, primeiro ocorre a construção do modelo de estrutura de qualidade do teste de *software*, e a partir dele o modelo de detalhes.

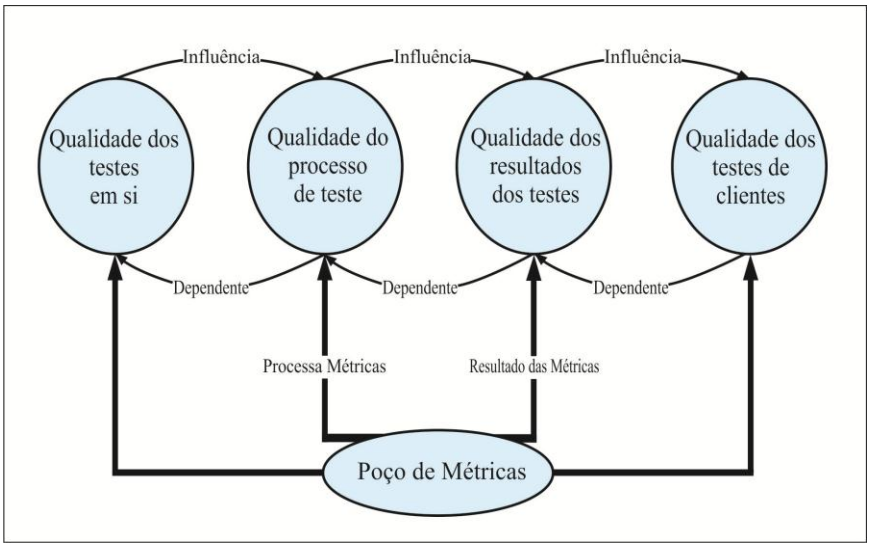


Figura 4 Modelo da estrutura de qualidade do teste de *software*
Fonte: Adapatado de Jin e Zeng (2011)

Indicadores de métricas de cada um dos atributos de qualidade:

- **Qualidade do teste em si:** O teste de software pode ter defeitos, e estes têm uma influência negativa sobre a capacidade dos testes de detectarem "*bugs*".
- **Qualidade do processo de teste:** Processos típicos como estabilidade do ambiente de teste, eficiência do caso de teste, habilidade dos engenheiros de teste.
- **Qualidade dos resultados do teste:** Métricas geradas a partir do relatório de teste, como a relação de defeitos e o seu custo.
- **Qualidade dos testes de clientes:** São os clientes que sentem diretamente os efeitos dos testes, dessa maneira sua avaliação deve ser levada em consideração.

A transferência de um modelo para outro objetiva descobrir os indicadores adequados de métricas. Pode se dar de duas maneiras: a partir de

padrões autorizados e pelo método GQM, para desenvolver a partir dos quatro atributos do modelo de estrutura de qualidade. Essas duas maneiras são combinadas na escolha do indicador de métrica.

O modelo de métrica é decomposto do alto de um nível para outro e inversamente o processo de integração do nível baixo para o alto nível.

No processo de avaliação, são coletados todos os dados do ciclo de vida do teste, em seguida, calcula-se o valor dos indicadores que é levado em consideração. Os valores não podem ser identificados em número, e é usada a matemática *fuzzy* para calcular o processo de avaliação.

Os autores acreditam que as métricas do modelo de atributos será a prioridade no futuro. E como a maior dificuldade relataram a aplicação de métricas precisas para os engenheiros de teste.

Constatou-se como uma das lacunas do trabalho, a falta de um exemplo prático onde fosse avaliada a qualidade de um teste por meio das métricas do modelo apresentado.

Os indicadores de métricas foram apresentados por meio de um modelo e cada um definido separadamente, mas tornaria mais fácil o processo de entendimento e mais completa a apresentação destes se fossem demonstrados na prática diante de um exemplo.

3.3 Automação de Testes Utilizando Ferramentas Open Source (Lousa, Nunes, 2006)

O trabalho teve como foco principal automatizar uma parte do processo de teste de um *software*, aumentando assim sua qualidade, com o uso de ferramentas *open source*.

Demonstram-se tipos de testes e suas definições em seguida. Como se dá o processo de desenvolvimento *open source* comparado com o espiral e

cascata. O espiral é um trabalho em conjunto com os testes de design, de desenvolvimento, de planejamento e de execução. Foram apontadas algumas limitações do desenvolvimento open source, como algum conhecimento específico, ou software que não pode ser amplamente divulgado, e outras. Para o projeto *open source* identificou-se uma deficiência na documentação, porém o tempo dedicado à fase de testes mostrou-se bastante significativa. É dada uma ênfase grande no teste de campo, desenvolvedores argumentam que grande parte dos erros, principalmente os difíceis de detectar, são encontrados pelos usuários. Com a grande contribuição de desenvolvedores e usuários exige-se uma grande atenção na integridade do *software*.

Adiante os autores analisam modelos de desenvolvimento *open source* de *softwares* amplamente conhecidos como o *OpenOffice* por exemplo, focando no processo de teste.

Em outro momento, os autores falam sobre a automatização dos testes. Segundo Fewster e Graham (1999 citado por LOUSA; NUNES, 2006), a automatização de um teste é mais cara que sua execução uma única vez manualmente, porém esse valor é compensado quando o teste é executado várias vezes. Portanto é necessária uma avaliação quanto ao custo-benefício de sua aplicação.

Na fase final, é feita a implementação dos testes no *software* proposto, assim como uma análise de cada etapa dos testes, e conclui-se o desempenho de cada *plugin* de teste utilizado na IDE Eclipse.

Por último eles concluem que a qualidade de *software* tem se tornado cada vez mais relevante, e que os testes de *software* fazem parte da Garantia da Qualidade de *Software*, e que para escolha das ferramentas *open source* é importante definir critérios, alocar um tempo para pesquisa, como também para instalação e configuração.

O trabalho contém uma explicação detalhada da metodologia seguida na hora da execução dos testes, bem como os erros encontrados e como foram feitas as correções. Tornaria a apresentação do processo mais completa se fosse esclarecido as etapas seguidas para utilização de cada *plugin*, demonstrando como se dá o uso destes, para uma maior facilidade em futuras aplicações.

3.4 *Information Theory, Information View, and Software Testing* (Zuo, 2010)

O trabalho estudou os testes de *software* a partir de um novo ponto de vista chamado "visão de informação". Este aponta os testes de *software* como produtores de informações para reduzir incertezas. Mostra que equações usadas na teoria da informação podem calcular a quantidade de informações que serão geradas por cada caso de teste, e têm a possibilidade de serem usadas para mensurar o processo de teste.

Uma tentativa de usar a teoria da informação para o estudo do teste de software.

A Figura 5 mostra a transformação que ocorre com os testes de software. A transformação de incertezas em informações, e partes que não se alteram com o teste.

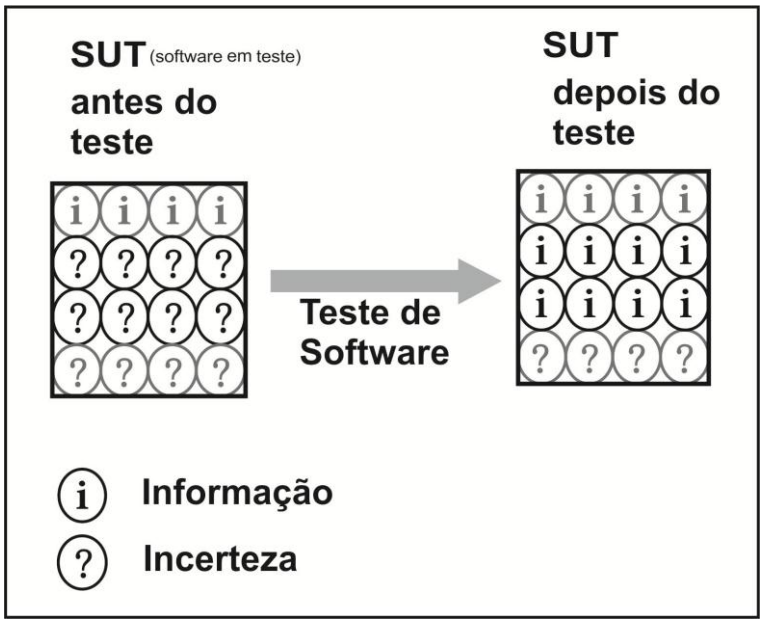


Figura 5 Ilustração da transformação ocorrida no software
Fonte: Adaptado de Zuo (2010)

O autor compara o produto do teste na visão de informação com outras atividades genéricas de outros produtos, como exemplo cita a perfuração de petróleo, esta terá como produto o óleo físico e aquela, as informações.

O autor diz que em tecnologia da informação o conceito de entropia é a quantidade de incerteza e a quantidade de informação é dada pela mudança da entropia. Dessa maneira, a fórmula do montante de informação é dada pela diferença da entropia antes e depois dos testes.

Para finalizar o autor aponta algumas questões para serem discutidas em trabalhos futuros, como por exemplo "Como calcular o valor de mercado para informações geradas a partir do teste de *software*?", dado que o resultado de outros produtos do mercado têm seus valores, para isso é preciso ver a informação também como um produto.

A visão de que o teste de software transforma, com sua aplicação, incertezas contidas no software em informações é muito válida para utilização na melhoria da qualidade do software. Com a utilização de todos os pontos favoráveis trazidos pelo teste de software, a probabilidade de alcançar um software de qualidade é alta. Esta visão apresentada, portanto, pode ser útil aos engenheiros de software, para aplicação em seus trabalhos.

4 DEFINIÇÃO E APLICAÇÃO DA METODOLOGIA PARA ANÁLISE EXPLORATÓRIA

Este Capítulo descreve a metodologia usada no trabalho para a análise, colocando todos os passos seguidos. Os capítulos seguintes contem os resultados da aplicação de cada etapa.

4.1 Metodologia

O trabalho foi realizado seguindo-se alguns passos sistematizados para obtenção do resultado final.

No primeiro passo, relacionar *plugins* do Netbeans e do Eclipse. Para essa etapa a sistemática escolhida foi a visita ao site oficial de ambas as IDES. Foi feita uma pesquisa na página específica dos *plugins*, dentro da classificação dos *plugins* de teste. Com o resultado obtido foi feita a relação, apresentando as características principais de cada um.

No segundo passo, selecionar *plugins* que seriam usados no teste prático, o critério usado foi a gratuidade dos mesmos e a referência a testes de unidade. Em seguida é apresentada a descrição de cada *plugin* selecionado a partir de pesquisa feita detalhando suas funções e características.

