CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Renan Batista Teixeira

AUTOMATIZAÇÃO DE TESTE UNITÁRIO DE SOFTWARE: LEVANTAMENTO BIBLIOGRÁFICO

Timóteo

2016

Renan Batista Teixeira

AUTOMATIZAÇÃO DE TESTE UNITÁRIO DE SOFTWARE: LEVANTAMENTO BIBLIOGRÁFICO

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientadora: Deisymar Botega Tavares

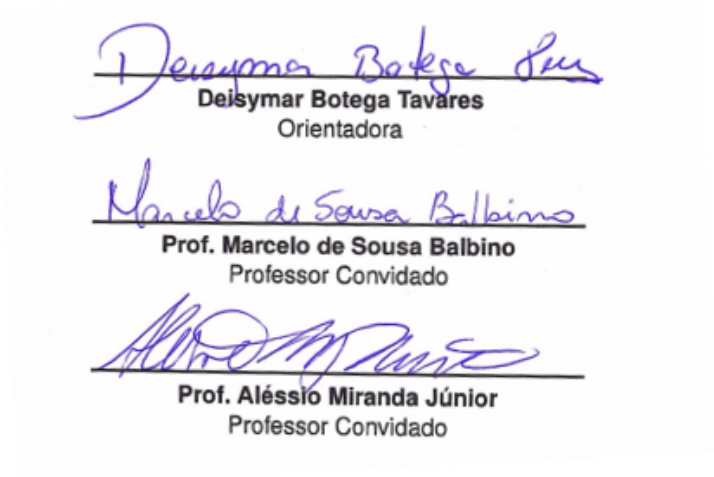
Timóteo

2016

Renan Batista Teixeira

Automatização de Teste Unitário de Software: levantamento bibliográfico

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.



Deisymar Botega Tavares
Orientadora

Prof. Marcelo de Sousa Balbino
Professor Convidado

Prof. Aléssio Miranda Júnior
Professor Convidado

Timóteo
2016

Dedico este trabalho à memória de meu pai Antônio Teixeira Neto,
que foi o meu maior exemplo de vida.

Agradecimentos

Agradeço primeiramente a Deus pelas graças concedidas, aos meus pais que sempre acreditaram em mim, meu irmão pelo companheirismo e apoio, e minha namorada pela paciência. Além dos meus amigos que me acompanharam durante toda minha trajetória acadêmica.

Em especial, agradeço a minha orientadora Deysimar Botega Tavares, que é uma pessoa fantástica e uma excelente profissional.

“Que os vossos esforços desafiem as impossibilidades, lembrai-vos de que as grandes coisas do homem foram conquistadas do que parecia impossível.”.

Charles Chaplin

Resumo

Devido à grande demanda do uso de software pelas mais variadas empresas e organizações aumenta-se também a demanda por programas com cada vez mais qualidade. Sabe-se que o teste de software é um importante recurso para auxiliar na obtenção da qualidade do sistema, no entanto os cursos de graduação na área de TI não exploram de forma mais ampla e profunda as atividades relativas a essa prática, o que pode resultar em desenvolvedores não tão qualificados para a criação de bons testes ou, ainda, para a escolha de técnicas e ferramentas inadequadas pelos profissionais, ocasionando desperdício de tempo e influenciando negativamente no sucesso do software. O presente trabalho dedica-se ao levantamento bibliográfico nas bases científicas Science Direct, IEE e ACM e em revistas técnicas sobre o teste unitário automatizado de software, visando ser um aparato para suprir a deficiência quanto ao conhecimento do tema. Esse levantamento apresenta ferramentas como o JUnit (*framework* de teste unitário mais popular para a linguagem Java) e o Evosuite (executa automaticamente os casos de teste - valores de entrada e saída esperada), além de suas mais relevantes características no processo automático do teste unitário. As técnicas e suas respectivas ferramentas que dão apoio a essa atividade são descritas: teste de mutação, teste de cobertura, complexidade ciclomática e objetos *mock*. Este último isola a determinada unidade para ser avaliada, enquanto os primeiros auxiliam na elaboração de casos de teste eficientes. Com isso, o trabalho é uma fonte bem organizada capaz de direcionar professores, alunos e desenvolvedores na automatização do teste unitário.

Palavras-chave: teste unitário automatizado, casos de teste, ferramentas de teste unitário, levantamento bibliográfico.

Abstract

Due to the great demand of the use of software by the most varied companies and organizations, the demand for programs with high quality is increasing as well. It is known that the software test is a very important resource to help in obtaining the quality of the system, nevertheless the graduation courses in IT don't explore the activities related to it deeply, which may result in not too qualified developers in the creation of good tests or, in the choice of wrong techniques and tools by professionals causing waste of time and bad influence in the success of the software. This paper is dedicated to the bibliographical survey in the scientific bases Science Direct, IEE and ACM and in technical magazines about the automatic unitary software test, aiming to be a mechanism in order to supply the deficiency in the knowledge of the subject. This survey presents tools such as JUnit (the most popular unitary framework test for the Java language) and Evosuite (which executes the tests cases automatically- expected input and output values), besides its most relevant characteristics in the automatic unitary test. The techniques and their respective tools responsible for giving support to this activity are: mutation test, cover test, cyclomatic complexity and mock objects. The latter isolates the particular unit to be evaluated, whereas the former assists in the elaboration of the efficient test cases. So, this paper is a well-organized source capable of directing teachers, students and developers in the automation of unitary test.

Keywords: automated unit test, tests cases, unit test tools, bibliographical survey.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O mundo todo está cada vez mais dependente dos softwares e isso se confirma pelo estreito relacionamento entre sociedade e a tecnologia computacional no cotidiano. As ações mais corriqueiras do dia a dia implicam no uso da tecnologia de processamento de dados. Sacar dinheiro no caixa eletrônico, utilizar o cartão na catraca do ônibus, fotografar com o *smartphone* e passar as compras no caixa do supermercado são apenas alguns exemplos. Destaca-se, também, o uso de sistemas computacionais no âmbito industrial, onde as corporações automatizam os seus processos de gerenciamento, controle, desenvolvimento e produção, além da redução de custos. A comunicação é outro forte exemplo da relevância dos softwares, pois a interligação de pessoas, seja em vídeo, áudio, áudio/vídeo e por texto, só é factível graças ao desenvolvimento da tecnologia de programação.

Casos há, em que os sistemas computacionais podem apresentar alguma falha, gerando impacto negativo, até mesmo, de alcance global. Com isso, um software defeituoso pode gerar prejuízos financeiros e de tempo, além de prejudicar a reputação das empresas perante o público, tanto daquela que desenvolveu o programa quanto daquela que fez uso do mesmo. Em alguns casos as perdas podem ser incalculáveis conforme o emprego que se faz do software defeituoso, podendo causar danos, por exemplo, ao meio ambiente e na qualidade de vida das pessoas.

Com todos esses fatores, aumenta-se a exigência sobre organizações desenvolvedoras de softwares para que as mesmas elaborem seus produtos com o maior grau de qualidade e confiabilidade possíveis, de modo que os sistemas operem da forma esperada pelo cliente. Portanto, com o intuito de aferir os atributos do software, certificando-se de sua excelência e estado de uso, é de suma importância, dentro do processo de desenvolvimento, o exercício das atividades de teste das suas fórmulas, processamento e resultados (LUFT, 2012).

Segundo Myers (2004), a atividade de teste é um mecanismo que tem como objetivo encontrar defeitos no software. Tendo a fase de testes cumprido seu objetivo com êxito, é possível então detectar falhas e corrigi-las, além de aprimorar o produto, de modo que ele possa atender de forma plena o desejo do cliente que o requisitou. Contudo, na realização dessa atividade é inviável testar todas as possíveis entradas e analisar se elas estão de acordo com o que se espera, haja vista o grande número de dados a serem controlados, o que pode gerar, consequentemente, maior custo para o desenvolvimento do software. Por isso, segundo Pressman e Maxim (2016), tem-se definido técnicas e critérios que auxiliem na criação de um subconjunto de valores de entradas com potencial para encontrar pelo menos a maioria dos defeitos de um determinado programa.

Para dar suporte a criação, gestão e execução desses subconjuntos de valores de entrada existem técnicas, critérios e ferramentas que automatizam o teste na menor unidade de um sistema de software, conhecido como teste unitário. Proporcionando a elaboração de um conjunto de casos de teste de qualidade, e sua execução de forma automatizada através dos

frameworks da família XUnit. Porém, essa abordagem é feita de forma superficial na literatura acadêmica de Engenharia de Software, ocasionando dificuldades em encontrar esse tipo de conteúdo na forma concentrada e organizada.

Alguns desenvolvedores de software necessitam desse tipo de material para auxilia-los, pois segundo Wang e Offutt (2009), o teste de unidade é realizado por aquele que escreveu a parte do código que está sendo avaliado. A graduação não explora com ênfase a prática de teste de software (WAHID; ALMALAISE, 2011), resultando em desenvolvedores não tão qualificados para a criação de bons testes ou que são sujeitos a escolherem técnicas e ferramentas inadequadas, trazendo desperdício de tempo, além de poder influenciar negativamente no sucesso do software. É evidente então a necessidade de se ter um material sobre teste unitário automatizado, podendo ser um mecanismo que contribua para o desenvolvimento de software e uma base de apoio para alunos e professores de graduação.

1.1 Objetivos

O presente trabalho tem como objetivo realizar o levantamento bibliográfico sobre a automatização do teste unitário. Atingido o objetivo, espera-se como resultado um material de apoio aos docentes para abordagem do tema com maior propriedade nas disciplinas da graduação, visto que nas faculdades é um assunto pouco explorado. Além de ser uma colaboração para os desenvolvedores, podendo auxilia-los na automatização do teste em seu próprio código fonte, levando à redução do tempo e consequentemente dos custos em se testar o software.

Como objetivos específicos, o projeto busca descrever sobre:

- O teste unitário automatizado;
- As técnicas de apoio ao teste unitário automatizado;
- As ferramentas para o teste unitário e para o apoio ao teste unitário.

1.1.1 Questões de Pesquisa

Este trabalho busca responder os seguintes questionamentos sobre o teste unitário automatizado de software:

- Q1: Quais são as ferramentas *open source* de teste unitário mais discutidas na academia?
- Q2: Quais são as ferramentas *open source* de apoio ao teste unitário mais discutidas na academia?
- Q3: Quais são as características das ferramentas *open source* de teste unitário mais discutidas na academia?
- Q4: Quais são as características das ferramentas *open source* de apoio ao teste unitário mais discutidas na academia?

1.2 Estrutura do Texto

No Capítulo 1 foram apresentados a introdução e os objetivos deste trabalho. No Capítulo 2 são apontados os materiais e métodos utilizados no desenvolvimento do trabalho. No Capítulo 3 são descritos os principais conceitos de teste de software. Quanto a pesquisa bibliográfica, esta é dividida nos seguintes Capítulos: o Capítulo 4 faz abordagem sobre automatização do teste de software, o Capítulo 5 exibe estudo sobre automatização do teste unitário e o Capítulo 6 descreve as técnicas e ferramentas de apoio ao teste unitário de software. A análise dos resultados é apresentada no Capítulo 7 e o Capítulo 8 contém as considerações finais.

2 Materiais e Métodos

Este trabalho consiste em uma pesquisa exploratória por meio de uma revisão de literatura sobre teste unitário automatizado, realizado em bases científicas e revistas técnicas. Para a realização dos procedimentos de pesquisa e seleção de artigos relevantes foi seguido o método utilizado por Cota (2014), que é descrito a seguir.

A pesquisa bibliográfica foi realizada nas bases de consultas que são referências em repositórios de trabalhos nas áreas de ciências exatas e de tecnologia. As bases pesquisadas foram: ACM Digital Library, IEEE Xplore e a Science Direct. Além disso, as revistas técnicas Engenharia de Software Magazine, Testing Experience e Professional Tester Magazine também foram fontes utilizadas.

Com o propósito de obter uma melhor qualidade no resultado, os seguintes critérios foram definidos como filtros de pesquisa:

- Artigos publicados entre 1999 e 2016, pois segundo Rafi et al. (2012), "[...] as ferramentas de teste evoluíram na última década e tornaram-se mais poderosas";
- Artigos escritos no idioma inglês ou português;
- Artigos científicos publicados.

De uma forma similar, os seguintes critérios foram definidos para a exclusão:

- Estudos em forma de slides;
- Artigos repetidos;
- Artigos desprovidos de resumo, introdução ou conclusão;
- Publicações de livros, capítulo de livros, resenhas e relatórios.

As palavras chaves de busca foram definidas a partir do estudo nos artigos base encontrados nas pesquisas iniciais que abordavam os assuntos *unit testing*, *tools unit testing* e *automated software testing*. Com isso, foram elaborados 9 (nove) *strings* de buscas para serem utilizadas nas bases de pesquisa científicas, como mostra a Tabela 1.

Tabela 1 – *Strings* de busca para as bases científicas

Número	String
1	(((comparison) AND tools) AND automated testing)
2	(((research) AND tools) AND automated testing)
3	(((tools) AND automated testing) AND survey)
4	((automated software testing) AND survey)
5	((unit testing) AND ((tools) or (tool)))
6	((unit testing) AND automated)
7	((unit testing) AND framework)
8	((unit testing) AND survey)
9	(automated software testing)

Fonte: Autor.

A revista Engenharia de Software Magazine é a única das revistas selecionadas que possui algoritmo de busca, porém não tão sofisticado quanto os das bases científicas. Assim sendo, foi necessário uma adaptação das strings de busca para essa publicação, retirando todos operadores lógicos que estavam presentes na sentença, resultando em 13 sentenças chave de busca como mostra a Tabela 2. No entanto, nas outras revistas, devido à ausência de uma ferramenta de busca foi realizada uma procura através das leituras nos sumários de cada edição.

Tabela 2 – *Strings* de busca para a revista Engenharia de Software Magazine

Número	String
1	comparação ferramentas de teste automatizado
2	ferramenta de teste de unidade
3	ferramenta de teste unitário
4	framework teste de unidade
5	framework teste unitário
6	pesquisa ferramentas teste automatizado
7	survey ferramentas teste automatizado
8	survey teste automatizado de software
9	survey teste de unidade
10	survey teste unitário
11	teste automatizado de software
12	teste de unidade automatizado
13	teste unitário automatizado

Fonte: Autor.

Juntamente com a realização do procedimento de consulta nas fontes, foi feita a seleção dos artigos seguindo os critérios abaixo listados:

- 1º Critério de seleção: leitura do título durante a busca inicial, verificando se o estudo condiz com o tema do trabalho.
- 2º Critério de seleção: leitura do resumo de cada um dos artigos selecionados do primeiro critério, verificando a sua relevância perante o trabalho.

- 3º Critério de seleção: leitura da introdução e conclusão de cada um dos artigos selecionados no segundo critério, verificando se as ideias apresentadas eram significativas para o trabalho.

Todos os artigos selecionados do terceiro filtro foram lidos e classificados com notas de 1 a 5, sendo que aqueles que obtiveram a nota maior ou igual a 3 foram enfim utilizados no presente trabalho. A Tabela 3 mostra os critérios utilizados para a definição de cada nota, ressaltando que essas notas não tem o propósito de determinar a qualidade do artigo e sim, ser um instrumento de controle que indica o quanto o mesmo é útil para o trabalho. Incidindo o artigo em mais de um critério foi considerado, para efeito classificatório, aquele que correspondia à maior nota. Por exemplo, se um determinado artigo atende os critérios 2 e 8, consequentemente receberá uma nota entre 1 e 2,5 e outra nota entre 4 e 5. Segundo o critério aqui estabelecido tal artigo ficará com a maior nota, ou seja, com a nota referente ao critério 8.

Tabela 3 – Critérios das notas dos artigos

Nota	**Critérios**
1 a 2.5	1 - Apresenta ideias centrais já citadas por outros artigos encontrados. 2 - Apenas cita alguma(s) ferramenta(s), sem nenhum tipo de detalhe. 3 - Apenas cita alguma(s) metodologia(s) ou técnica(s), sem nenhum tipo de detalhe.
3 a 3.5	4 - Apresenta ideias centrais ainda não citadas por outros artigos. 5 - Cita alguma(s) ferramenta(s) com algum tipo de detalhe. 6 - Cita alguma(s) metodologia(s) ou técnica com algum tipo de detalhe.
4 a 5	7 - Cita alguma(s) ferramenta(s) com maior índice de detalhes. 8 - Cita alguma(s) metodologia(s) ou técnica(s) com um maior índice de detalhes.

Fonte: Autor.

A relação dos artigos que foram descartados a partir do segundo critério de seleção ou que foram classificados com notas abaixo de 3, estão disponíveis no Apêndice A deste trabalho.

3 Teste de Software

Para Pressman e Maxim (2016) a atividade de teste proporciona o último elemento que estima a qualidade do software. Pfleeger (2004) afirma que muitos programadores acreditam que os testes são uma oportunidade de demonstrar que seus programas funcionam de maneira correta, porém Sommerville (2011) informa que ao contrário do que se pensa, os testes não podem demonstrar que um software é livre de defeitos. Quando se realiza o processo da atividade de teste, o seu objetivo é a execução de um determinado programa com a intenção de encontrar erros. O cumprimento do objetivo de encontra-los e posteriormente corrigi-los tem como resultado o aumento da confiabilidade e da qualidade do programa. Com isso, pode-se entender que uma boa atividade de teste é aquela que faça o programa errar (MYERS, 2004).

3.1 Estágios do Teste de Software

Segundo Sommerville (2009), o processo de teste é realizado em três estágios que são o teste de unidade, o teste de sistema e o teste de aceitação.

No teste de unidade, cada componente do programa é testado separado das outras unidades do sistema. Uma unidade pode ser entendida como o menor trecho de código de um sistema que pode ser testado, podendo ser uma função ou módulo em programas procedimentais ou métodos ou classes em programas orientados a objetos (MALDONADO et al., 2004). Este teste, também conhecido como teste de módulo, avalia se o componente funciona de forma adequada aos tipos de entrada esperada, a partir do estudo do projeto do componente (PFLEEGER, 2004).

O teste de sistema está relacionado com a busca de erros que resultam das interações não previstas entre os componentes e problemas de interface de componentes. Para sistemas de grande porte, isso pode ser um processo de vários estágios no qual os componentes são integrados para formar subsistemas testados individualmente antes que sejam integrados para formar o sistema final (SOMMERVILLE, 2009). Para outros autores como Pfleeger (2004), o teste de sistema é um estágio que testa o sistema como um todo e que possui um outro estágio chamado teste de integração que testa a interação entre os componentes.

O teste de aceitação é o último estágio do processo de teste, antes que o sistema seja aceito para o uso operacional. O sistema é testado com os dados fornecidos pelo cliente do sistema, em vez de dados simulados de teste. O teste de aceitação pode revelar erros e omissões na definição de requisitos do sistema, pois os dados reais exercitam o sistema de formas diferentes dos dados de teste. Além de revelar problemas de requisitos, em que as funções do sistema não atendem as necessidades do usuário (SOMMERVILLE, 2009).

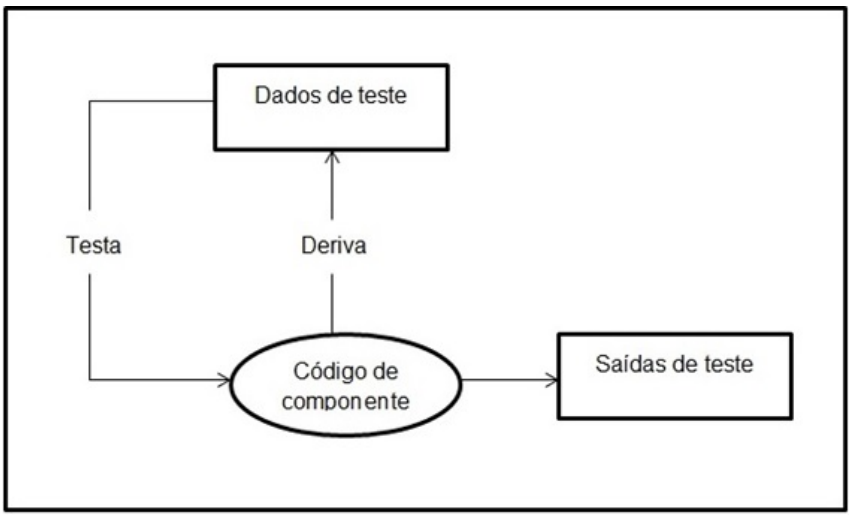
3.2 Técnicas de Teste de Software

O ponto determinante para uma atividade de teste é o projeto e a avaliação da qualidade de um conjunto de testes, pois é uma tarefa inviável utilizar todas as entradas possíveis para que se possa testar um produto. Então, é importante o estudo e elaboração de casos de teste que objetiva encontrar a maioria dos defeitos em menor tempo e esforço. Para isso tem-se definido técnicas e critérios que auxiliem na criação de um subconjunto de valores de entradas que tem grande potencial de encontrar pelo menos a maioria dos defeitos de um determinado programa. Basicamente os critérios de teste de software são estabelecidos por três técnicas, à técnica funcional, estrutural e baseado em erros (DOMINGUES, 2002).

O teste funcional é conhecido como teste da caixa preta pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e da qual é visível apenas o lado externo, ou seja, os dados de entrada e saída do sistema. Nesta técnica são verificadas as funções do sistema sem se preocupar com a implementação (MYERS, 2004). O teste funcional busca encontrar funções incorretas ou ausentes, erros de interface, erros nas estruturas de dados ou o acesso ao banco de dados externos, erros de desempenho e erros de inicialização e término (PRESSMAN, 2000).

O teste estrutural é conhecido como teste da caixa branca pelo fato de tratar o software com uma caixa branca cujo conteúdo é conhecido, logo, oposto ao teste da caixa preta. Com o teste estrutural, os aspectos da implementação são fundamentais na escolha dos casos de teste (MYERS, 2004), como pode ser observado na Figura 1.

Figura 1 – Teste estrutural



Fonte: Adaptado de (SOMMERVILLE, 2009).

Dentro do teste estrutural existe uma estratégia chamada teste de caminho, cujo objetivo é garantir que todos os possíveis caminhos sejam executados pelo menos uma vez, garantindo o conjunto de casos de teste.

A técnica de teste baseado em erros usa informações sobre os tipos de erros que

geralmente aparecem no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência (MALDONADO et al., 2004).

4 Automatização do Teste de Software

A automatização da atividade de teste pode ser considerada como uma parte relevante no processo moderno de desenvolvimento de software (WIKLUND et al., 2014), pois medidas de garantia de qualidade, como testes, gera um grande custo em projetos de software. Segundo Harrold (2000), mais de cinquenta por cento dos custos de um sistema é relativo ao teste. No entanto, de acordo com Ramler e Wolfmaier (2006), o aumento da concorrência que reduz cronogramas e orçamentos, e um rápido processo de desenvolvimento, coloca pressão no uso de mecanismos que melhorem o teste de software. Com isso, a automatização de teste tem sido um meio para reduzir os custos, podendo ser uma maneira eficaz em diminuir as dificuldades envolvidas no desenvolvimento dos testes, excluindo ou minimizando tarefas repetitivas e reduzindo falhas humanas (WIKLUND et al., 2014). Além disso, fornecedores de ferramentas de teste asseguram que a execução automática de scripts de teste possibilita as empresas elevarem a quantidade de testes executados (RAMLER; WOLFMAIER, 2006).

A escolha do critério de teste que será utilizado e a ferramenta de teste que o suporte reflete na qualidade e produtividade da atividade de teste. Sem o uso de uma ferramenta automatizada, a aplicação de um critério torna-se uma atividade inclinada a erros e suficiente apenas aos programas muito simples (MALDONADO et al., 2004). O objetivo de uma ferramenta de teste é proporcionar que o testador aplique o seu teste para a tarefa em questão de uma forma mais eficiente do que se feito manualmente. Se isso falhar, a ferramenta não tem valor (WIKLUND et al., 2014). O efeito dessa prática é que não só apenas os testadores especializados, mas também os desenvolvedores de software em geral possam realizar os testes.