O Capítulo 5 expõe o primeiro e o segundo passo, até a descrição dos *plugins* selecionados

No passo seguinte, cada *plugin* selecionado foi visto na prática. Com um software escolhido e no seu ambiente de atuação, o desempenho dele será acompanhado, destacando-se os resultados que cada um apresentou. Passo apresentado no Capítulo 6.

O Capítulo 7 traz uma tabela com os pontos de maior destaque de cada um. Funcionará como um guia de consulta para aqueles que precisarem saber mais sobre os *plugins* antes de utilizá-los.

Este conjunto de passos está ilustrado na Figura 6.

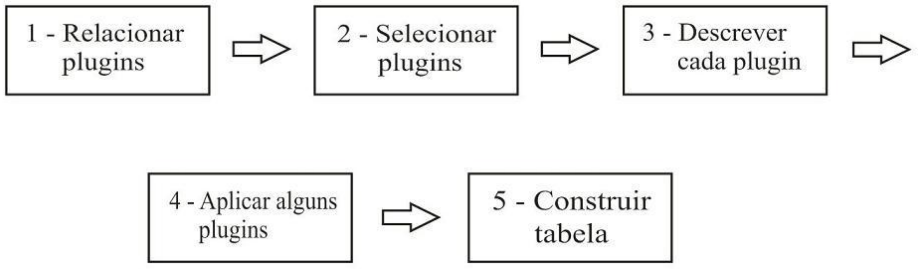


Figura 6 Esquema dos passos seguidos neste trabalho

5 DESENVOLVIMENTO

5.1.1 Resultado no site oficial do Eclipse

A pesquisa foi feita no site oficial do Eclipse, na parte de "*Testing*", "*AllMarket*" e restrição da busca com a palavra "java".

Segue a lista de *plugins* encontrada na pesquisa do Eclipse:

- BlackBerry Java Plugin for Eclipse
- eFitNesse - Testing framework for OSGI and embedded Java applicantions
- Squish
- EclEmma
- Cantata ++
- Jnario
- mBProfiler
- CodingSpectator
- CloudBees
- IBM WebSphere Application Server v8.5 Developer Tools Beta1
- Testdroid Recorder
- Xcarecrows ITM 4
- Xcarecrows IRM 4
- Xored Q7
- Jubula
- EMF
- Chronon

- JSnapshot
- Maveryx
- Sureassert
- EDepChk
- Rap Corporate Portal
- MailSnag
- SpryTest
- Oracle Enterprise Pack
- Atlassian Clover
- eCobertura
- GUIdancer
- MibTiger
- JUnit Flux
- JES email Server launcher
- FlexUnit
- Treaty
- GrinderStone
- AppPerfect Unit Tester
- AppPerfect Testing/Analysis
- WindowsTester Pro
- CodePro AnalytiX
- JUnit Factory
- JDepend4Eclipse
- HTTP4E
- GlobalTester
- MockCentral

- JDemo
- Cute C++

A tabela completa, que compreende a lista de todos os plugins, a descrição de cada um e as suas licenças, pode ser visualizada no APÊNDICE A.

5.1.2 Resultado no site oficial do NetBeans

Pesquisa foi feita no site oficial do NetBeans na categoria "*testing*" e com a restrição de *plugins* para a versão 7.1 da IDE. Segue o resultado:

- Unit Tests Code Coverage Plugin
- Selenium Module for Maven
- Selenium Module for PHP
- Selenium Module for Ant Projects
- Selenium Server

A tabela com a lista completa de todos os plugins do NetBeans, suas descrições e sob qual licença funcionam se encontra no APÊNDICE B.

5.2 Plugins selecionados

Os critérios adotados para realizar a seleção dos *plugins* foram a gratuidade do *plugin* e se estavam relacionados a teste de unidade e de cobertura. Foram escolhidos dois *plugins* que efetuam o teste de unidade em si, e outros dois que fazem a cobertura desses testes de unidade.

5.2.1 Sureassert

Lincença: Gratuito e *open-source* **IDEs:** Eclipse **Última atualização:** 23/10/2011 **Técnica de teste:** Teste de unidade ou caixa-branca

Sureassert é um conjunto de "testes de unidade Java integrados para Eclipse" como definição no site oficial.

A ideia do *plugin* é substituir a grande quantidade de códigos de teste de unidade por anotações que definem um conjunto de testes para cada um dos métodos da classe. Ele irá checar as anotações, gerar os testes apropriados e executá-los. Assim os testes se tornam uma parte mais integral do seu desenvolvimento, como resultado é possível ver os erros de teste do mesmo modo que os erros de compilação.

Objetiva maximizar os benefícios do desenvolvimento *Contract-First* (identificar em primeiro lugar as interfaces que vamos usar, antes de iniciar a escrita do código. Depois desta fase de identificação das operações (métodos) e respectivos parâmetros, estamos em condições de proceder à geração automática do código) e o TDD.

A anotação "*Exemplar*" do Sureassert pode fazer parte da especificação do método. Ele permite testes em um método testes de unidade declarativa. Um exemplo:

```
@Exemplar(args={"1", "2"}, expect="3")
public int add(int x, int y) {

      return x + y;
}
```

Cada atributo "*Exemplar*" recebe uma string, que é tratada pelo Sureassert como uma expressão SIN. As expressões SIN são tipos prefixados que simplificam a criação de objetos. As expressões podem obter valores, que referem-se aos objetos em teste ou parâmetros do método em teste, e fazer qualquer chamada do construtor ou método. Desta maneira é possível construir os testes com o "*Exemplar*".

Sureassert também inclui relatórios de cobertura de testes integrado. Este também é anexado ao processo de compilação do Eclipse.

5.2.2 EclEmma

Lincença: Eclipse *Public License*
IDEs: Eclipse
Última atualização: 09/05/2012
Técnica de teste: Cobertura de teste de unidade

O EclEmma é uma ferramenta de verificação de cobertura de testes unitários. Com esta ferramenta é possível verificar a porcentagem de código que foi efetivamente testada pelos testes unitários desenvolvidos para a aplicação.

EclEmma é uma ferramenta Java livre de cobertura de código para o Eclipse, disponível sob a licença Eclipse *Public License*. Internamente era baseada na Java EMMA grande ferramenta de cobertura de código, tentando adotar a filosofia Emma para a bancada do Eclipse:

- **ciclo rápido de desenvolvimento/teste**: Como execuções de teste JUnit pode ser analisado diretamente para cobertura de código;
- **análise de cobertura Rich**: Cobertura e resultados são imediatamente resumidas e destacadas nos editores de código Java fonte;
- **não-invasiva**: EclEmma não requer modificação ou realização de qualquer outra configuração.

A partir da versão 2.0 é baseada na biblioteca JaCoCo. Um conjunto diferente de contadores é utilizado por esta para calcular métricas de cobertura. Esses contadores provêm de informações existentes nos arquivos de classes Java, que basicamente são as instruções *bytecodes* e as informações de depuração.

A integração do Eclipse tem seu foco no apoio ao desenvolvedor individual de uma forma bastante interativa.

5.2.3 JUnit

Lincença: *Common Public License*
IDEs: Eclipse e NetBeans
Última atualização: 16/04/2012
Técnica de teste: Teste de unidade ou caixa-branca

A ferramenta JUnit fornece uma API ou conjunto de classes completo para a construção de aplicações e para a execução dos testes criados. Seu funcionamento baseia-se na verificação de cada unidade do código, avaliando se o seu desempenho dá-se da forma esperada. Apresenta um modo facilitado de criação e execução automática dos testes e a apresentação dos seus resultados. É um *framework* gratuito e pode ser obtido no site oficial da ferramenta. Já vem configurado nas versões recentes do Eclipse e NetBeans.

A Figura 7 demonstra como as classes do JUnit estão organizadas dentro da API do *framewok*. A classe Test possui um método runTest que executa testes particulares. A classe TestCase que herda da classe Test, testa o resultado dos métodos. E a classe TestSuit possibilita a execução de todos os TestCases de uma só vez. O método setUp() pertencente a classe TestCase indica o início do processo de teste, por esta razão deve ser localizado antes do método de teste, é responsável por inicializar as variáveis e criar os objetos. Já o método tearDown() indica o final do processo, ou seja, contrário ao método setUp(), apresentando-se ao final do método, responsável por finalizar corretamente o método. Métodos setUP() e tearDown() são importantes pois de maneira geral os dados são reutilizados por vários testes. O método assertEquals faz a comparação entre o resultado do método testado e o resultado passado pelo desenvolvedor.

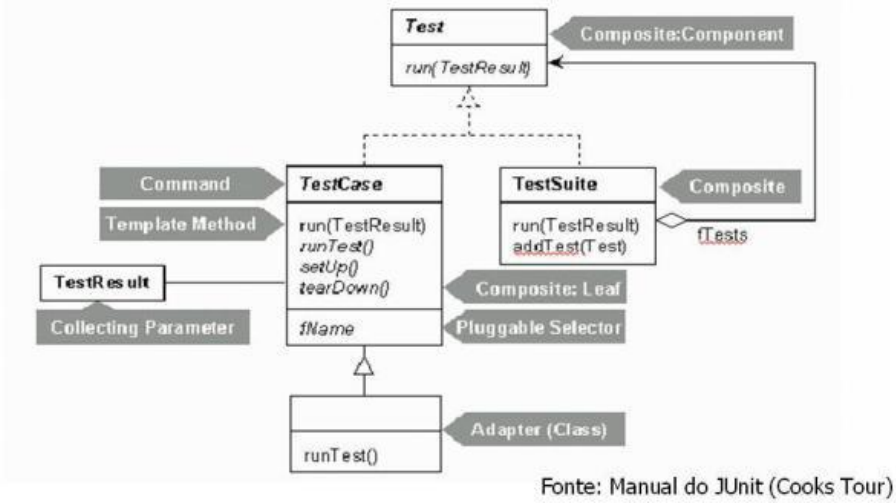


Figura 7 Organização das classes do JUnit

Fonte: Adaptado de Manual do JUnit (Cooks Tour 1999)

5.2.4 Unit Tests Code Coverage

Lincença: *Common Development and Distribution License*
IDEs: NetBeans
Última atualização: 11/05/2010
Técnica de teste: Cobertura de teste de unidade

Plugin de cobertura de testes de unidade para a IDE NetBeans, internamente dependente da biblioteca de cobertura de código Emma. Tem seu funcionamento baseado em classes instrumentadas, ou seja, uma alteração nas classes originais para que seja possível a verificação de quais partes do código está executando realmente. Os *bytecodes* inseridos nestas classes armazenam diversos bits de dados para uma posterior análise.