As empresas de desenvolvimento de software utilizam as ferramentas para automatizar o teste por muitas razões, tais como ganhar velocidade, manter repetitivos testes e gestão de mais testes em menos tempo (WIKLUND et al., 2014). De acordo com a pesquisa feita por Rafi et al. (2012), a prática de automatizar o teste de software apresenta muitos outros benefícios identificados na Tabela 4, e também as limitações, como mostra a Tabela 5.

Tabela 4 – Benefícios em praticar o teste automatizado de software

Benefícios	**Descrição**
Melhoria da qualidade do produto	Redução na quantidade de defeitos presente no software.
Cobertura de teste	Elevada cobertura do teste no código fonte através da automatização de teste.
Redução no tempo de teste	Menor o tempo gasto em realizar o teste.
Aumento na confiança	Aumento da confiança na qualidade do sistema.
Reutilização de testes	Quando o software passa por manutenção, os testes utilizados na primeira vez podem ser reutilizados.
Menor esforço humano	A automatização reduz o esforço humano que pode ser usado para outras atividades.
Redução nos custos	Com um alto grau de automatização, os custos para atividade de teste são reduzidos.
Aumento na detecção de falhas	Capacidade de encontrar uma grande parcela de defeitos.

Fonte: Adaptado de (RAFI et al., 2012, **tradução nossa**).

Tabela 5 – Limitações em praticar o teste automatizado de software

Limitações	**Descrição**
Automatização não pode substituir o teste manual	Nem todas as atividades de teste podem ser facilmente automatizados, principalmente aqueles que exigem amplo conhecimento de um domínio.
Dificuldade na manutenção de teste automatizados	Mudança na tecnologia e evolução dos produtos de software levam a dificuldade em manter testes automatizados.
Processo de automatização de teste requer tempo para amadurecer	A criação da infraestrutura da automatização de teste exige tempo.
A falta de pessoas qualificadas	Para automatizar o teste muitas habilidades são necessárias, como por exemplo, o conhecimento das ferramentas de teste, desenvolvimento de software em geral, entre outros.
Estratégia imprópria de automatização de teste	Não é simples uma elaboração de teste adequado.

Fonte: Adaptado de (RAFI et al., 2012, **tradução nossa**).

A constatação de que o teste é importante resultou no surgimento de diversas ferramentas que auxiliam na atividade de testar o software (GAFFNEY; TREFFTZ; JORGENSEN, 2004). Dentre essas ferramentas não há nenhuma que sozinha é favorável a todos os sistemas possíveis, levando a complicada decisão de escolher uma ferramenta específica para um projeto (YANG; LI; WEISS, 2006).

O levantamento bibliográfico realizado neste trabalho descreve sobre o teste unitário automatizado e suas técnicas de apoio, além disso, apresenta todas as ferramentas encontradas nas fontes de pesquisa estabelecida no Capítulo 2.

5 Automatização do Teste Unitário

O teste de unidade não apresenta a necessidade de um testador profissional ser o responsável pela sua execução. De acordo com Wang e Offutt (2009), o teste de unidade tem o propósito de que cada porção de código do programa detêm seus próprios testes e que esses testes sejam elaborados, de preferência, por aquele que escreveu a parte do código que está sob avaliação, no caso o desenvolvedor. Porém, alguns desenvolvedores não realizam muito esse tipo de teste por não serem qualificados para a criação de bons testes. Uma das causas é o fato do teste de software ser uma prática que geralmente não é ofertada com ênfase nas faculdades, principalmente nas práticas de laboratório (WAHID; ALMALAISE, 2011). Além da falta de domínio, há também outros motivos, como o tempo gasto e a manutenção dos testes. Segundo Wang e Offutt (2009), os desenvolvedores não consideram que o tempo e a dedicação vão valer a pena, e com a necessidade da preservação dos testes de unidade, a sua manutenção geralmente não está contabilizada no orçamento do projeto. Porém mesmo com essas dificuldades, Daka e Fraser (2014) afirmam que o teste de unidade é um fator importante no desenvolvimento de software.

Com isso as ferramentas de teste de unidade automatizado tem uma grande importância na redução das dificuldades em se fazer o processo de avaliação em uma parte do software. Tais ferramentas contribuem na redução do tempo e o esforço necessário para se projetar, executar e manter os testes unitários. Além de ser desnecessário o desenvolvedor conhecer todo o procedimento da atividade, inclusive implementação de testes de alta qualidade (WANG; OFFUTT, 2009).

Para automatizar os testes de unidade se tem como base o uso de casos de teste. Segundo Lages (2010), caso de teste é a representação de uma instância de teste para aquela funcionalidade que está sendo avaliada, de acordo com as entradas e o resultado esperado é percorrido um determinado caminho no algoritmo para que se avalie aquela situação. Logo, existem duas categorias de ferramentas, uma que gera conjuntos de casos de teste e a outra que os executam e verificam o resultado, ambos de modo automático. Nas próximas seções são apresentadas essas categorias e as suas respectivas ferramentas.

5.1 *Frameworks* de Teste de Unidade

Uma atividade comum e praticada na década de 90 era a criação de uma função que continha simulações de teste para cada classe ou módulo do sistema. O transtorno dessa prática era a desorganização, pois o código adicional era agregado ao próprio sistema deixando o algoritmo menos legível. Então surgiram os arcabouços ou *frameworks* de teste, que tinha o intuito de auxiliar e padronizar as escritas de teste automatizados. Além de simplificar a separação do código de teste com o código do programa original (BERNARDO; KON, 2008).

Estes *frameworks* de teste de unidade são conhecidos como parte da família de arca-

bouços xUnit. Criado por Kent Beck a família xUnit teve como a sua primeira implementação o SUnit, framework para a linguagem de programação SmalTalk. A versão original consistia na formação de pequenos testes, um para cada classe, assim, caso algum erro ocorresse durante a implementação, ele poderia ser rapidamente detectado e corrigido. Devido a essa característica de implementação dos testes por classes, a alteração em uma classe resulta na alteração dos testes apenas na respectiva classe. O teste realizado por xUnit baseia-se na comparação do resultado obtido ao executar a função com o resultado esperado. Há implementações para outras linguagens como JUnit para Java, JSUnit para JavaScript, csUnit para .NET entre outros (BERNARDO; KON, 2008).

Segundo Acharya (2015), o JUnit é o *framework* de testes unitário mais popular para a linguagem Java. Além disso, segundo Wahid e Almalaise (2011), o JUnit é uma estrutura orientada a objeto madura, amplamente utilizada e compreendida, além de ser considerada API padrão para teste de unidade em Java. O JUnit possui métodos que auxiliam na comparação dos resultados obtidos com os resultados esperados, tais como assertEquals que analisa se o conteúdo do objeto é igual, assertSame que compara se duas referências se referem ao mesmo objeto, assertNull que verifica se uma determinada referencia é nula (BERNARDO; KON, 2008). Como exemplo de um dos métodos, a Figura 2 apresenta uma função que executa a soma de dois números, chamada ExecutaCalculo. Para testa-la, é utilizado o método assertEquals que analisa se a resposta esperada é igual a resposta exibida pela função, como mostra a Figura 3.

Figura 2 – Função ExecutaCalculo

```
package Fonte;

public class Calculo {
    public static float ExecutaCalculo(float Valor1, float Valor2) {
        float Soma = Valor1 + Valor2;
        return Soma;
    }

}
```

Fonte: Autor.

Figura 3 – Função teste ExecutaCalculo

```
package Fonte;

import junit.framework.TestCase;

public class CalculoTest extends TestCase {

    public void testExecutaCalculo() {
        float PassaValor1 = 10;
        float PassaValor2 = 5;
        float RetornoEsperado = 15;

        float RetornoFeito = Calculo.ExecutaCalculo(PassaValor1, PassaValor2);
        assertEquals(RetornoEsperado,RetornoFeito);

    }
```

Fonte: Autor.

Um aspecto relevante que facilita ainda mais a execução do conjunto de testes e a leitura dos resultados obtidos, além de ser comum em outros arcabouços, é a integração da ferramenta nas IDES de desenvolvimento, como é o caso do JUnit integrado ao Eclipse. A Figura 4 mostra o relatório com os resultados dos testes gerados pelo *plugin* JUnit no Eclipse. O relatório contém: uma barra verde quando todos os testes passam com sucesso ou vermelha quando não passam, indicação do número de falhas ou erros, e a exposição do caso de teste que aprovou a função ou encontrou a falha (BERNARDO; KON, 2008). O JUnit não é o único que executa os testes unitários para a linguagem Java, o *framework* TestNG também exerce essa função. No entanto, o mesmo não é descrita no trabalho por ser de artigos que foram descartados nos critérios de seleção.

Figura 4 – Relatório JUnit

Fonte: Adaptado de (BERNARDO; KON, 2008).

5.2 Geradoras de Casos de Teste

Os *frameworks* de teste unitário como a família xUnit não oferecem suporte para a elaboração dos casos de teste, pois a ideia é que os próprios desenvolvedores os projetem (ZHU, 2015). Porém, ao utilizar esses *frameworks*, o desenvolvedor ou o testador pode não precisar escrever manualmente todos os casos de teste (FRASER et al., 2013), visto que, é um processo cansativo e que demanda tempo para serem elaborados. Para tal, existem ferramentas que realizam geração automática de um conjunto de dados de entradas e saídas esperadas para teste, que podem ser trabalhadas em uma unidade do programa a ser testado.

Os casos de testes automatizados proporcionam um menor esforço para o desenvolvedor, porém, não se pode depender apenas desse artificio porque, de acordo com (LEITNER et al., 2007), os desenvolvedores são melhores na criação de dados de testes mais complexos,

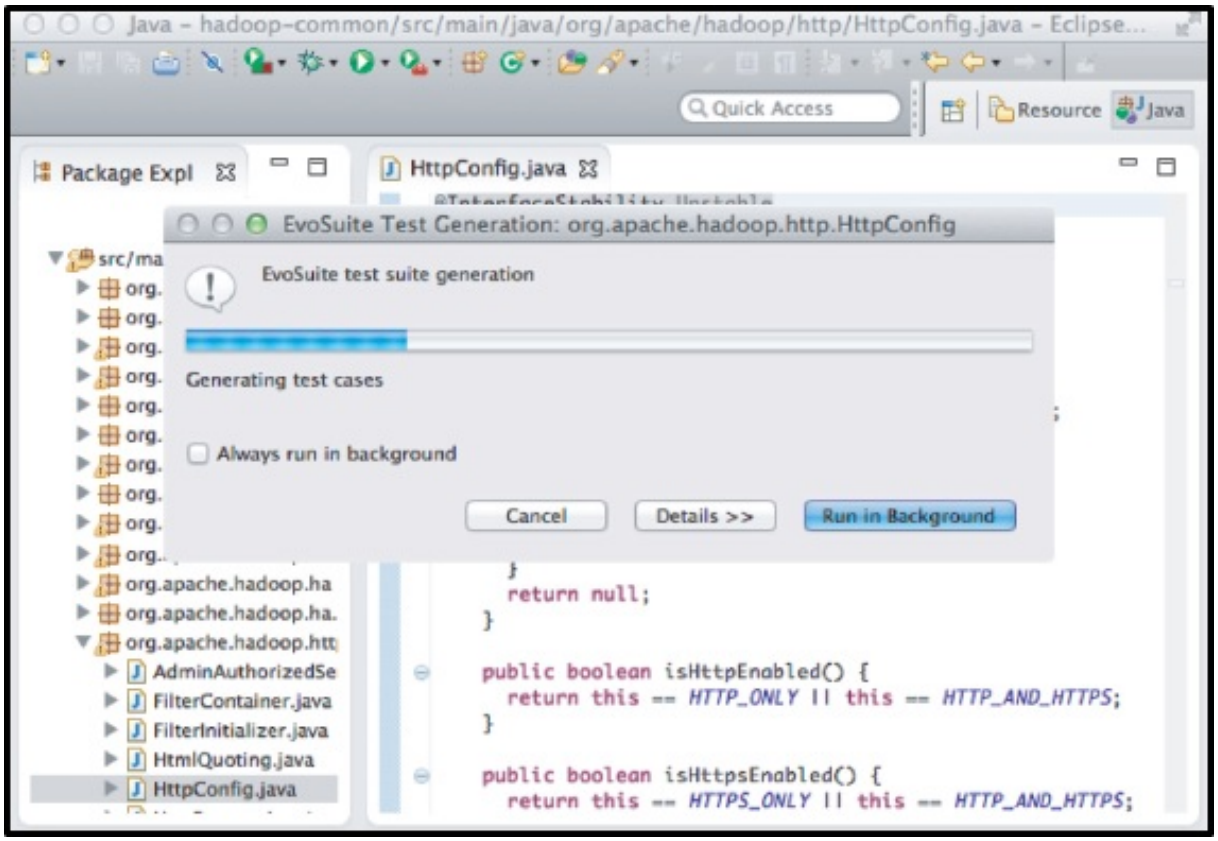
sendo importante na descoberta de casos de erros incomuns. No entanto, a automatização tem a capacidade de produzir uma maior cobertura de código por causa da grande quantidade de casos de teste gerado. Com isso, Leitner et al. (2007) ainda afirma que os testes automatizados são bons em amplitude, mas muito menos em profundidade.