Como relatado na página no *plugin*, sua funcionalidade proporciona facilidade e rapidez na identificação das partes do código com baixa cobertura, auxiliando o desenvolvimento dos testes. Também disponibiliza recursos como código colorido conforme cobertura da última execução de testes de unidade, atualização automática da marcação do código após execução de testes de unidade ou reabertura dos arquivos.

Pode ser instalado a partir do próprio NetBeans, através do menu "Ferramentas", opção "Plugins". Após instalado, precisa ser ativado para que seja possível a sua utilização, isso é feito por meio do menu do projeto, na opção "*Coverage*". Uma opção disponível também é a "*Show Project Coverage Statistics*" que demonstra a porcentagem coberta e o número de linhas.

6 ESTUDO DE CASO E ANÁLISE

Este capítulo ilustrará a aplicação dos *plugins* selecionados na prática, para tanto será usado um projeto de uma calculadora.

6.1 Apresentação do projeto que será testado

O projeto escolhido para a realização dos testes com os *plugins* selecionados foi encontrado nos projetos disponíveis no site Java, é um projeto simples de uma calculadora, foi finalizado em 2009. Este trabalho focará em uma classe específica do projeto, chamada "PCMath.java" onde é realizada a quase totalidade das operações da calculadora.

Abaixo será transcrito o diagrama de classe da "PCMath.java", para auxiliar no entendimento dos testes. O diagrama descreve a classe, que não contém atributos e com métodos implementando operações matemáticas como *add* (operação de adição), *subtract* (operação de subtração), *multiply* (operação de multiplicação), dentre outras. O código-fonte completo referente à esta classe está disponível no APÊNDICE C.

PCMath
+ booleanToDouble(): void +factorial(): void +add(): void +subtract(): void +multiply(): void +divide(): void

+logicalAnd(): void +logicalOr(): void +logicalXor(): void

Após uma análise na classe, foram necessárias algumas modificações visto que os métodos que realizam ações com os operadores lógicos (logicalAnd, logicalOr, logicarXor) não estavam em pleno funcionamento devido ao uso das funções de comparação "== " e "!=", pois tais funções não são adequadas para uso com o tipo "double". O caminho encontrado para resolver o problema foi criar uma variável chamada tolerância que era ser comparada com a diferença dos parâmetros. Onde a diferença se apresentasse menor que a tolerância, os valores seriam tomados como iguais, caso contrário, diferentes. Abaixo os métodos com as modificações aplicadas, de maneira que funcionasse corretamente (Código-fonte 1, Código-fonte 2, Código-fonte 3):

```
1. public static Double logicalAnd(Double left,
Double right) {
2.    Double result = 0.0;
3.    Double toleranceAnd = 0.001;
4.
5.    if (Math.abs(left - right)< toleranceAnd){
6.          result = 1.0;
7.    }else{
8.          result = 0.0;
9.    }
10.   return result;
11.}
```

Código-fonte 1 Método logicalAnd modificado para funcionar corretamente

```
1. public static Double logicalOr(Double left, Double
right) {
2. Double result = 0.0;
3. Double toleranceOr = 0.001;
4.
5.    if (Math.abs(left - right)> toleranceOr){
6.          result = 1.0;
7.    }else{
8.          result = 0.0;
9.    }
10.
11.   return result;
12. }
```

Código-fonte 2 Método logicalOr modificado para funcionar corretamente

```
1. public static Double logicalXOr(Double left,
Double right) {
2. Double result = 0.0;
3. Double toleranceXOr = 0.001;
4.
5.    if (!(Math.abs(left - right)> toleranceXOr)){
6.          result = 1.0;
7.    }else{
8.          result = 0.0;
9.    }
10.
11.   return result;
12.}
```

Código-fonte 3 Método logicalXor modificado para funcionar corretamente

6.2 Aplicação do plugin JUnit no Eclipse

O *plugin* JUnit não se encontra disponível como opção no site do Eclipse nem do NetBeans, porém nas versões mais recentes dessas duas IDE's ele já vem configurado. É uma ferramenta bastante utilizada e possui uma extensa documentação. É uma ferramenta *open-source*. A versão da IDE Eclipse utilizada foi o Eclipse SDK 4.0.

Para a realização dos testes com JUnit, torna-se indispensável a construção de uma classe onde se encontrarão os testes JUnit. A classe de teste deve herdar a classe junit.framework.TestCase, e conter os métodos de teste, geralmente com o mesmo nome do método que será testado. No exemplo que será apresentado o nome dos métodos de teste virá semelhante ao método que ele testa, antecedido de “test”; por exemplo o método “add”, que realiza a soma, terá como método de teste do JUnit o de nome “testAdd”. A classe a ser testada será a “PCMath” do projeto “PersonalCalculator”. Como observa-se nas linhas de 31 a 40 do código-fonte (método testAdd, que faz o teste no método Add), criaram-se variáveis para se passar os parâmetros do método “ add”, o valor esperado e o resultado. A linha 37 faz a chamada do método add e armazena o resultado e na linha 39 foi feita a comparação entre o valor esperado e o resultado por meio do método “assertEquals” do JUnit.

Essa mesma lógica é utilizada para testar os demais métodos, isto é, criam-se as variáveis necessárias para os parâmetros mais as variáveis para o valor correto esperado e o retorno dado pelo método. Chama-se o método correspondente ao método de teste para armazenar o resultado e ,por fim, faz-se a comparação entre o valor correto esperado e o resultado. Abaixo se encontra a classe de teste JUnit “PCMathTest” (Código-fonte 4):

```
1. package org.lane.personalcalculator.test;
2. import junit.framework.TestCase;
3.
4. import org.lane.personalcalculator.math.PCMath;
5.
6. public class PCMathTest extends TestCase{
7.    public void testBooleanToDouble(){
8.          //Valores que serão testados
9.          boolean valor1 = false;
10.         boolean valor2 = true;
11.         double esperado1 = 0.0;
12.         double e1 = 1.0;
```

```
13.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
14.         double retorno1 =
PCMath.booleanToDouble(valor1);
15.         double r1 =
PCMath.booleanToDouble(valor2);
16.         //Compara o valor retornado com o
esperado
17.         assertEquals(esperado1, retorno1, 0);
18.         assertEquals(e1, r1, 0);
19.   }
20.
21.   public void testFactorial(){
22.         //Valores que serão testados
23.         double valor2 = 5;
24.         double esperado2 = 120;
25.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
26.         double retorno2 =
PCMath.factorial(valor2);
27.         //Compara o valor retornado com o
esperado
28.         assertEquals(esperado2, retorno2, 0);
29.   }
30.
31.   public void testAdd(){
32.         //Valores que serão testados
33.         double valor3left = 5.0;
34.         double valor3right = 8.0;
35.         double esperado3 = 13.0;
36.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
37.         double retorno3 = PCMath.add(valor3left,
valor3right);
38.         //Compara o valor retornado com o
esperado
39.         assertEquals(esperado3, retorno3, 0);
40.   }
41.
42.   public void testSubtract(){
43.         //Valores que serão testados
44.         double valor4left = 70.0;
45.         double valor4right = 35.5;
46.         double esperado4 = 34.5;
47.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
```

```
48.         double retorno4 =
PCMath.subtract(valor4left, valor4right);
49.         //Compara o valor retornado com o
esperado
50.         assertEquals(esperado4, retorno4, 0);
51.   }
52.
53.   public void testMultiply(){
54.         //Valores que serão testados
55.         double valor5left = 30.0;
56.         double valor5right = 3.0;
57.         double esperado5 = 90.0;
58.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
59.         double retorno5 =
PCMath.multiply(valor5left, valor5right);
60.         //Compara o valor retornado com o
esperado
61.         assertEquals(esperado5, retorno5, 0);
62.   }
63.
64.   public void testDivide(){
65.         //Valores que serão testados
66.         double valor6left = 21.0;
67.         double valor6right = 3.0;
68.         double esperado6 = 7.0;
69.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
70.         double retorno6 =
PCMath.divide(valor6left, valor6right);
71.         //Compara o valor retornado com o
esperado
72.         assertEquals(esperado6, retorno6, 0);
73.   }
74.
75.   public void testLogicalAnd(){
76.         //Valores que serão testados
77.         double valor7left = 5.0;
78.         double valor7right = 9.0;
79.         double esperado7 = 0.0;
80.         double e7 = 1.0;
81.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
82.         double retorno7 =
PCMath.logicalAnd(valor7left, valor7right);
```

```
83.         double r7 = PCMath.logicalAnd(valor7left,
valor7left);
84.         //Compara o valor retornado com o
esperado
85.         assertEquals(esperado7, retorno7, 0);
86.         assertEquals(e7, r7, 0);
87.   }
88.
89.   public void testLogicalOr(){
90.         //Valores que serão testados
91.         double valor8left = 3.0;
92.         double valor8right = 5.0;
93.         double esperado8 = 1.0;
94.         double e8 = 0.0;
95.         //Executa método booleanToDouble da
classe PCMath e armazena resultado.
96.         double retorno8 =
PCMath.logicalOr(valor8left, valor8right);
97.         double r8 = PCMath.logicalOr(valor8left,
valor8left);
98.         //Compara o valor retornado com o
esperado
99.         assertEquals(esperado8, retorno8, 0);
100.        assertEquals(e8, r8, 0);
101.  }
102.
103.  public void testLogicalXOr(){
104.        //Valores que serão testados
105.        double valor9left = 3.0;
106.        double valor9right = 5.0;
107.        double esperado9 = 0.0;
108.        double e9 = 1.0;
109.        //Executa método booleanToDouble da
classe PCMath e armazena resultado.
110.        double retorno9 =
PCMath.logicalXOr(valor9left, valor9right);
111.        double r9 = PCMath.logicalXOr(valor9left,
valor9left);
112.        //Compara o valor retornado com o
esperado
113.        assertEquals(esperado9, retorno9, 0);
114.        assertEquals(e9, r9, 0);
115.  }
116.}
```

Código-fonte 4 Classe de teste JUnit PCMathTest

Ao realizar os testes o JUnit apresenta um resultado detalhado de cada método testado, como mostra a Figura 8.