As ferramentas de geração de casos de teste automatizado são eficientes no que elas foram projetadas para fazer, tendendo a produzir uma cobertura de instrução superior aos casos de teste manual e podendo apoiar os desenvolvedores no melhoramento da cobertura das provas escritas por eles. A utilização dessas ferramentas pode resultar na criação de um ponto de partida diferente para o teste em relação ao manual tradicional, influenciando na forma de como o conjunto de testes será desenvolvido (FRASER et al., 2013). Vale ressaltar que o simples fato de fornecer uma ferramenta de geração de teste para um desenvolvedor não vai proporcionar um melhor resultado, pois é necessário educação e experiência sobre quando e como usar essas ferramentas além de também saber determinar quais são as melhores práticas de teste (ROJAS; FRASER; ARCURI, 2015). Na próxima seção é descrita a ferramenta Evosuite para geração de casos de teste em classes da linguagem Java, porém, outras ferramentas que exercem essa função como JTExpert, EvoSuite-Mosa, T3, CT, GRT e Randoop não são descritas por serem de artigos que não possuem informações relevantes sobre as mesmas.

5.2.1 Ferramenta Evosuite

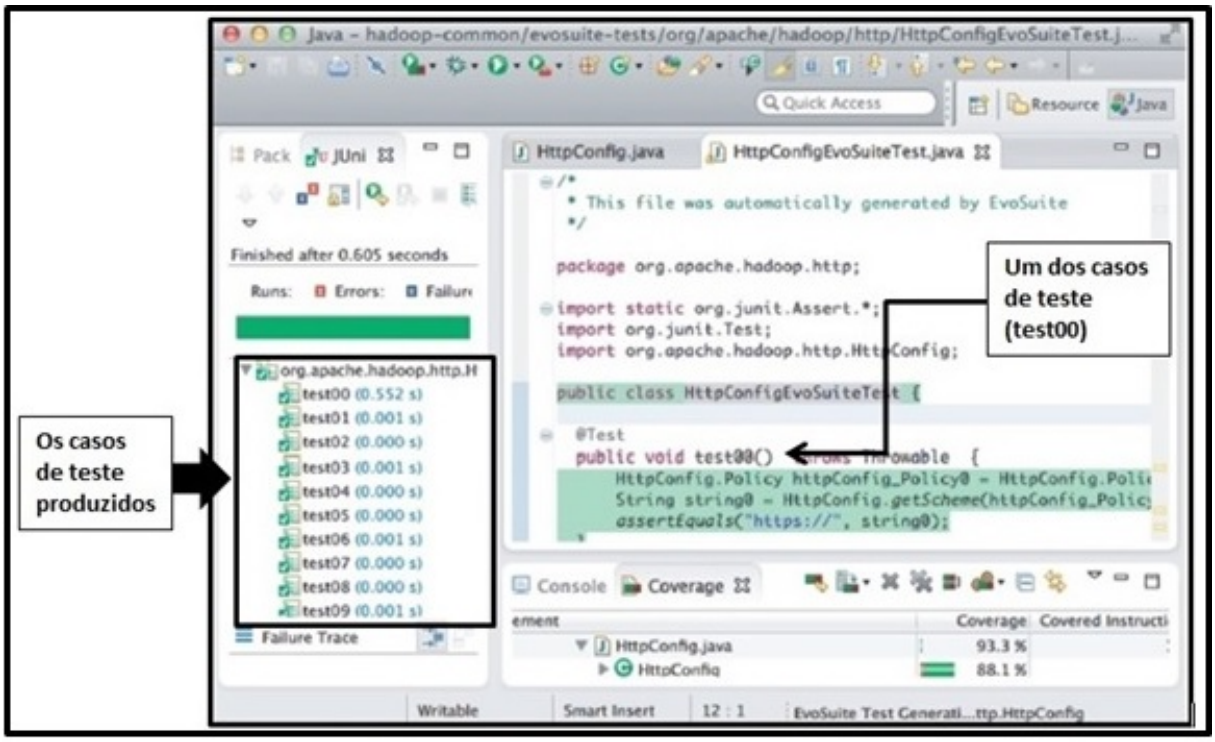
Evosuite é uma ferramenta que gera automaticamente conjuntos de casos de teste para as classes da linguagem Java no formato JUnit, visando dentre outros critérios de cobertura, a cobertura de ramos, em que alcança bons níveis (FRASER; ARCURI, 2014). Essa ferramenta que ficou em segundo lugar na competição de ferramentas SBST (*Search-Based Software Testing*) 2015, não exige código fonte para se trabalhar, pois ela atua em nível de *bytecode*. Ao utiliza-la, tanto no seu *plugin* do Eclipse quanto na própria ferramenta, o usuário apenas seleciona as classes que se quer testar e então os testes são gerados. A Figura 5 mostra a interface da ferramenta no momento da elaboração dos casos de teste, e a Figura 6 exibe o resultado, que são os casos de teste produzidos. Evosuite tem como base o algoritmo genético em que utiliza operadores inspirados pela evolução natural, de tal modo que as melhores soluções com o respeito ao alvo são produzidas (FRASER; ARCURI, 2015). Com os candidatos a casos de teste gerados, a ferramenta seleciona aqueles mais eficazes utilizando a análise de mutação (PRASETYA, 2015). A Evosuite possui seu próprio gerente de segurança, para que se possam restringir execuções que tragam efeitos colaterais indesejados (FRASER; ARCURI, 2014).

Figura 5 – Geração dos testes



Fonte: (FRASER; ARCURI, 2014).

Figura 6 – Testes gerados



Fonte: Adaptado de (FRASER; ARCURI, 2014).

6 Técnicas e Ferramentas de Apoio ao Teste Unitário de Software

Os desenvolvedores de software precisam de orientação para fazer e escrever bons testes (DAKA; FRASER, 2014), então, neste capítulo são apresentadas as técnicas de apoio ao teste unitário de software, que são: análise de mutação, teste cobertura, complexidade ciclomática e *mocks objects*. Além de suas respectivas ferramentas.

6.1 Análise de Mutação

Teste de mutação ou análise de mutação é uma técnica na qual todas as possíveis versões defeituosas do programa original são gerados para avaliar os casos de teste criados pelo desenvolvedor, resultando o *feedback* sobre a qualidade do conjunto de testes que está sendo trabalhado (AHMED; ZAHOOR; YOUNAS, 2010). Este *feedback* pode ser usado como um meio de auxílio no ajuste dos critérios de teste, bem como orientação para conduzir os testes para as partes ainda não cobertas no programa. Segundo Smith e Willians (2007), teste de mutação é definido como uma “[...] metodologia de teste em que duas ou mais mutações de um programa são executados contra o mesmo conjunto de testes para avaliar a capacidade do suíte de teste em detectar essas alterações”. Smith e Williams (2009) afirmam ainda que é possível se utilizar da análise de mutação como um recurso eficiente para o aprimoramento do conjunto de teste em encontrar falhas de um programa.

Os operadores de mutação exercem pequenas alterações no programa produzindo variações no programa original, sendo chamados de mutantes. Vários operadores de mutação podem ser utilizados de acordo com o tipo de mutante que se deseja. O presente trabalho apresenta alguns dos operadores de mutação Java divididos em dois grupos, os operadores tradicionais exibidos na Tabela 6 e os operadores em nível de classe exibidos na Tabela 7 (RIBEIRO; ARAúJO, 2012).

Tabela 6 – Operadores tradicionais

Oeradores	**Descrição**
AORB	É um operador que realiza substituição de operadores,aritméticos, esse procedimento é feito em operadores + (adição), - (subtração), / (divisão), * (multiplicação).
AORS	É um operador que realiza substituição de operadores aritméticos, esse procedimento é feito em operadores ++ ou –. Grande parte desses operadores é utilizada dentro de loops onde existem variáveis de controle para determinar a quantidade de vezes que o fluxo de processamento está passando pelo loop.
AOIU	É um operador que realiza inserção de operadores aritméticos, esse operador influencia nas variáveis numéricas ao inserir o sinal de subtração nas mesmas.
AOIS	É um operador que realiza inserção de operadores aritméticos, esse operador insere operadores ++ (incremento de 1) ou – (decremento de 1) nas variáveis numéricos de uma classe.
ROR	É um operador que realiza substituição de operadores relacionais, esse operador realiza a troca de um operador relacional por outro.
COD	É um operador que remove os operadores de negação em uma condição.
COI	É um operador que funciona de uma maneira inversa ao COD, em que insere o operador de negação.
LOI	É um operador de inserção lógico, em que insere os operadores $\sim$ (incremento e inversão de sinal) e – (subtração). Esse operador de mutação tem a finalidade de encontrar falhas que verificam métodos que executam operações aritméticas.
ASRS	É um operador que realiza alterações nas instruções do tipo +=, -=, =, /=, %=, &=, |=, ^=, «=, »=, e »>=.

Fonte: (RIBEIRO; ARAúJO, 2012).

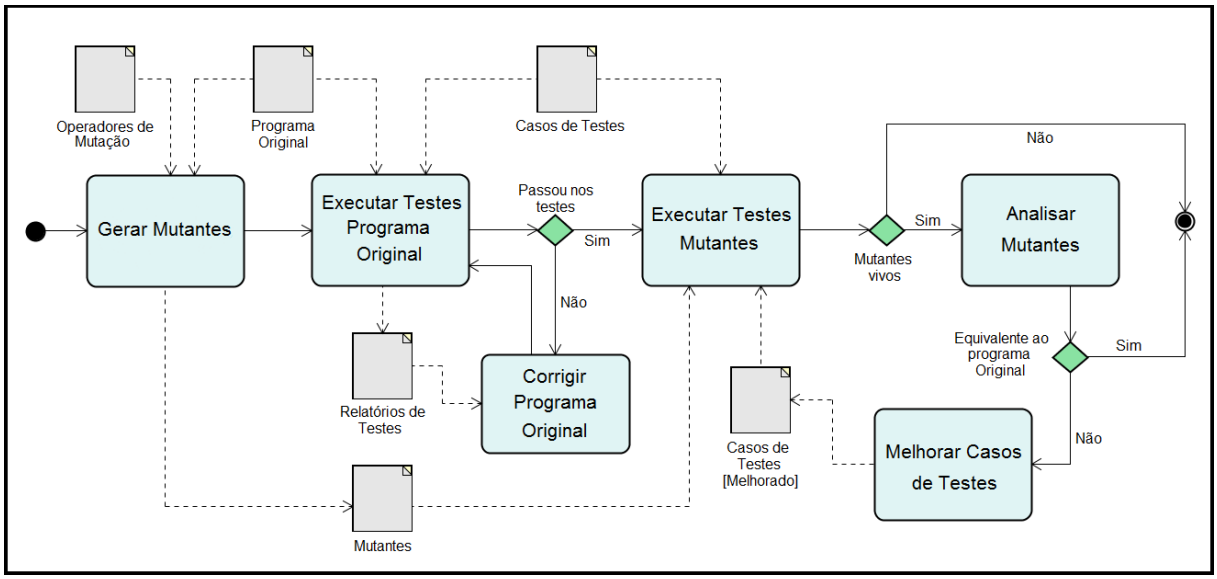
Tabela 7 – Operadores em nível de classe

Limitações	**Descrição**
IHD	Esse operador apaga um método ou atributo sobrescrito em uma subclasse.
ISI	Esse operador insere a palavra chave super em métodos que são sobrescritos.
ISD	Esse operador é o inverso do operador de mutação ISI. Ele apaga a palavra chave super de onde ela estiver sendo utilizada.
PRV	Esse operador altera as variáveis de uma classe.
JTI	Esse operador adiciona a palavra chave this aos atributos e métodos.
JTD	Esse operador é o inverso do operador de mutação JTI. Ele remove a palavra this.
JSI	Esse operador de mutação insere a palavra chave static nos atributos e métodos de uma classe.
JSD	Esse operador é o oposto do operador JSI. Ao invés de inserir a palavra chave static, ele a remove.
JID	Esse operador exclui a inicialização de variáveis.
EAM	Esse operador altera os métodos de acesso por outros métodos compatíveis.