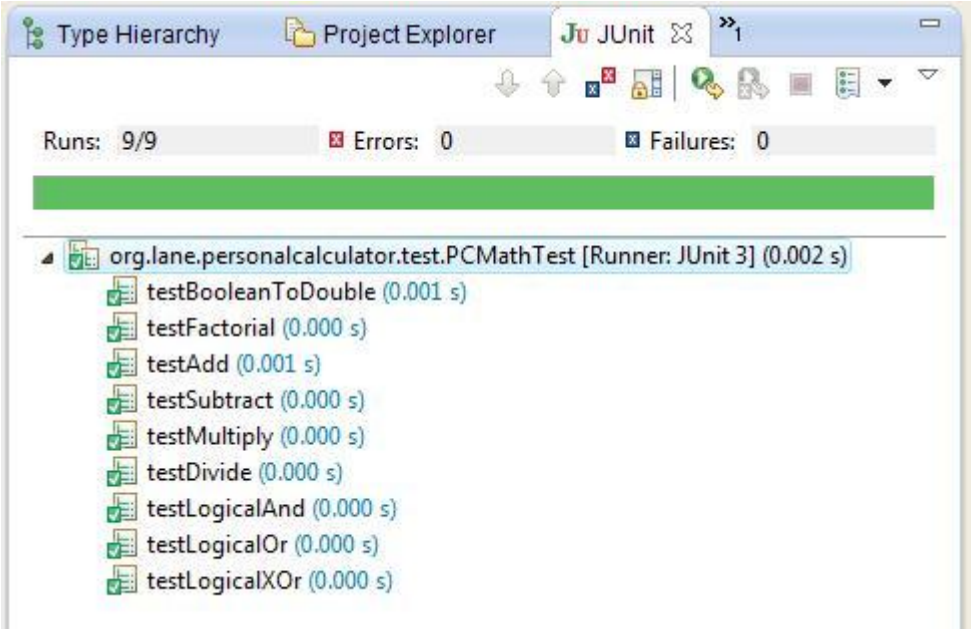


Figura 8 Resultado apresentado pelo JUnit para a classe de teste “PCMathTest”

Quando um dos métodos de teste apresenta erro, o Junit apresenta um relatório com as falhas, indicando o resultado esperado e o resultado efetivamente retornado. Pela Figura 9, observa-se que é mostrado com um ‘x’ o método que apresentou a falha e abaixo onde se encontra o relatório, há a mensagem indicando o resultado esperado como “0.0” no método “testBooleanToDouble” e indicando também que o resultado que realmente foi retornado “1.0”, diminuindo assim o trabalho para detectar a causa do erro e facilitando a correção do mesmo.

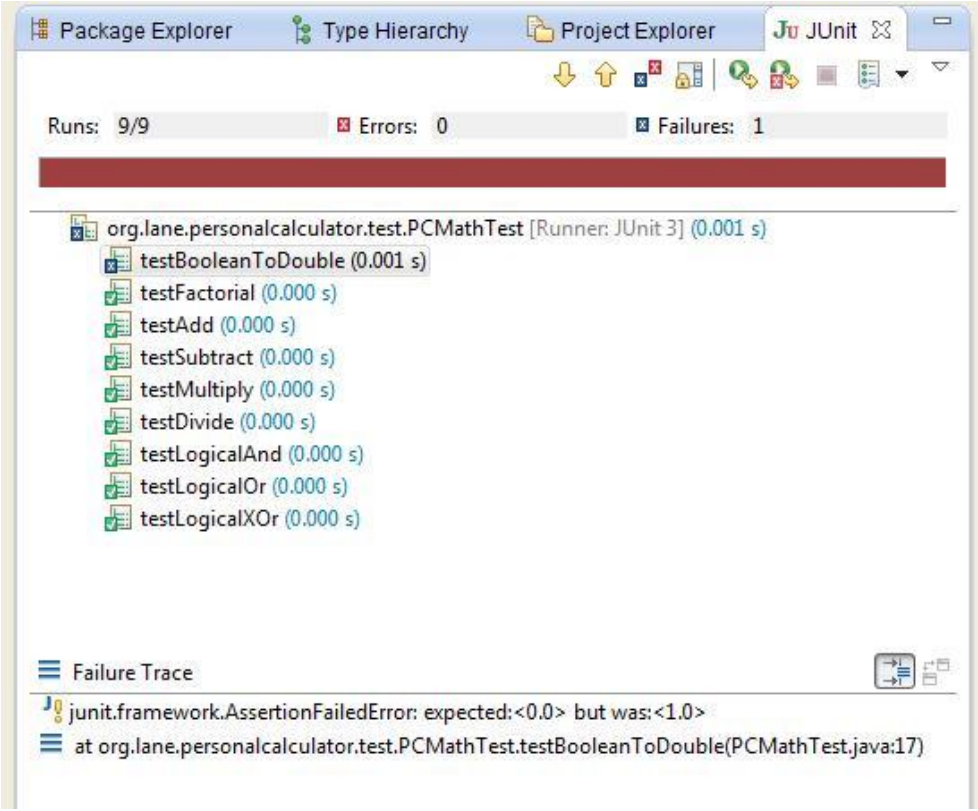


Figura 9 Erro no método "testBooleanToDouble"

6.3 Aplicação do plugin EclEmma no Eclipse

Para a aplicação do *plugin* de cobertura EclEmma no Eclipse o download do mesmo foi feito através do próprio Eclipse, na guia "Help", opção "Install New Software...". Para mostrá-lo na prática foram feitos testes JUnit no mesmo projeto "PersonalCalculator", onde o EclEmma mostra qual a cobertura dos testes.

O *plugin* é iniciado clicando com o botão direito do mouse sobre a classe de teste JUnit do projeto que se deseja, escolhendo a opção "Coverage

As” e “JUnit”, será então realizado o teste JUnit e exibido o resultado. Na aba “Coverage”, é ilustrado a porcentagem de cobertura do teste e na classe testada, são colorida de verde as linhas que foram cobertas e de vermelho as que não foram cobertas pelo teste, e ainda de amarelo as linhas que foram parcialmente cobertas pelo teste. O resultado da cobertura na classe “PCMath.java”, com as linhas que foram cobertas e as que não foram pode ser visualizado no APÊNDICE D.

Para maior clareza segue o método “booleanToDouble” isolado, que transforma um tipo ”boolean” no tipo “double”. Segue o método com as indicações de linha do EclEmma (Código-fonte 5):

```
public static Double booleanToDouble(boolean arg0) {
        Double result = 0.0;

        if(arg0) {
                result = 1.0;
        } else {
                result = 0.0;
        }

        return result;
}
```

Código-fonte 5 Cobertura do método booleanToDouble

A seguir o método JUnit que faz o teste deste método (Código-fonte 6):

```
	public void testBooleanToDouble(){
		//Valores que serão testados
		boolean valor1 = false;
		double esperado1 = 0.0;
		//Executa método booleanToDouble da classe
PCMath e armazena resultado.
		double retorno1 =
PCMath.booleanToDouble(valor1);
		//Compara o valor retornado com o esperado
		assertEquals(esperado1, retorno1, 0);
	}
```

Código-fonte 6 Método testBooleanToDouble que testa o método booleanToDouble

Na observação do método de teste, verifica-se que o teste só verifica um possível valor da variável de entrada. É conveniente observar as cores das linhas do método onde a linha amarela demonstra uma cobertura parcial, pois como analisado só um valor da variável foi coberto, e ele só entrará em uma condição, onde a linha foi demonstrada de verde, a outra linha que não chegou a ser executada está colorida de vermelho.

Esta é uma importante função da ferramenta EclEmma, que auxilia a incrementar os testes para que haja uma maior cobertura do código testado. Para a resolução deste problema é necessário fazer com que o método de teste passe pela linha que não foi coberta anteriormente, abaixo o método com a modificação necessária (Código-fonte 7):

```
		public void testBooleanToDouble(){
			//Valores que serão testados
			boolean valor1 = false;
			boolean valor2 = true;
			double esperado1 = 0.0;
			double e1 = 1.0;
			//Executa método booleanToDouble da
classe PCMath e armazena resultado.
			double retorno1 =
PCMath.booleanToDouble(valor1);
```

```
                double r1 =
PCMath.booleanToDouble(valor2);
                //Compara o valor retornado com o
esperado
                assertEquals(esperado1, retorno1, 0);
                assertEquals(e1, r1, 0);
        }
```

Código-fonte 7 Método testBooleanToDouble melhorado

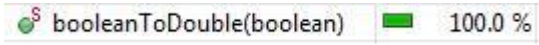
O método de teste agora, aborda dois valores, que são suficientes para que sejam testadas todas as funções do método em questão, como pode ser visto pelo retorno que o plugin demonstra:

```
public static Double booleanToDouble(boolean arg0) {
        Double result = 0.0;

        if(arg0) {
                result = 1.0;
        } else {
                result = 0.0;
        }

        return result;
}
```

Código-fonte 8 Cobertura do método booleanToDouble

Agora todas as linhas do método foram coloridas de verde, indicando que o método de teste obteve uma cobertura de 100%. A figura mostra a cobertura do método antes da modificação do teste:

E a cobertura após a modificação:



Ao final o ideal é uma cobertura de 100% em todos os métodos da classe testada como foi possível na classe “PCMath” utilizada como exemplo. A Figura 10 mostra o resultado obtido neste exemplo.

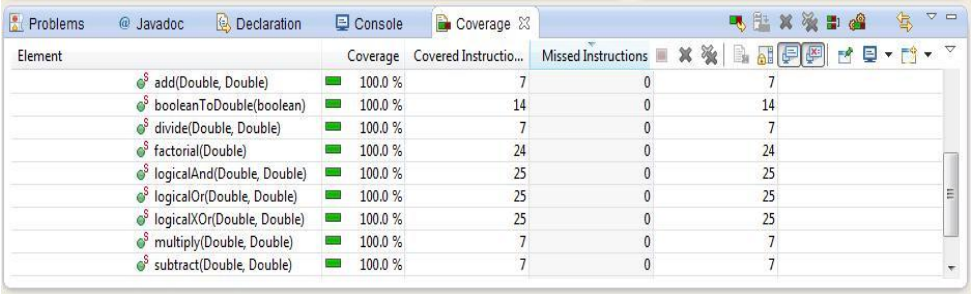


Figura 10 Resultado da cobertura do teste na classe “PCMath”

6.4 Aplicação do plugin Sureassert no Eclipse

O site do *plugin* disponibiliza um tutorial de fácil entendimento, que demonstra que a construção dos testes é bastante simplificada pela anotação.

A instalação do *plugin* é muito simples, feita através do próprio Eclipse pela guia “Help” e a escolha da opção “Install New Software...”. Após a instalação, é necessário apenas habilitar o plugin no projeto em que se deseja aplicá-lo, para isso basta um clique no projeto com o botão direito, acessar a opção “Sureassert” e “Enable”.

Para aplicação no projeto “Calculator”, optou-se pela classe “PCMath”, onde há os métodos que realizam as operações, para o teste das anotações do Sureassert.

Os métodos da classe, e as respectivas anotações do *plugin* podem ser visualizadas no APÊNDICE E. Abaixo segue ,como exemplo, o método booleanToDouble e sua anotação correspondente (Código-fonte 9):

```
@Exemplar( args={"false"}, expect={"0.0"})
public static Double booleanToDouble(boolean arg0) {
Double result = 0.0;


if(arg0) {
result = 1.0;
} else {
result = 0.0;
}

return result;
}
```

Código-fonte 9 Método booleanToDouble e anotação do plugin Sureassert

A anotação “Exemplar” recebe expressões SIN, que como já foi citado, facilita a criação de objetos, e podem chamar os métodos e construtores. Em cada anotação foi passado os argumentos que seriam usados no respectivo método, e o valor que seria esperado de retorno, desta maneira o *plugin* entra com os argumentos passados e compara o valor esperado com o valor que foi retornado na execução do método, e da um retorno para o programador.

Caso o *plugin* tenha encontrado o valor esperado é mostrado na margem esquerda um sinal verde como visto na linha 6, caso contrário é o sinal de um “X” que é mostrado na linha 19, como pode ser conferido na Figura 11.

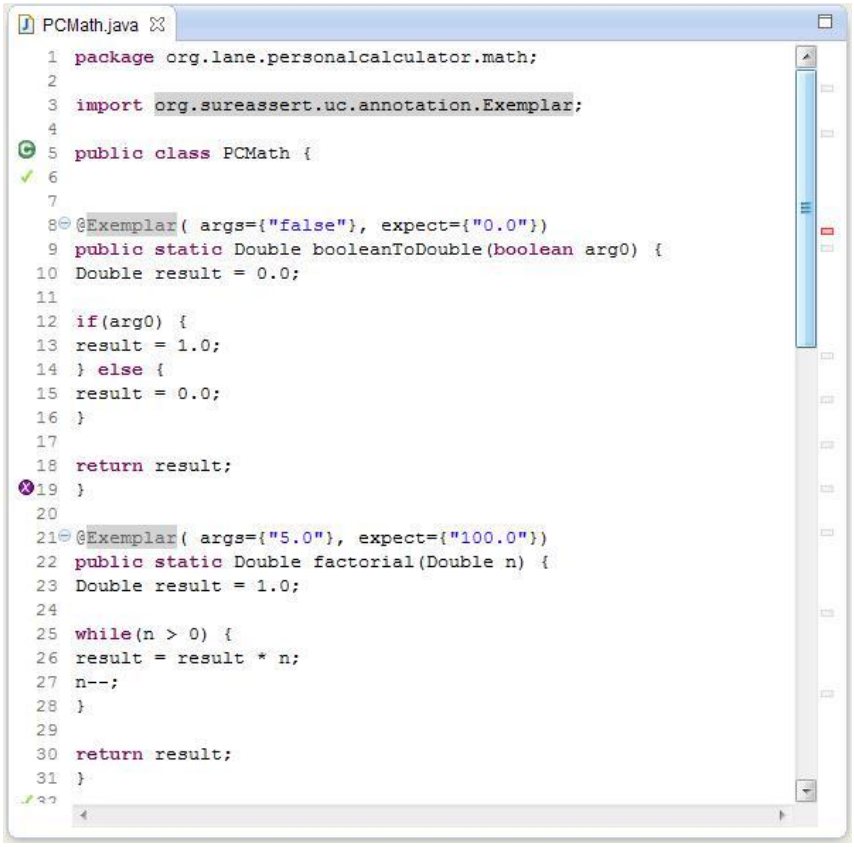


Figura 11 Ilustração da aplicação do plugin Sureassert

O primeiro método é o “booleanToDouble”, que transforma um booleano passado em um tipo Double. Na anotação passou-se como entrada o tipo booleano “false”, e como valor esperado de saída “0.0”, que como pode ser visto na Figura 11 é o próprio resultado que será retornado. Assim a marcação apresentada pelo *plugin* foi o sinal verde, significando que o teste passou. O segundo método é o “factorial”, que realiza o cálculo do fatorial do número de entrada, na anotação do *plugin* o argumento de entrada foi o número 5, cujo fatorial é 120, porém verificando-se a Figura 11, o valor repassado como o

esperado foi 100, logo quando é comparado com o retorno do método não confere e o Sureassert apresenta o símbolo que demonstra a falha do teste.

6.5 Aplicação do plugin JUnit no NetBeans

Para a elaboração dos testes, a versão do NetBeans adotada foi a 7.1, que não dispõe do JUnit incluído por padrão.

O funcionamento do JUnit nas duas IDE's é muito semelhante, porém no NetBeans há uma função de "criar testes JUnit" que já apresenta uma classe com toda a estrutura do teste elaborada, mais completa que a opção do Eclipse, com testes para todos os métodos da classe selecionada para a construção, sendo necessário apenas inserir os valores que devem ser usados no teste. A classe de teste é criada em uma pasta chamada "Pacotes de teste", e o nome da classe é o mesmo da que está sendo testada acrescida de "Test" ao final, tudo criado automaticamente.

Para este teste no NetBeans, foi utilizado o JUnit 4, diferente do teste realizado no Eclipse, onde foi aplicado o teste com o JUnit 3. Houveram algumas mudanças de uma versão para a outra, por exemplo, no JUnit 4 não é mais necessário estender TestCase para implementar os testes, uma classe Java simples pode ser usada para os testes sendo preciso uma anotação "@Test".

Outro ponto é que como as classes JUnit não herdam o TestCase, o método Assert, não está disponível, para utilizá-lo é preciso usar a sintaxe prefixada "Assert.assertEquals()" ou uma importação estática "import static org.junit.Assert.*;".

A classe PCMathTest completa está disponível no APÊNDICE F.

Segue o método de teste booleanToDouble da classe de teste "PCMathTest.java" criada para teste na IDE NetBeans:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package org.lane.personalcalculator.math;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import org.junit.*;

/**
 *
 * @author Carolina
 */
public class PCMathTest {

    public PCMathTest() {
    }

   /**
     * Test of booleanToDouble method, of class
PCMath.
     */
    @Test
    public void testBooleanToDouble() {
        System.out.println("booleanToDouble");
        boolean arg0 = false;
        Double expResult = 0.0;
        Double result = PCMath.booleanToDouble(arg0);
        assertEquals(expResult, result);
    }
```

A Figura 12 apresenta o resultado da execução do teste JUnit:

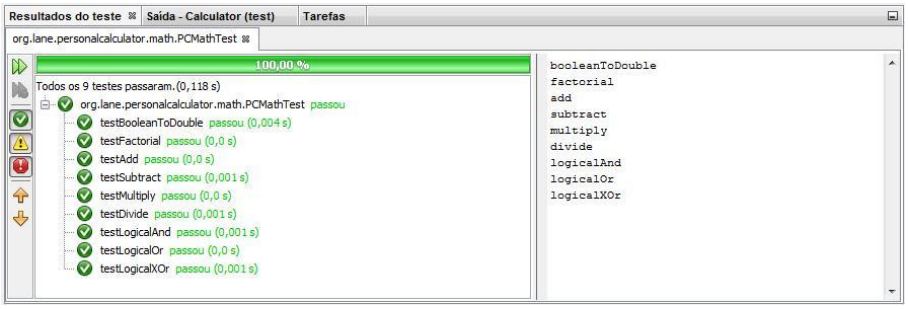


Figura 12 Resultado do teste JUnit

6.6 Aplicação do plugin Unit Tests Code Coverage no NetBeans

O Unit Tests Code Coverage, assim como o EclEmma é um *plugin* de cobertura de testes de unidade porém para a IDE NetBeans, também é baseado em Emma. Será apresentado o *plugin* em ação no NetBeans para o mesmo projeto e com a mesma classe de teste que foi utilizada na aplicação no EclEmma, para que seja possível uma comparação das duas ferramentas.

A Figura 13 ilustra o relatório exposto pela ferramenta ao ser utilizada no projeto PersonalCalculator, como apresentado anteriormente a classe de teste construída testa apenas a classe "PCMath.java". Analisando a Figura 13, percebe-se que o relatório é menos detalhando que da ferramenta EclEmma, ele é exibido por pacote do projeto, não sendo possível expandi-lo como na equivalente do Eclipse, onde é admissível a expansão para as classes do pacote, e mais ainda para os métodos de cada uma, o que facilita a visualização da cobertura.

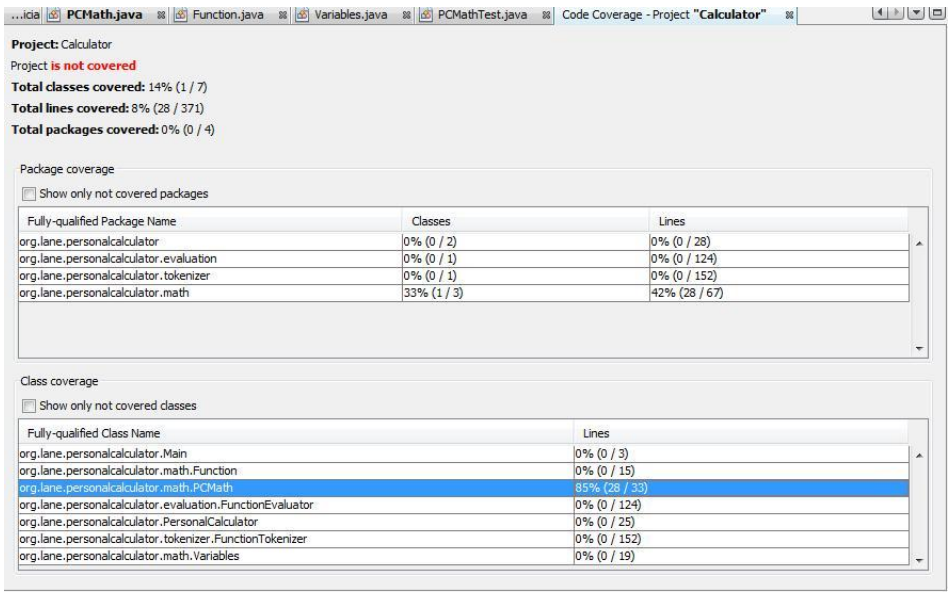


Figura 13 Relatório de retorno da ferramenta

Semelhante ao EclEmma, o Unit Tests Code Coverage também nos indica quais linhas foram cobertas e quais não foram, as linhas cobertas são coloridas de verde, e as linhas que não foram cobertas não recebem nenhuma cor. Neste plugin, ao contrário do EclEmma as linhas parcialmente cobertas não são destacadas.

Para uma melhor visualização, como no exemplo do Eclipse, será mostrado isoladamente a cobertura do método “booleanToDouble” (Figura 14).