Fonte: (RIBEIRO; ARAúJO, 2012).

O processo que envolve os testes de mutação é dividido em quatro fases, como mostra a Figura 7. Na primeira fase é a geração de mutantes, que são pequenas modificações elaboradas no programa original através de operadores de mutação. A segunda fase do processo é a execução dos testes sobre o programa original. Caso algum teste assinale um erro no programa é fundamental que se verifique manualmente o código fonte para a correção do defeito e depois rodar os testes novamente. Essa fase se repete até que o programa original seja aprovado nos testes. A fase três é a execução dos testes sobre os mutantes em que cada um dos mutantes gerados passa pelos mesmos testes que o programa original. Os mutantes que não foram aprovados nos testes são considerados mutantes mortos, isso significa que aquele conjunto de testes é capaz de encontrar os tipos de defeitos inseridos pelos operadores de mutação. A última fase consiste na análise dos mutantes que foram aprovados, chamados de mutantes vivos. É exigida uma intensa intervenção humana para que se descubra se os mutantes vivos são equivalentes ao programa original ou se os casos de testes não foram bons para identificar as alterações inseridas (RIBEIRO; ARAúJO, 2012).

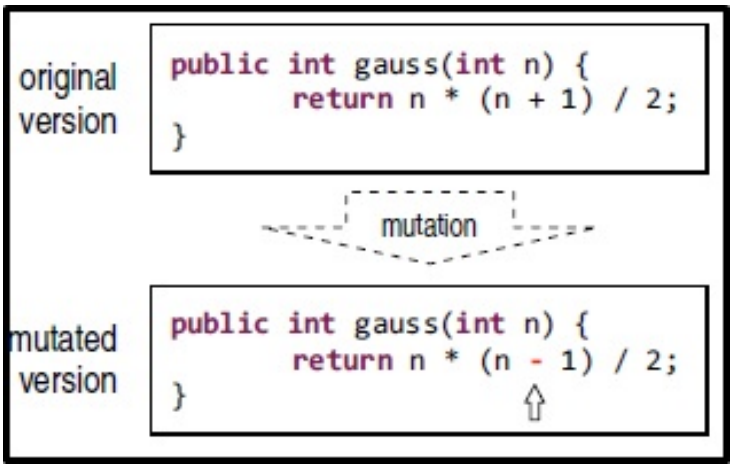
Figura 7 – Processo de criação do teste de mutação



Fonte: (RIBEIRO; ARAúJO, 2012).

O exemplo que segue tem o intuito de esclarecer ainda mais sobre o teste de mutação. Na Figura 8 é mostrada a codificação da função de Gauss e o seu mutante. Essa versão mutada tem como modificação a troca do operador aritmético “+” pelo operador “-“. Um caso de teste considerado eficiente é aquele que detecta o erro causado pela modificação no código. Em uma situação que o caso de teste possui a entrada N=2 e avalia a saída esperada com R=3, falharia quando se executado com a versão mutada. O mutante é “morto” gerando uma pontuação de mutação de 100 por cento. É gerada uma discordância quando o caso de teste é N=0 e R=0, não sendo capaz de matar o mutante e deste modo resultaria uma pontuação de mutação igual à zero por cento (RAMLER; KASPAR, 2012).

Figura 8 – Mutante do código fonte original



Fonte: (RAMLER; KASPAR, 2012).

Analise de mutação por muito tempo foi classificado como um método demorado e

caro, isso se deve ao fato do código fonte estar sempre sendo modificado e depois recompilado. Porém nos dias atuais esse conceito vem mudando, isso porque há um aumento de ferramentas avançadas dessa natureza e que se aliam ao poder de processamento dos computadores modernos, resultando no impulso da adesão do processo de mutação em uma esfera global. Existem ferramentas que cobrem um grande número de linguagens de programação sendo que alguns deles são de código fonte aberto, consequentemente cada vez mais resultados positivos são relatados a partir da aplicação de análise de mutação no desenvolvimento de softwares (RAMLER; KASPAR, 2012). Na próxima seção são abordadas as ferramentas de teste de mutação encontradas no levantamento bibliográfico.

6.1.1 Ferramentas de Mutação

MuJava é um sistema de mutação de classe automatizado que gera no modo automático mutantes para as classes da linguagem Java, e avalia conjuntos de casos de teste através do cálculo do número de mutantes mortos. MuJava pode testar classes individuais e pacotes de múltiplas classes, utilizando 24 operadores de mutação para a criação de mutantes orientadas a objetos (WANG; OFFUTT, 2009).

O MuClipse é um *plugin* do Eclipse que funciona como uma ponte entre Mujava e o Eclipse. O seu propósito é executar teste de mutação em programas implementados com a linguagem Java. Para realizar essa atividade, o MuClipse determina que o .class da aplicação não fique no mesmo diretório do .class dos testes, então é preciso configurar de onde os .class devem ser gerados. De acordo com o exemplo da Figura 9, a classe do projeto chamado Aluno é criada dentro de um pacote no *Source Folder* src e seus casos de teste são criados dentro de um pacote no *Source Folder* testset (RIBEIRO; ARAúJO, 2008).

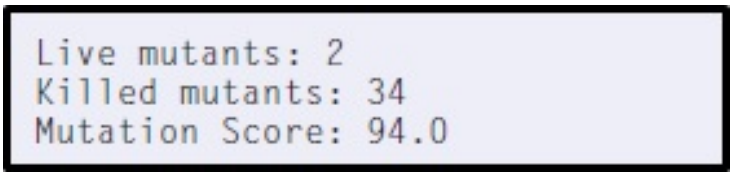
Figura 9 – Estrutura do projeto

Fonte: (RIBEIRO; ARAúJO, 2008).

Com o projeto criado e as configurações necessárias feitas, é possível gerar os mutantes. O MuClipse oferece entre outras novas funções no Eclipse, o MuClipse: *Mutants* e o MuClipse:*Tests*. A primeira proporciona a geração de mutantes, podendo configura-los e deter-

minar quais operadores de mutação serão utilizados. E o segundo, executa os casos de teste com os mutantes gerados, apresentando no final da execução, dentre outras informações, o *Console* com o resumo do procedimento, como a Figura 10 mostra. O *Live mutants* informa o número de mutantes vivos, *Killer mutants* o número de mutantes mortos e o *Mutation score* é a representação do quanto eficaz são os casos de teste (RIBEIRO; ARAúJO, 2008).

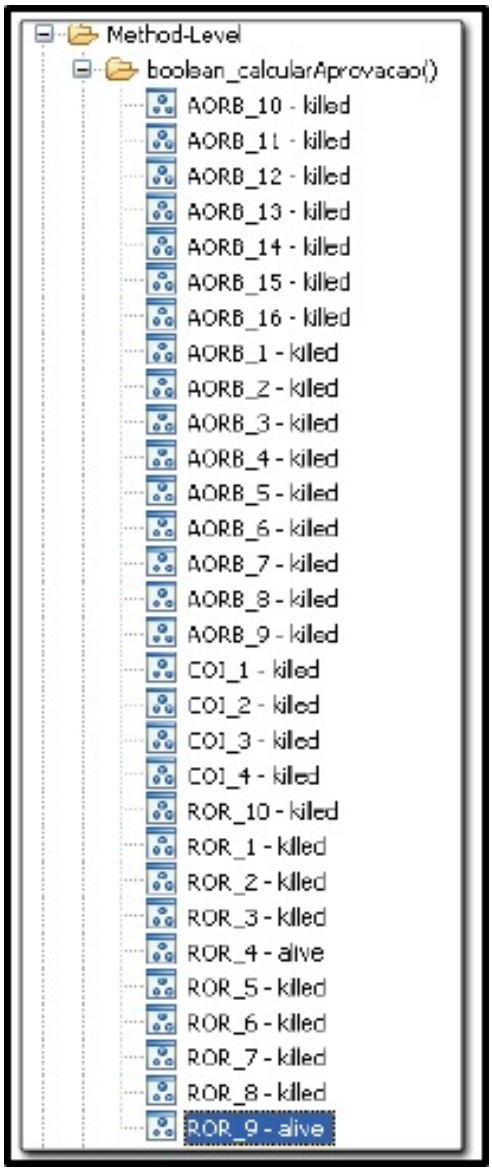
Figura 10 – Resumo execução dos mutantes



Fonte: (RIBEIRO; ARAúJO, 2008).

A ferramenta ainda oferece outra funcionalidade que auxilia na análise dos mutantes vivos. Isso é muito importante para diagnosticar se os mutantes são equivalentes ao programa original ou se é necessário melhorar os casos de teste. A função *View Mutants and Results* exibe quais foram os mutantes vivos e mortos, como mostra a Figura 11. Bastando um duplo clique naquele mutante vivo é exibida uma comparação entre ele e o programa original, sendo possível realizar uma avalição do mesmo (RIBEIRO; ARAúJO, 2008).

Figura 11 – Mutantes vivos e mortos



Fonte: (RIBEIRO; ARAúJO, 2008).

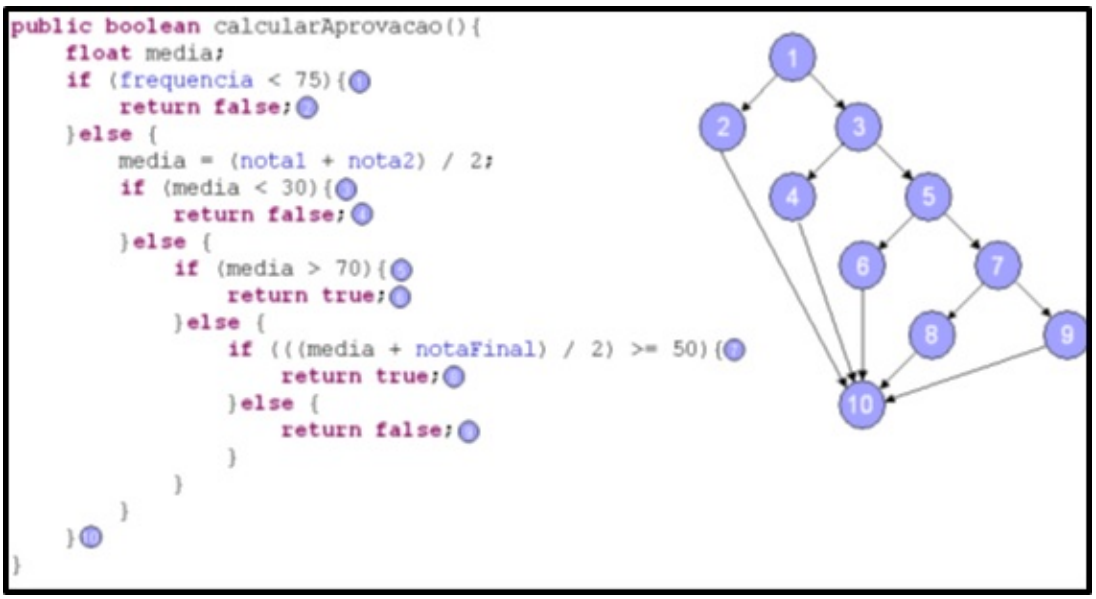
O restante da abordagem do MuClipse está disponível no Anexo A. O conteúdo é relativo às questões técnicas referentes os procedimentos de configuração das funcionalidades da ferramenta.