```
6  public static Double booleanToDouble(boolean arg0) {
       Double result = 0.0;
8
       if(arg0) {
10             result = 1.0;
11     } else {
               result = 0.0;
13     }
14
       return result;
16 }
```

Figura 14 Ilustração da cobertura do método "booleanToDouble"

Não é explícito que as linhas 10 e 11 não foram cobertas, mas é possível inferir pela ausência da coloração verde. É um resultado semelhante ao encontrado com o EclEmma. Na classe de teste foi passado como valor somente o valor "false" para ser testado, assim o método não passa pelo primeiro IF. Para um teste completo é recomendado testar valores que caiam nos dois casos. A Figura 15 mostra o método de teste complementado para realizar o teste completo, testando os dois casos possíveis.

```
39     public void testBooleanToDouble() {
40         System.out.println("booleanToDouble");
41         boolean arg0 = false;
42         boolean arg01 = true;
43         Double expResult = 0.0;
44         Double expResult2 = 1.0;
45         Double result = PCMath.booleanToDouble(arg0);
46         Double result2 = PCMath.booleanToDouble(arg01);
47         assertEquals(expResult, result);
48         assertEquals(expResult2, result2);
49     }
```

Figura 15 Método de teste do método "booleanToDouble"

A partir do novo método de teste mostrado(Figura 15), resultou uma cobertura maior do teste sendo cobertas todas as linhas do método testado. A Figura 16 aponta o resultado da nova cobertura do método, onde nota-se uma cobertura completa do método testado.

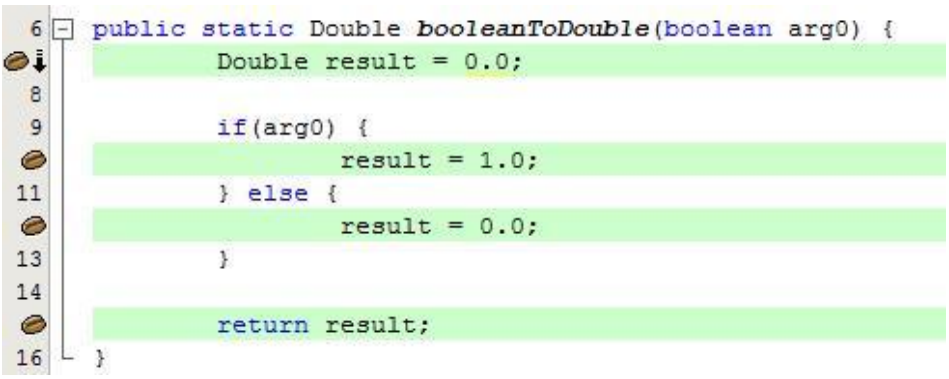
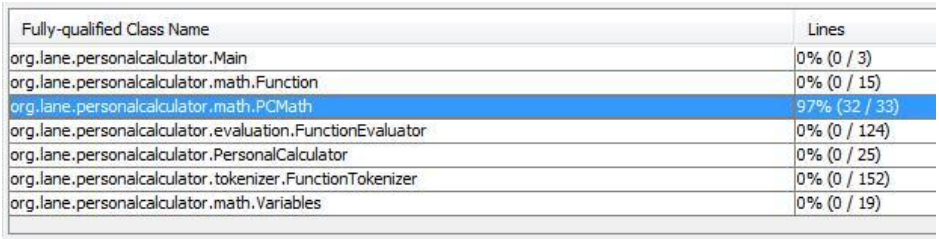


Figura 16 Cobertura do método “booleanToDouble”

Feitas as correções nos métodos da classe de teste para cobertura das linhas que não estavam sendo cobertas, a porcentagem de cobertura indicada pelo plugin aumentou na classe “PCMath.java”, antes de 85% passou, com os ajustes, para 97%, como indicado na Figura 17.



Fully-qualified Class Name	Lines
org.lane.personalcalculator.Main	0% (0 / 3)
org.lane.personalcalculator.math.Function	0% (0 / 15)
org.lane.personalcalculator.math.PCMath	97% (32 / 33)
org.lane.personalcalculator.evaluation.FunctionEvaluator	0% (0 / 124)
org.lane.personalcalculator.PersonalCalculator	0% (0 / 25)
org.lane.personalcalculator.tokenizer.FunctionTokenizer	0% (0 / 152)
org.lane.personalcalculator.math.Variables	0% (0 / 19)

Figura 17 Porcentagem de cobertura da classe “PCMath.java”

7 RESULTADOS

Com o aumento da procura pela qualidade e confiabilidade do software, há um aumento do investimento em testes. Resultante desses investimentos hoje se tornou muito acessível o teste de software principalmente integrado com a IDE, caracterizando-se o teste de unidade, que na maior parte dos casos já vem por default no ambiente de desenvolvimento.

Alguns *plugins* já vêm com a opção de executarem até em segundo plano, automaticamente enquanto o código é criado, facilitando ainda mais a vida do programador, que não corre mais o risco de esquecer de executar os testes adequados.

Com relação ao *plugin* JUnit apresentado, é visível a evolução da ferramenta, comprovando que há investimentos e um crescente uso, a sintaxe de escrita dos testes na versão JUnit4 por exemplo, se tornou mais simples com o uso de anotações. O uso dos nomes dos métodos se tornou também mais maleável, uma vez que métodos de teste são indicados pela anotação "@Test", não mais sendo requerido o prefixo test. Outra facilidade apresentada pela nova versão são as anotações "@BeforeClass" e "@AfterClass" que fixam métodos para serem executados antes e depois de executar a classe de teste.

A ferramenta SureAssert oferece testes de unidade simplificados, por meio de anotações na própria classe que está em teste, dispensando a criação de uma nova classe com esta finalidade. É um *plugin* para a IDE Eclipse, e não foi encontrado uma ferramenta semelhante que estivesse disponível para a IDE NetBeans.

Os *plugins* de cobertura de teste EclEmma e Unit Tests Code Coverage funcionam nas IDEs Eclipse e NetBeans respectivamente, a primeira é atualmente baseada na biblioteca JaCoCo e a segunda na Emma, e ambas adaptadas para as IDEs. Os *plugins* demonstraram algumas diferenças no

funcionamento em cada IDE, como por exemplo o EclEmma ilustra a cobertura, colorindo as linhas cobertas de verde, as parcialmente cobertas de amarelo e as que não foram cobertas de vermelho; já o Unit Tests Code Coverage só colore as linhas totalmente cobertas de verde, enquanto as demais não apresentam nenhuma coloração. Outra diferença entre eles, é o relatório de cobertura, enquanto o do EclEmma é mais detalhado com a porcentagem de cobertura de cada classe separada, assim como de cada método da classe, no relatório do *plugin* do NetBeans só é mostrada a das classes, não sendo possível vizualizar a cobertura dos métodos separadamente.

O engenheiro de software que estiver em busca de um *plugin* que possa facilitar o seu trabalho de teste de unidade ou de cobertura, pode conhecer, por meio desta monografia, o funcionamento dos principais, a sua forma de exibição do resultado, avaliando assim em um primeiro momento qual forma seria mais interessante para a execução do seu trabalho.

Após a escolha do plugin desejado, o trabalho ainda pode ser uma ferramenta para apoiar a implementação, desde a hora da instalação do mesmo até o seu primeiro uso.

7.1 Quadro-resumo do estudo realizado

A seguir um quadro que apresenta resumidamente os plugins estudados.

Plugins	**Licença**	**Técnica de teste aplicada**	**Nível de teste**	**Detalhamento**	**IDE suportada**
Unit Tests Code Coverage	Common Development and Distribution License	Teste caixa-branca	Teste de unidade	• Internamente necessita da biblioteca do Emma code coverage; • Seu funcionamento ocorre através de classe instrumentada; • Emma pode instrumentar arquivos .class individuais ou .jar inteiros.	NetBeans
EclEmma	Eclipse Public License	Teste caixa-branca e caixa-preta	Teste de cobertura	• Desde a versão 2.0 é baseada na biblioteca JaCoCo; • Esta abordagem permite uma instrumentação *on the fly;* • Existem algumas limitações nesta abordagem, como por exemplo os arquivos de classe devem ser	Eclipse

Plugins	**Licença**	**Técnica de teste aplicada**	**Nível de teste**	**Detalhamento**	**IDE suportada**
				compilados com informações de depuração para calcular o nível da linha de cobertura. Nem todas as construções Java podem ser compiladas diretamente para o *bytecode* correspondente, neste caso, o compilador cria o chamado código sintético que muitas vezes retorna resultados inesperados; • Originalmente o EclEmma era baseado na bilioteca Emma.	
JUnit	Common Public License	Teste caixa-branca	Teste de unidade	• As duas principais classes do JUnit são: TestCase e TestSuite. • Essas classes implementam a interface Test que contém o método *run* responsável pela execução dos testes;	Eclipse e NetBeans

Plugins	Licença	Técnica de teste aplicada	Nível de teste	Detalhamento	IDE suportada
				• Toda vez que um caso de teste é executado no JUnit ele: 1. Executa o método SetUp(): serve para inicializar variáveis e criar objetos. 2. Executa o próprio caso de teste: serve para testar um método de uma classe. 3. Executa o método TearDown(): serve para finalizar corretamente o método testado. • O JUnit repete esse procedimento usando reflexão até encontrar todos os métodos de teste; • As expectativas dos testes são validadas através dos métodos da classe Assert; • O JUnit registra todo o caminho percorrido durante as falhas obtidas nos métodos da classe e relata	

Plugins	**Licença**	**Técnica de teste aplicada**	**Nível de teste**	**Detalhamento**	**IDE suportada**
				os resultados após a execução de todos os testes. • A partir da versão 4, a classe de teste não precisa mais herdar a classe TestCase. Isto foi substituído por import static org.junit.Assert.* ; • O método de teste não precisa mais iniciar pela palavra "test", a anotação @Test já indica o método de teste.	
Sureassert	Gratuito e open-source	Gratuito e open-source	Teste de unidade	• Trata os testes de unidade como parte do contrato de cada método e executa-os automaticamente;	Eclipse

Plugins	**Licença**	**Técnica de teste aplicada**	**Nível de teste**	**Detalhamento**	**IDE suportada**
				• As anotações *Exemplars* Sureassert oferecem uma verdadeira capacidade de teste isolando a funcionalidade do método, são definidos como contratos de código declarativos que não necessitam do código de teste; • A engenharia UC gera e executa testes e códigos *stub* baseados no conjunto de classes, métodos e anotações fornecidas pela ferramenta. Faz isso de forma automática conectando no Eclipse o processo de compilação incremental; • As anotações são utilizadas para definir testes para os métodos da classe. O teste passa a ser uma parte integrante do processo de desenvolvimento, os erros de compilação e os erros do teste são visualizados da mesma maneira.	

8 CONCLUSÕES

O objetivo deste trabalho foi elaborar uma análise exploratória de alguns *plugins* de teste, tendo como finalidade reunir as principais características e técnicas de determinadas ferramentas. Os critérios adotados para a seleção foram a gratuidade, os que realizavam teste de unidade ou de cobertura e aqueles que trabalhavam com *softwares* na linguagem java.

Dentre os *plugins* gratuitos, de teste de unidade e de cobertura, selecionou-se o JUnit, o Sureassert, o EclEmma e o Unit Tests Code Coverage que são aqueles com maior número de usuários.

Para a consecução dos objetivos realizou-se uma revisão bibliográfica, expondo os principais termos relacionados ao teste de software e conceituando-os; e um levantamento de trabalhos relacionados. Em seguida, foram identificados *plugins* de teste do NetBeans e do Eclipse, em uma busca nos sites oficiais das IDEs, dos quais somente quatro foram avaliados.

A partir do resultado deste TCC, os engenheiros de *software* poderão ter o trabalho de seleção de *plugins* facilitado. Como trabalhos futuros segere-se avaliar outros *plugins*, envolvendo outras técnicas; e identificar técnicas de testes que ainda não são suportadas por nenhum *plugin*.

REFERÊNCIAS BIBLIOGRÁFICAS

BANDERA, A. V. M. D; OLIVEIRA, D. A. Z.; SILVA, J. N. **Um estudo sobre os aspectos teóricos e práticos de teste de software.** 2008. 68p. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade Federal De Mato Grosso Do Sul, Mato Grosso do Sul, 2008.

BOEHM, B. Software Engineering Economics, Prentice-Hall, 1981.

DEVMEDIA. **Portal sobre programação.** Disponível em <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acessado em 30 de março de 2012.

ECLEMMA. **Site da ferramenta.** Disponível em <http://www.eclemma.org/index.html>. Acessado em 20 de março de 2012.

ECLIPSE. **Loja de plugins.** Disponível em: <http://marketplace.eclipse.org/>. Acessado em 25 de fevereiro de 2012.

EMMA. **Site da ferramenta.** Disponível em <http://emma.sourceforge.net/>. Acessado em 20 de março de 2012.

FEWSTER M.; GRAHAM D. Software Test Automation. Effective Use of Test Execution Tools. ACM Press, New York, 1999.

JAVA. **Projetos Java.** Disponível em <http://java.net/projects/personalcalculator>. Acessado em 30 de março de 2012.

JIN, H; ZENG, F. **Research on the Definition and Modelo f Software Testing Quality.** College of Reliability and System Engineering, 2011.

LOUSA, H. A. A.; NUNES, C. T. **Automação de Testes Utilizando Ferramentas Open Source.** Universidade de Brasília, 2006.

NETBEANS. **Portal de plugins.** Disponível em: <http://plugins.netbeans.org/>. Acessado em 25 de fevereiro de 2012.

PRESSMAN, R. Engenharia de software. 6. Ed. Mc Graw Hill, 2005.

PRESSMAN, R. S. Engenharia de Software. São Paulo: Makron Books, 2000.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. et al., Qualidade de software – Teoria e prática, Prentice Hall, São Paulo, 2001.

SOMMERVILLE, I. Engenharia de Software, 6ª Edição São Paulo: Addison Wesley, 2003.

SUREASSERT. **Site da ferramenta.** Disponível em: <www.sureassert.com>. Acessado em 20 de março de 2012.

SWEBOK, Guide to the Software Engineering Body of Knowledge, 2004.

UNIVERSIDADE FEDERAL DE MATO GROSO DO SUL. **Um estudo sobre os aspectos teóricos e práticos de teste de software.** Mato Grosso do Sul, 2008. Disponível em: <http://qualipso.icmc.usp.br/files/monografia.pdf>. Acesso em 25 de fevereiro de 2012.

ZUO, **R. Information Theory, Information View, and Software Testing.** Seventh International Conference on Information Technology. Cisco Systems, Inc. San Jose, CA, USA, 2010.

APÊNDICES

APÊNDICE A – Tabela com relação dos plugins para Eclipse

Plugin	**Descrição**	**Licença**
BlackBerry Java Plugin for Eclipse	Permite escrever, compilar e testar aplicativos Java ME que rodam em *smartphones* BlackBerry. Simula a experiência do usuário final-de-final *online* e *offline*. Características da integração dos principais Java *Specification Requests* (JSRs).	*Commercial*.
eFitNesse - Testing framework for OSGI and embedded Java applicantions	E um *framework* de teste para aplicações OSGi que permite aplicar diferentes tipos de suíte de teste como JUnit ou testes de aceitação FitNesse.	*Free General Public License*
Squish	Ferramenta automatizada de teste GUI para testes funcionais de regressão.	*Commercial*
EclEmma	Ferramenta de analise de cobertura de código para o Eclipse.	Eclipse *Public License*
Cantata ++	Permite executar teste de unidade e de integração. Oferece um conjunto de teste, analise de cobertura e analise estática. Teste funcional e estrutural e testes orientado a objetos. E as métricas para analise de cobertura são: Pontos de entrada; Retorno de chamada; Demonstraçoes; Blocos Basicos; Decisoes(ramos); Condiçoes; MC/DC	*Commercial*

	(para DO-178B).	
Jnario	Tem como objetivo tornar a escrita de especificações executáveis o mais simples possível. Oferece duas linguagens, uma para escrita de especificação de aceitação em alto nível, e outra para especificação de unidade em baixo nível.	Eclipse *Public License*
mBProfiler	E uma ferramenta de otimização de código para projetos JAVA. Eficiência do produto é obtida através da detecção de pontos quentes e gargalos e reduçao do uso de memória e consumo de recursos.	*Commercial*
CodingSpectator	*Plugin* para pesquisa como os desenvolvedores JAVA interagem com a IDE Eclipse da Universidade de Illinois.	*Other Open Source*
CloudBees	Criar e monitorar aplicações na plataforma Jenkins.	*Commercial*
IBM WebSphere Application Server v8.5 Developer Tools Beta 1	Contém ferramentas para gerenciar servidor, e publicar em um servidor local ou remoto, e para controlar a publicação incremental. É um editor de páginas *Rich*, um editor *WYSIWYG* que torna mais fácil para editar arquivos HTML, adicionar widgets Dojo para páginas HTML, criar e editar páginas da web para dispositivos móveis.	*Commercial-Free*
Testdroid Recorder	Permite o teste automático de interface do usuário para aplicativos Android.	
Xcarecrows ITM	Permite a geraçao de *scripts Rational*	*General Public*

4	*Functional Tester* a partir de um modelo UML *Testing*(UTP); sequencia de diagramas UML; scripts unitários que especificam açoes da aplicaçao de teste; um arquivo de multiplicidade para definir todos os casos de teste.	*License*
Xcarecrows IRM 4	Oferece um conjunto de recursos para ligar requisitos aos modelos; sincronizar o modelo com as suas exigências;validar o modelo com os requisitos; rastrear as exigências do projeto.	*General Public License*
Xored Q7	Testes funcionais e testes GUI.	*Commercial*
Jubula	E uma ferramenta para teste UI RCP, SWT, Swing e aplicações web.	*Eclipse Public License*
EMF	Ajuda com a geração de construtores Java para metamodelos EMF. As classes Java geradas seguem uma API fluente.	*Eclipse Public License*
Chronon	Depurador de tempo que permite dar uma "passo atrás" no código.	*Commercial*
JSnapshot	E uma forma de registro, monitoramento e análise.	*Commercial*
Maveryx	E uma ferramenta de teste automatizado para teste funcional e de regressão.	*General Public License*
Sureassert	Teste de unidade. Possibilita o uso de anotações para definir um conjunto de testes para cada um dos métodos de sua classe. SureAssert verifica as anotações e gera os testes apropriados e executá-los.	*Commercial-Free*

EDepChk	Verificador de dependência para arquivos de classe JVM.	*Berkeley Software Distribution*
Rap Corporate Portal	Construtor de aplicações *rich client* e sua implantação para uso na web.	*Commercial-Free*
MailSnag	Depurador de email. Cria um servidor SMTP simples para inspecionar emails enviados por um aplicativo durante o desenvolvimento.	*Eclipse Public License*
SpryTest	É uma solução de testes de unidade automatizados para Java. Permite aos desenvolvedores testar métodos java para a lógica de negócio, validações, exceções, etc. Gera testes Junit proximos dos escritos a mao.	*Commercial*
Oracle Enterprise Pack	Oferece ferramentas que tornam mais fácil desenvolver aplicações que utilizam Oracle *WebLogic Server* e Banco de Dados Oracle.	*Commercial*
Atlassian Clover	Suas principais ferramentas são o teste de cobertura e a otimização de teste.	*Commercial*
eCobertura	Ferramenta de relatórios de cobertura de código Java.	*Eclipse Public License*
GUIdancer	Ferramenta para testes funcionais automatizados, baseada na Jubula, através da GUI.	*Commercial*
MibTiger	É usado para navegar e definir objetos em um dispositivo SNMP.	*Free for non-commercial use*

JUnit Flux	Executa testes JUnit automaticamente quando você salvar sua classe ou quando testá-la.	*Free Eclipse Public License*
JES email Server launcher	Ferramenta para testar programa de serviço de email.	*Free General Public License*
FlexUnit	Ferramenta igual JUnit porém para *Actionscript.*	*Free Common Public License*
Treaty	Verifica *plugins* instalados no Eclipse, sua forma de comunicação, observando o contrato entre eles.	*Commercial*
GrinderStone	Possui capacidade de executar e depurar *scripts* criados para The Grinder.	*Eclipse Public License*
AppPerfect Unit Tester	É um sistema de gerenciamento de teste de unidade. Usa o *framework* de teste JUnit para geração e execução de arquivos Java. Baseado em objetos *mock* e fornece estrutura para definir e gerenciar objetos reutilizáveis.	*Commercial*
AppPerfect Testing/Analysis	É um conjunto para testes funcionais e de carga para aplicações *web* e baseadas em Windows.	*Commercial*
WindowsTester Pro	Simplifica o testes de aplicações *rich client* fornecendo ferramentas para automatizar a gravação, geração de teste, cobertura de código e reprodução de interações GUI.	*Commercial-Free*
CodePro AnalytiX	Análise de código estático; detecção, reparação e relatório de erros; automação de geração de código JUnit;	*Commercial-Free*

	e mais.	
JUnit Factory	Geração de testes JUnit de acordo com o código.	*Commercial*
JDepend4Eclipse	Gera métricas de qualidade para cada pacote Java depois de percorridos um conjunto de classe Java e diretórios de arquivo de origem.	Eclipse *Public License*
HTTP4E	*Plugin* visual para desenvolvimento *web*, *web service* (SOAP e REST), que permite chamadas HTTP do Eclipse, proporcionando ganho no desenvolvimento e teste de aplicações.	*Commercial*
GlobalTester	É uma ferramenta para testar cartões de chip. Permite a execução de um conjunto de script de teste estruturado com XML e definidos usando JavaScript.	*General Public License*
MockCentral	Disponibiliza um ambiente para criar e acessar bibliotecas de objetos *mock* externos ao código de teste, permitindo casos de testes limpos, organização e reutilização de objetos mock.	*Free General Public License*
JDemo	É o *framework* de demonstração Java com conceito semelhante ao JUnit.	*Other Open Source*
Cute C++	É semelhante ao JUnit porém para teste de código C++.	*Commercial-Free*

APÊNDICE B - Tabela com relação dos plugins para NetBeans

Plugins	Descrição	Licença
Unit Tests Code Coverage Plugin	Ferramenta de cobertura de código para testes de unidade baseada no Emma, integrada com JUnit.	*Common Development and Distribution License*
Selenium Module for Maven	Ferramenta de teste Selenium para aplicações *web* baseadas em Maven. Requer Selenium Server instalado.	*GNU GPL v2 ou Common Development Distribution License*
Selenium Module for PHP	Cria novos tipos de arquivos com Selenium para projetos PHP.	*GNU GPL v2 ou Common Development Distribution License*
Selenium Module for Ant Projects	Ferramenta de teste Selenium para aplicações *web* baseadas em Ant.	*GNU GPL v2 ou Common Development Distribution License*
Selenium Server	É indispensável para todos os outros *plugins* Selenium. Permite executar casos de teste Selenium localmente.	*GNU GPL v2 ou Common Development Distribution License*

APÊNDICE C – Código-fonte da Classe PCMath