6.2 Complexidade Ciclomática

A complexidade ciclomática é uma métrica criada por McCabe no ano de 1976 que fornece a medida quantitativa da complexidade lógica do programa. Essa métrica permite determinar a quantidade de caminhos existentes de um algoritmo através do seu número de condições, e assim identificar a complexidade do sistema e por consequência determinar a quantidade mínima necessária de casos de teste para cobrir todo o algoritmo. A contagem dos caminhos possíveis é uma das formas para se calcular a complexidade ciclomática. O exemplo

da Figura 12 mostra o algoritmo que calcula a aprovação de um aluno, os caminhos possíveis são: 1, 2 e 10; 1, 3, 4 e 10; 1, 3, 5, 6 e 10; 1, 3, 5, 7, 8 e 10; 1, 3, 5, 7, 9 e 10. Portanto, a complexidade ciclomática é igual a 5 (ABREU; MOTA; ARAúJO, 2010).

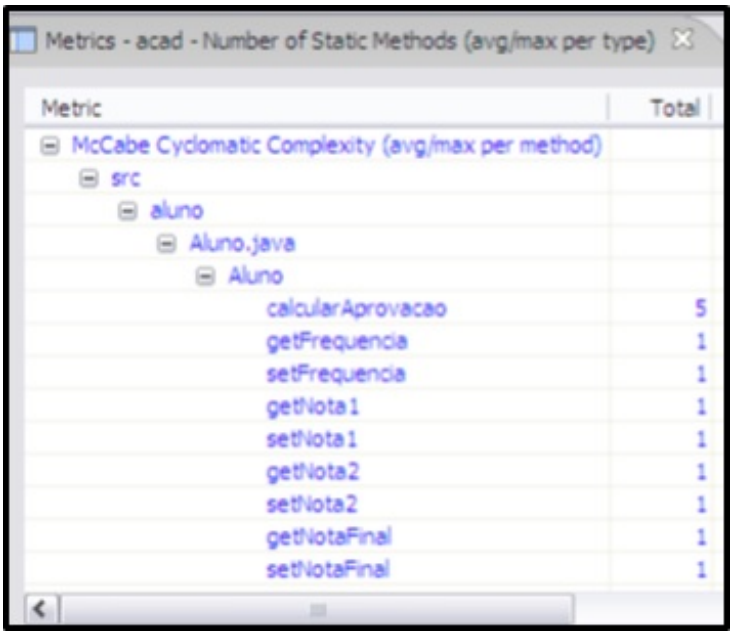
Figura 12 – Quantidade de possíveis caminhos do algoritmo calcularAprovacao



Fonte: (ABREU; MOTA; ARAúJO, 2010).

Para algoritmos que são grandes e complexos medir a complexidade ciclomática de cada função pode se tornar uma atividade cansativa e demorada. Por isso, há ferramentas que calculam automaticamente independentemente da complexidade e tamanho do programa. Como é o caso do Plugin Metrics, um *plugin* gratuito para a IDE Eclipse que calcula a complexidade ciclomática em sistemas desenvolvidos na linguagem Java. A Figura 13 mostra um exemplo dessa ferramenta que executa o cálculo para as funções de uma classe chamada Aluno (ABREU; MOTA; ARAÚJO, 2010).

Figura 13 – Cálculo da complexidade ciclomática com Plugin Metrics



Fonte: (ABREU; MOTA; ARAúJO, 2010).

6.3 Teste de Cobertura

Frameworks xUnit não oferecem nenhum tipo de informação que sugere o quanto de cobertura aquele caso de teste realizou dentro do código em teste. O desenvolvedor quando utiliza esse tipo de ferramenta fica a par apenas se aquela unidade passou ou não nos testes, sem saber se todas as linhas daquele código ou quais delas foram realmente testadas (GAFFNEY; TREFFTZ; JORGENSEN, 2004). Sem esse retorno pode haver o risco de deixar partes importantes daquela unidade do programa fora da avaliação.

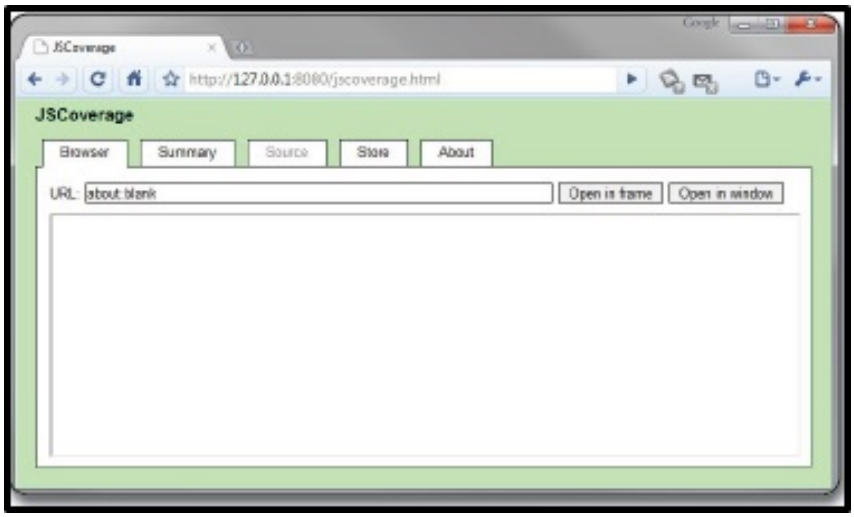
O teste de cobertura avalia a competência do teste de caixa branca, utilizando diferentes critérios de medição como cobertura de declaração (cobertura de linha), cobertura de decisão (cobertura de ramo), cobertura de função / método, cobertura de classe e etc (YANG; LI; WEISS, 2006). Com base nesses critérios, as ferramentas de cobertura informam a quantidade de vezes que um determinado conjunto de casos teste executou cada linha de código, exibindo as linhas que não foram executadas com uma coloração diferenciada, auxiliando assim o testador a melhorar o conjunto de teste para que possa atender as partes ainda não cobertas do código. Essas informações resultantes do teste de cobertura são certamente necessárias para que cada fragmento do código seja avaliado pelo menos uma vez. Porém, os testadores devem se orientar de que a cobertura de código completo é uma condição útil, mas não suficiente para afirmar que um pedaço de código foi completamente testado (GAFFNEY; TREFFTZ; JORGENSEN, 2004). Alcançar a cobertura de trajeto completo, que envolve todas as alternativas de execução possíveis através dos casos de teste, pode ser extremamente difícil (GAFFNEY; TREFFTZ; JORGENSEN, 2004 apud DEZINGER, 2002), principalmente quando se utiliza os laços "*if*" e "*while*" em que proporcionam um número de diferentes caminhos a serem executados, fazendo com que as ferramentas não garantam que todos os

caminhos possíveis foram atravessados (GAFFNEY; TREFFTZ; JORGENSEN, 2004). No entanto, a técnica auxilia em alcançar um nível suficiente de cobertura (JORGENSEN, 2002). Na próxima seção é apresentada a ferramenta JsCoverage, que realiza teste de cobertura para programas em linguagem JavaScript. Ressaltando que há outras ferramentas como Agitar, Bullseye, Clover, Cobertura, CodeTest, Dynamic, EMMA, eXVantage, gcov, Insure++, JCover, JTest, PurifyPlus, Semantic Designs (SD) e TCAT que exercem essa função até mesmo para outras linguagens, porém, não são descritas por serem de artigos descartados nos critérios de seleção.

6.3.1 Ferramenta Jscoverage

JsCoverage é uma ferramenta que realiza testes de cobertura para programas desenvolvidos na linguagem JavaScript. O seu funcionamento é através da instrumentação do código JavaScript usado pela página HTML, com o acréscimo das funções específicas do JsCoverage que visam retornar o número de execuções de cada linha de código, obtendo assim uma análise de cobertura completa. A Figura 14 mostra a interface do JsCoverage (TOLEDO et al., 2010).

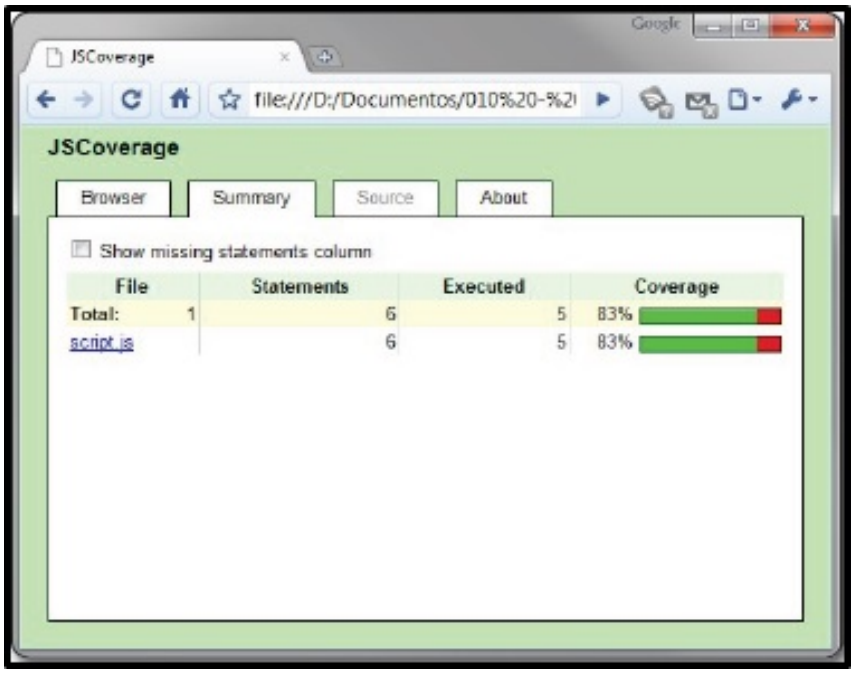
Figura 14 – Interface JsCoverage



Fonte: (TOLEDO et al., 2010).

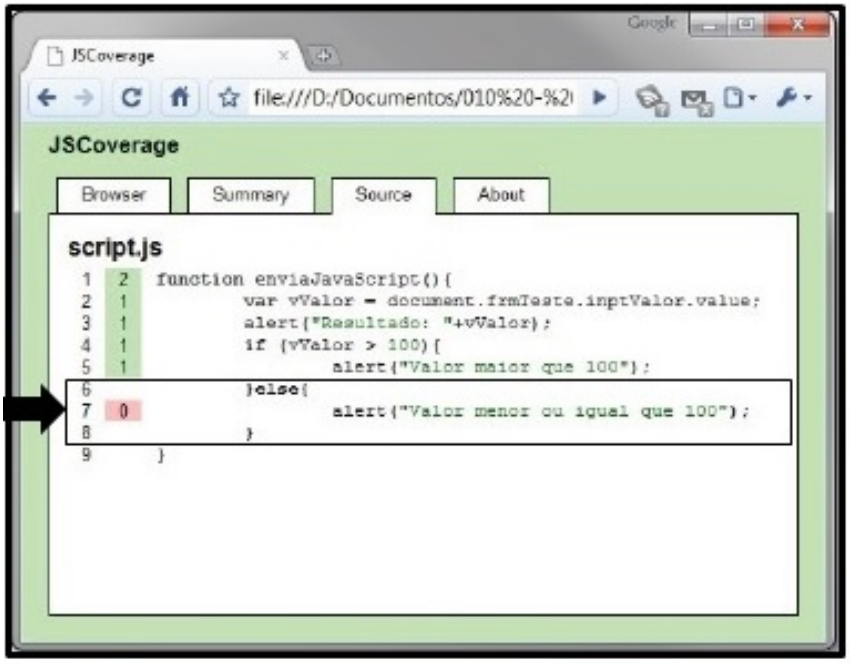
Informando o endereço do arquivo HTML que será testado na opção URL e acionando a função *Open in frame* é aberto a página HTML no *frame* da interface do JsCoverage. No exemplo a seguir, o programa solicita a informação de um valor numérico, digitando o número 200 e clicando no botão enviar o JavaScript é acionado e retorna que o valor é maior que 100. Após a execução e acessar a aba *Summary*, é possível visualizar todos os arquivos JavaScript e as coberturas de cada um como mostra a Figura 15. Ao clicar no arquivo, o JsCoverage exibe todas as linhas cobertas e não cobertas pelo teste. No exemplo da Figura 16, o caso de teste não cobriu a parte do código (indicado pela seta) que analisa se o valor é menor ou igual a 100 (TOLEDO et al., 2010).