```
01.package org.lane.personalcalculator.math;
02.
03.public class PCMath {
04.
05.   public static Double booleanToDouble(boolean
arg0) {
06.   Double result = 0.0;
07.
08.   if(arg0) {
09.         result = 1.0;
10.   } else {
11.         result = 0.0;
12.   }
13.
14.   return result;
15.   }
16.
17.   public static Double factorial(Double n) {
18.   Double result = 1.0;
19.
20.   while(n > 0) {
21.         result = result * n;
22.         n--;
23.         }
24.
25.   return result;
26.   }
27.
28.   public static Double add(Double left, Double
right) {
29.         return (left + right);
30.   }
31.
32.   public static Double subtract(Double left,
Double right) {
33.         return (left - right);
34.   }
35.
36.   public static Double multiply(Double left,
Double right) {
```

```
37.         return (left * right);
38.   }
39.
40.   public static Double divide(Double left, Double
right) {
41.         return (left / right);
42.   }
43.
44.   public static Double logicalAnd(Double left,
Double right) {
45.         Double result = 0.0;
46.
47.         result = booleanToDouble(left == right);
48.
49.         return result;
50.   }
51.
52.   public static Double logicalOr(Double left,
Double right) {
53.         Double result = 0.0;
54.
55.         result = booleanToDouble(left != right);
56.
57.         return result;
58.   }
59.
60.   public static Double logicalXOr(Double left,
Double right) {
61.         Double result = 0.0;
62.
63.         result = booleanToDouble(!(left !=
right));
64.
65.         return result;
66.   }
67.
68.}
```

APÊNDICE D – Cobertura da Classe PCMath pelo EclEmma

```
package org.lane.personalcalculator.math;

public class PCMath {

public static Double booleanToDouble(boolean arg0) {
	Double result = 0.0;

	if(arg0) {
		result = 1.0;
	} else {
		result = 0.0;
	}

	return result;
}

public static Double factorial(Double n) {
	Double result = 1.0;

	while(n > 0) {
		result = result * n;
		n--;
	}

	return result;
}

public static Double add(Double left, Double right) {
return (left + right);
}

public static Double subtract(Double left, Double
right) {
return (left - right);
}

public static Double multiply(Double left, Double
right) {
return (left * right);
}

public static Double divide(Double left, Double
right) {
```

```
	return (left / right);
	}

	public static Double logicalAnd(Double left, Double
right) {
		Double result = 0.0;
		Double toleranceAnd = 0.001;

		if (Math.abs(left - right)< toleranceAnd){
			result = 1.0;
		}else{
			result = 0.0;
		}
		return result;
	}

	public static Double logicalOr(Double left, Double
right) {
	Double result = 0.0;
	Double toleranceOr = 0.001;

		if (Math.abs(left - right)> toleranceOr){
			result = 1.0;
		}else{
			result = 0.0;
		}

		return result;
	}

	public static Double logicalXOr(Double left, Double
right) {
	Double result = 0.0;
	Double toleranceXOr = 0.001;

		if (!(Math.abs(left - right)> toleranceXOr)){
			result = 1.0;
		}else{
			result = 0.0;
		}

		return result;
	}

	}
```

APÊNDICE E – Classe PCMath e anotações do plugin Sureassert

```
@Exemplar( args={"false"}, expect={"0.0"})
public static Double booleanToDouble(boolean arg0) {
Double result = 0.0;


if(arg0) {
result = 1.0;
} else {
result = 0.0;
}

return result;
}

@Exemplar( args={"5.0"}, expect={"120.0"})
public static Double factorial(Double n) {
Double result = 1.0;

while(n > 0) {
result = result * n;
n--;
}

return result;
}

@Exemplar( args={"5.0","8.0"}, expect={"13.0"})
public static Double add(Double left, Double right) {
return (left + right);
}

@Exemplar( args={"70.0","35.5"}, expect={"34.5"})
public static Double subtract(Double left, Double
right) {
return (left - right);
}

@Exemplar( args={"30.0","3.0"}, expect={"90.0"})
public static Double multiply(Double left, Double
right) {
return (left * right);
}
```

```
      @Exemplar( args={"21.0","3.0"}, expect={"7.0"})
      public static Double divide(Double left, Double
right) {
      return (left / right);
      }

      @Exemplar( args={"5.0","5.0"}, expect={"0.0"})
      public static Double logicalAnd(Double left, Double
right) {
      Double result = 0.0;

      result = booleanToDouble(left == right);

      return result;
      }

      @Exemplar( args={"3.0","4.0"}, expect={"1.0"})
      public static Double logicalOr(Double left, Double
right) {
      Double result = 0.0;

      result = booleanToDouble(left != right);

      return result;
      }

      @Exemplar( args={"3.0","4.0"}, expect={"0.0"})
      public static Double logicalXOr(Double left, Double
right) {
      Double result = 0.0;

      result = booleanToDouble(!(left != right));

      return result;
      }
```

APÊNDICE F – Classe PCMathTest.java do JUnit utilizada no NetBeans

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package org.lane.personalcalculator.math;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import org.junit.*;

/**
 *
 * @author Carolina
 */
public class PCMathTest {

    public PCMathTest() {
    }

    /**
     * Test of booleanToDouble method, of class
PCMath.
     */
    @Test
    public void testBooleanToDouble() {
        System.out.println("booleanToDouble");
        boolean arg0 = false;
        Double expResult = 0.0;
        Double result = PCMath.booleanToDouble(arg0);
        assertEquals(expResult, result);
    }

    /**
     * Test of factorial method, of class PCMath.
     */
    @Test
    public void testFactorial() {
        System.out.println("factorial");
        Double n = 5.0;
        Double expResult = 120.0;
        Double result = PCMath.factorial(n);
```

```
    assertEquals(expResult, result);
}

/**
 * Test of add method, of class PCMath.
 */
@Test
public void testAdd() {
    System.out.println("add");
    Double left = 5.0;
    Double right = 8.0;
    Double expResult = 13.0;
    Double result = PCMath.add(left, right);
    assertEquals(expResult, result);
}

/**
 * Test of subtract method, of class PCMath.
 */
@Test
public void testSubtract() {
    System.out.println("subtract");
    Double left = 70.0;
    Double right = 35.5;
    Double expResult = 34.5;
    Double result = PCMath.subtract(left, right);
    assertEquals(expResult, result);
}

/**
 * Test of multiply method, of class PCMath.
 */
@Test
public void testMultiply() {
    System.out.println("multiply");
    Double left = 30.0;
    Double right = 3.0;
    Double expResult = 90.0;
    Double result = PCMath.multiply(left, right);
    assertEquals(expResult, result);
}

/**
 * Test of divide method, of class PCMath.
 */
@Test
```

```
    public void testDivide() {
        System.out.println("divide");
        Double left = 21.0;
        Double right = 3.0;
        Double expResult = 7.0;
        Double result = PCMath.divide(left, right);
        assertEquals(expResult, result);
    }

    /**
     * Test of logicalAnd method, of class PCMath.
     */
    @Test
    public void testLogicalAnd() {
        System.out.println("logicalAnd");
        Double left = 5.0;
        Double right = 9.0;
        Double expResult = 0.0;
        Double result = PCMath.logicalAnd(left,
right);
        assertEquals(expResult, result);
    }

    /**
     * Test of logicalOr method, of class PCMath.
     */
    @Test
    public void testLogicalOr() {
        System.out.println("logicalOr");
        Double left = 3.0;
        Double right = 5.0;
        Double expResult = 1.0;
        Double result = PCMath.logicalOr(left,
right);
        assertEquals(expResult, result);
    }

    /**
     * Test of logicalXOr method, of class PCMath.
     */
    @Test
    public void testLogicalXOr() {
        System.out.println("logicalXOr");
        Double left = 3.0;
        Double right = 5.0;
        Double expResult = 0.0;
```

```
            Double result = PCMath.logicalXOr(left,
right);
            assertEquals(expResult, result);
        }
    }
```