Figura 15 – Resultado de cobertura



Fonte: (TOLEDO et al., 2010).

Figura 16 – Cobertura da função enviaJavaScript

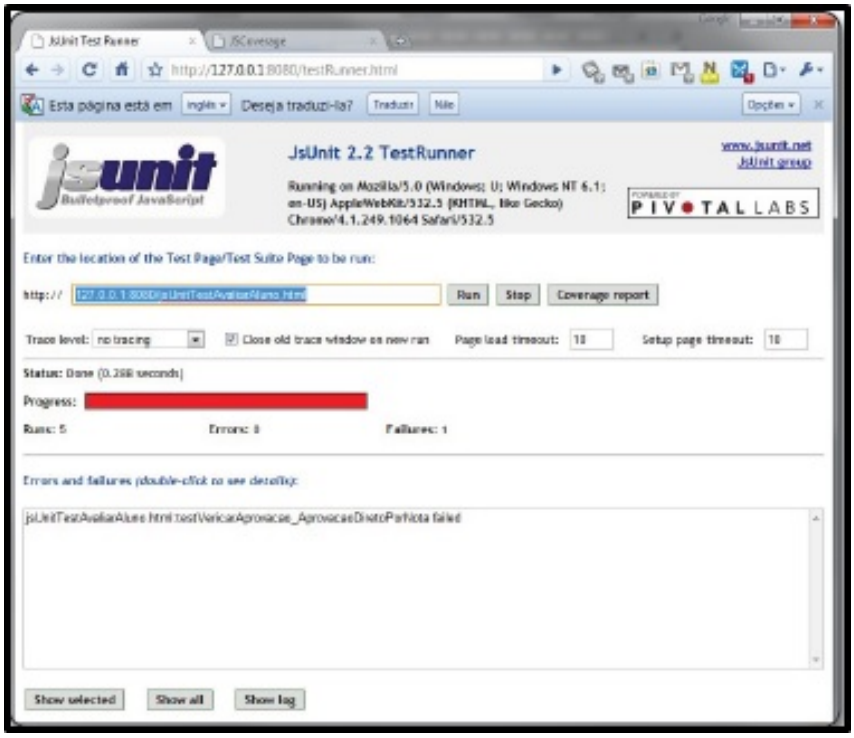


Fonte: Adaptado de (TOLEDO et al., 2010).

A família xUnit possui variantes que são específicas para diferentes linguagens, no caso do JavaScript tem-se o *framework* JsUnit que possui todas as conformidades e padrões do xUnit. Com isso é possível executar os casos de teste com JsUnit e em seguida analisar a cobertura dos testes com o JsCoverage, integrando as ferramentas. A interface dessa in-

tegração, como mostra a Figura 17, possui dentre outras opções, o carregamento do arquivo HTML com os testes unitários, o botão para executa-los e o botão que chama a ferramenta jsCoverage (TOLEDO et al., 2010).

Figura 17 – Interface JsUnit



Fonte: (TOLEDO et al., 2010).

O restante da abordagem do JsCoverage está disponível no Anexo B. O conteúdo deste anexo é relativo às questões técnicas, que são os procedimentos de configuração, de instrumentação do código, além da integração com o JsUnit.

6.4 *Mocks Objects*

O teste de unidade é uma técnica popular que direciona cada caso de teste para uma única parte do programa, porém na prática é difícil testa-lo de uma forma isolada (TILLMANN; SCHULTE, 2006). Porque dentro dos sistemas desenvolvidos em linguagens orientadas a objetos, diferentes classes se comunicam e seus métodos são dependentes das execuções realizadas por métodos de outras classes, o que dificulta o isolamento (SILVA; SIQUEIRA; PAGLIARES, 2013). *Mock Objects* é uma técnica que gera objetos que simulam o comportamento de uma classe ou objeto real (ARAúJO, 2008). Com isso, pode-se proporcionar a substituição de partes do programa que são irrelevantes para um determinado teste de unidade (TILLMANN; SCHULTE, 2006). Então *Mock Objects* é geralmente usado como um auxílio para se testar uma unidade de um programa ignorando todo o resto do código (KONG; YIN, 2006), tornando-se um método importante para a realização do teste unitário.

A principal exigência da utilização de objetos *mock* é a implementação de interfaces,

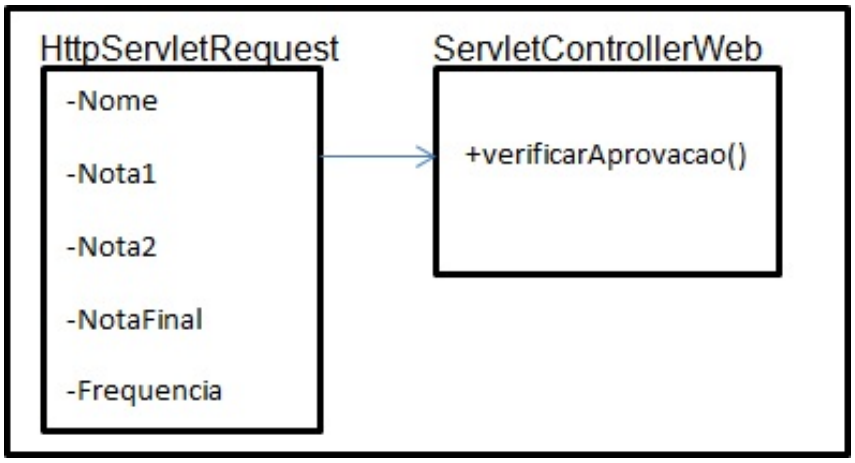
por serem uma boa prática de programação e a sua inclusão no modelo de classes não requerer muitas alterações (ARAúJO, 2008). Na próxima seção é apresentado as principais características da ferramenta EasyMock que auxilia na criação de objetos *mock* para linguagem Java e uma exemplificação do seu uso em testes unitários. Ferramentas como Mockito e JMock não são apresentadas por serem de artigos descartados nos critérios de seleção.

6.4.1 Ferramenta Easymock

O EasyMock é uma ferramenta para linguagem Java que oferece um mecanismo em gerar objetos *mock* de forma ágil para a construção de casos de teste, possibilitando que partes críticas do programa possam ser testadas automaticamente. O EasyMock é gratuito, de código aberto e que está disponível em http://sourceforge.net/easymock (ARAúJO, 2008).

A utilidade do EasyMock é apresentado aqui através de um exemplo que quer testar um método em Java que verifica se o aluno foi aprovado. Porém as informações são originadas de um formulário *web* onde ao preencher os campos Nome, Nota1, Nota2, NotaFinal e Frequencia e depois clicar no botão enviar, a página passa a requisição com os dados para o método verificarAprovacao, como mostra a interação entre os objetos da Figura 18. Logo é notório que se necessita das informações vindas do formulário *web* para se testar o método, então com EasyMock é possível simular uma requisição do tipo HttpServletRequest (ARAúJO, 2008).

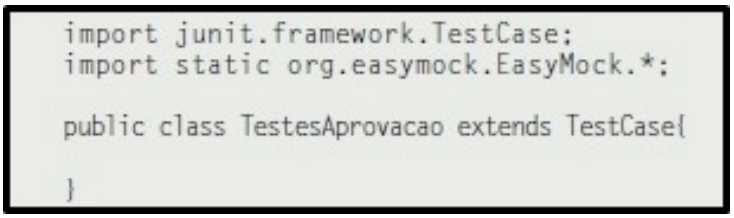
Figura 18 – Interação entre os objetos



Fonte: Adaptado de (ARAúJO, 2008).

Para iniciar o teste é essencial a criação de uma classe de teste que herda da classe TesteCase do JUnit, no exemplo chamada de TestesAprovacao. É preciso também a inclusão do *import static* da biblioteca EasyMock para poder criar os objetos mock, como mostra a Figura 19 (ARAúJO, 2008). Neste trabalho é relatado apenas um caso de teste como exemplo, pois o foco é apenas entender como é o procedimento da ferramenta.

Figura 19 – Criação da classe de teste

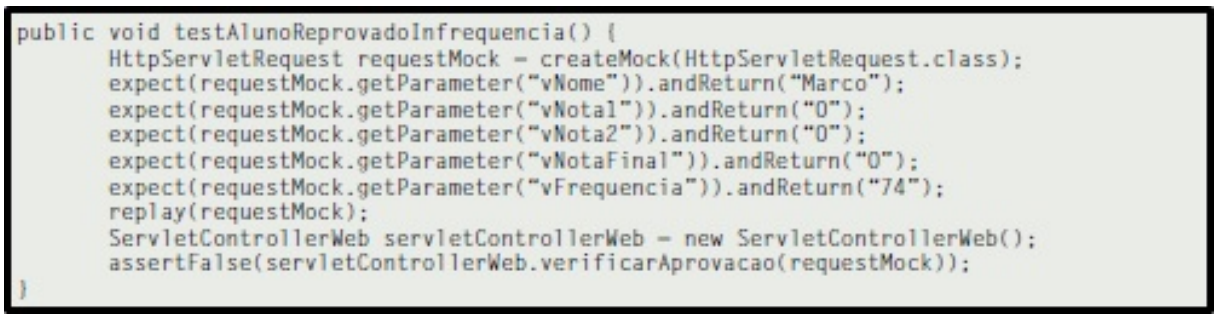


```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

public class TestesAprovacao extends TestCase{

}
```

Fonte: (ARAúJO, 2008).

O caso de teste implementado é o que testa a reprovação do aluno por frequência, e para isso é criado um método dentro da classe TestesAprovacao, chamado de testAlunoReprovadoInfrequencia, como mostra a Figura 20. O método se inicia com a criação do objeto requestMock, tal criação se assemelha a um objeto comum, com a diferença de que ao invés de chamar o construtor chama-se o método createMock(), passando como parâmetro a interface HttpServletRequest, que é o objeto falso. Porém como é um objeto falso ele não é capaz de retornar as chamadas dos métodos, então através do método expect o objeto *mock* é instruindo como ele deve proceder retornando o que for descrito no método andReturn(). Por fim utliza-se o método replay liberando o *mock* para o uso. Então, através da função AssertFalse do JUnit é possível avaliar se o método que está sendo testado retorna falso quando os dados do aluno são suficientes para ele não ser aprovado por causa da frequência (ARAúJO, 2008).

Figura 20 – Caso de teste testAlunoReprovadoInfrequencia



```
public void testAlunoReprovadoInfrequencia() {
        HttpServletRequest requestMock = createMock(HttpServletRequest.class);
        expect(requestMock.getParameter("vNome")).andReturn("Marco");
        expect(requestMock.getParameter("vNota1")).andReturn("0");
        expect(requestMock.getParameter("vNota2")).andReturn("0");
        expect(requestMock.getParameter("vNotaFinal")).andReturn("0");
        expect(requestMock.getParameter("vFrequencia")).andReturn("74");
        replay(requestMock);
        ServletControllerWeb servletControllerWeb = new ServletControllerWeb();
        assertFalse(servletControllerWeb.verificarAprovacao(requestMock));
}
```

Fonte: (ARAúJO, 2008).

O restante da abordagem do EasyMock está disponível no Anexo C. O conteúdo é relativo a instalação da ferramenta.

7 Análise dos Resultados

São abordados nesse capitulo os resultados da pesquisa, tanto dos métodos utilizados quanto das questões específicas do trabalho.

7.1 Aplicação dos Métodos

Com os métodos de busca e seleção de artigos definidos no Capítulo 2, os procedimentos foram postos em prática. Nas bases de pesquisa científicas foram obtidos vários artigos através das *strings* de buscas propostas (Tabela 1 e Tabela 2) e com o período de 17 anos (1999 a 2016) de publicação. A Science Direct foi uma base que pouco contribuiu para o trabalho, de acordo com a Tabela 8 é possível perceber que a quantidade de artigos encontrado e utilizado dessa fonte de pesquisa foi muito abaixo se comparado com a IEEE Xplore e ACM Digital Library.

Tabela 8 – Quantidade de artigos selecionados das bases científicas

Base	**Consulta Inicial**	**1º Critério de Seleção**	**2º Critério de Seleção**	**3º Critério de Seleção**
IEEE Xplore	1261	111	39	19
ACM Digital Library	1132	90	35	15
Science Direct	111	19	9	3
TOTAL	**2504**	**220**	**83**	**37**

Fonte: Autor.

De acordo com a Tabela 8, foram encontrados a partir do procedimento inicial de busca o total de 2504 artigos. À medida em que se seguiu as triagens previstas, a quantidade de artigos reduziu consideravelmente. Tanto que no final foram selecionados 37 artigos para serem estudados. Quanto ao período definido entre os anos de 1999 e 2016 é possível concluir que esse intervalo de tempo poderia ter sido reduzido, pois os artigos selecionados foram publicados entre os anos de 2004 e 2015, como mostra a Tabela 9.

Tabela 9 – Quantidade de artigos selecionados das bases científicas após o terceiro filtro por período

Base	**2015**	**2014**	**2013**	**2012**	**2011**	**2010**	**2009**	**2008**	**2007**	**2006**	**2005**	**2004**
IEEE Xplore	6	2		2	2	1	2		2	2		
ACM Digital Library	2	2	1		2	2		1	1	2	1	1
Science Direct			1	1			1					

Fonte: Autor.

Servindo como base de controle para determinar o grau de importância para esse trabalho, 37 arquivos estudados foram classificados com notas entre 1 e 5. Com isso, 20 artigos tiraram notas maior ou igual a 3, como mostra a Tabela 10. Logo, segundo os critérios relatados no Capítulo 2, foram esses os artigos selecionados das bases científicas para compor o trabalho.

Tabela 10 – Quantidade de artigos das bases científicas classificados por nota

Base	**1**	**1.5**	**2**	**2.5**	**3**	**3.5**	**4**	**4.5**	**5**
IEEE Xplore	1		1	4	1		5	1	6
ACM Digital Library	4		2	3	4				2
Science Direct		1	1			1			

Fonte: Autor.

As buscas nas revistas Testing Experience e Professional Tester Magazine foram realizadas de forma manual, por não existir uma interface de busca própria para as mesmas. Essas publicações periódicas que são destinadas ao teste de software, curiosamente não contribuíram para esse trabalho, pois não foram encontrados artigos que abrangesse o tema proposto. Já na revista Engenharia de Software Magazine foram encontrados 710 artigos no processo inicial e após as triagens subsequentes foram escolhidos 8 artigos para estudo, como mostra a Tabela 11.

Tabela 11 – Quantidade de artigos selecionados da revista Engenharia de Software Magazine

Revista	**Consulta Inicial**	**1º Critério de Seleção**	**2º Critério de Seleção**	**3º Critério de Seleção**
Engenharia de Software Magazine	710	72	13	8

Fonte: Autor.

A revista Engenharia de Software Magazine começou a ser publicada no ano de 2007 e os dados que subsidiam o presente trabalho foram coletados nas edições publicadas entre os anos de 2008 e 2013, como mostra a Tabela 12.

Tabela 12 – Quantidade de artigos selecionados da revista Engenharia de Software Magazine após o terceiro filtro por período

Revista	**2016**	**2015**	**2014**	**2013**	**2012**	**2011**	**2010**	**2009**	**2008**	**2007**
Engenharia de Software Magazine				1	1	1	2		3	

Fonte: Autor.

Os artigos selecionados para estudo também foram classificados por notas, utilizando como base os critérios descritos no Capítulo 2. Com isso, 6 artigos da revista foram considerados importantes para o trabalho, a Tabela 7 expõe a quantidade de artigos em cada pontuação.

Somando a quantidade de artigos selecionados das bases científicas com as revistas, foram escolhidos 26 artigos para compor o levantamento bibliográfico.

Tabela 13 – Quantidade de artigos da revista Engenharia de Software Magazine classificados por notas

Revista	**1**	**1.5**	**2**	**2.5**	**3**	**3.5**	**4**	**4.5**	**5**
Engenharia de Software Magazine	2				3				3

Fonte: Autor.

7.2 Análise das Pesquisas

O trabalho resulta na descrição de práticas existentes que automatizam o procedimento do teste unitário, além das técnicas que auxiliam na qualificação e apoio da atividade. Fornecendo um processo mais ágil, com diminuição de custos e com bons resultados. Pode-se perceber que a obtenção desse nível de qualidade passa pelo uso de ferramentas automatizadas, devido ao auxílio na diminuição das dificuldades em testar uma parte do código, gerar casos de teste que encontre falhas e que cubra grande parte do programa.

Existem duas categorias de ferramentas que automatizam o teste unitário. A primeira categoria é o *framework* que testa o código utilizando os casos de teste. A família xUnit é o arcabouço que possui implementações para várias linguagens, contendo uma estrutura padrão de teste com métodos já definidos, assim o testador não necessita programar elaboradas funções para testar aquela porção de código. Uma das implementações é o JUnit, o *framework* mais utilizado para linguagem Java. É importante ressaltar que muitos artigos encontrados nesse estudo fazem referência a família xUnit ou de suas implementações, reforçando a importância desse *framework* no cenário de teste unitário automatizado.

A outra categoria é aquela que gera automaticamente os casos de teste. São ferramentas que aceleram o processo, pois geram uma maior quantidade de casos de teste em um menor tempo se comparado com o testador que produz manualmente. Evosuite é uma das ferramentas que tem como característica principal o uso do algoritmo genético para a sua execução.

Há também técnicas que dão apoio ao teste unitário, tanto para qualificar os casos de teste quanto para executa-los. O teste de mutação procura informar se o teste elaborado pelo testador é suficiente para encontrar possíveis erros. Logo são gerados cópias do código fonte original acrescidos de erros e que são colocados aprova para analisar se os casos de teste os detectam. A ferramenta MuJava e o seu *plugin* do Eclipse MuClipse automatizam essa atividade para classes de linguagem Java. O uso dessas ferramentas são essenciais, pois o desenvolvedor gastaria mais tempo em elaborar o mesmo programa com erros diferentes de forma manual.

O teste de cobertura é outra técnica que informa o quanto do código fonte foi testado, especificando até mesmo as linhas que foram ou não cobertas. Utilizando a ferramenta JsCo-

verage para JavaScript torna-se prático essa análise. Há também a métrica chamada Complexidade Ciclomática, que determina a quantidade mínima necessária de casos de teste para abranger todo o código fonte. O Plugin Metrics é uma das ferramentas existente que exerçam esse cálculo para a linguagem Java.

A realização do teste em apenas uma parte do programa não é simples, uma unidade pode depender de outras que estão no sistema. Para isso existem os *Mock Objects*, que simulam o comportamento dos elementos que o programa precisa para ser executado corretamente, podendo então testar aquela unidade isoladamente. EasyMock é a ferramenta que gera Objetos *Mock* para linguagem Java.

Pode-se perceber, então, que a análise da pesquisa feita nesse capítulo responde aos questionamentos levantados nos objetivos deste trabalho, uma vez que foi possível apresentar as ferramentas *open source* de teste unitário e as que servem de apoio, além de suas características.

8 Considerações Finais

A atividade de teste é um recurso importante para desenvolver software de qualidade. No entanto, os cursos de graduação não exploram com ênfase esse processo, resultando em desenvolvedores não tão qualificados para a criação de bons testes. Devido a isso, o presente trabalho apresenta um levantamento bibliográfico sobre teste unitário automatizado de software, através de pesquisas nas bases científicas e em revistas técnicas, de modo a contribuir como fonte difusora do tema nos cursos de graduação, descrevendo categorias, técnicas e ferramentas a serem utilizadas nos testes unitários de software. Além do mais, esse material poderá auxiliar o desenvolvedor na execução de testes em seu próprio código fonte, por meio de casos qualificados.

A elaboração do caso de teste é a base fundamental para o teste unitário, é através daquele que se encontram as falhas existentes no programa. Devido a sua importância, há ferramentas que utilizam técnicas para cria-los e melhora-los. Ferramentas de teste de mutação, de cobertura e as que geram casos de teste são instrumentos que facilitam o trabalho do desenvolvedor em implementar o teste de boa qualidade.

Para executa-los, a família xUnit é a maior referência no que se diz respeito ao teste unitário de caixa branca. Ela possui ferramentas para linguagens distintas com a mesma definição de teste, baseando-se na elaboração de uma classe de teste onde se compara o resultado obtido da função com o resultado esperado. Uma das ferramentas desse conjunto é o JUnit, o *framework* mais utilizado para a linguagem Java.

Analisando as informações obtidas com o levantamento bibliográfico, pode-se perceber que para o desenvolvedor testar, de uma forma automatizada e de qualidade, uma porção do código que ele criou em linguagem Java (seja uma função, um método ou uma classe), primeiramente seria a elaboração de casos de teste com o Evosuite, que é uma ferramenta para esse fim de forma automática. No entanto, é importante o desenvolvedor também elaborar casos de teste, logo, o uso do Plugin Metrics proporciona um ponto de referência, pois a ferramenta calcula a complexidade ciclomática retornando a quantidade mínima necessária de casos de teste para cobrir todo aquele programa, sendo então uma informação para o desenvolvedor se basear de acordo com os seus objetivos no teste. Em seguida, o uso de ferramentas de cobertura como o EMMA e posteriormente a análise de mutação como MuClipse, auxiliam no melhoramento dos casos de teste, fazendo com que eles cubram o máximo do programa e que estejam aptas a encontrarem falhas.

Com os casos de teste elaborados, o passo seguinte é executa-los no programa através do JUnit, assim, este framework avalia aquela unidade retornando se foi encontrado falhas. Caso a execução do teste seja complexa devido à dificuldade em testar uma porção do código isolado de todo o resto, o EasyMock é uma ferramenta mock object que gera objetos que simulam o comportamento de partes do sistema que são necessárias para a execução daquele programa no JUnit. Assim sendo, o teste unitário automatizado é realizado com sucesso.

Existem outras ferramentas além das relatadas no presente trabalho. O JUnit tem como concorrente o TestNG, o Evosuite não é o único que gera casos de teste pois tem a companhia do JTExpert, EvoSuite-Mosa, T3, CT, GRT e Randoop. Dentro do teste de cobertura além do Jscoverage há também o Agitar, Bullseye, Clover, Cobertura, CodeTest, Dynamic, EMMA, eX-Vantage, gcov, Insure++, JCover, JTest, PurifyPlus, Semantic Designs (SD) e TCAT. O Mockito e JMock são outros que executam objetos *mock* para Java.

Todas essas últimas ferramentas, apesar de citadas nessa seção, não foram descritas no trabalho porque seus artigos eram deficientes quanto aos critérios de seleção ou porque não continham informações relevantes para comporem o texto. No entanto, a menção dos mesmos visa estimular o estudo em trabalhos futuros e, também, apresentar outras ferramentas, inclusive de outras linguagens, visto que a maioria das que foram descritas neste trabalho são para a linguagem Java.

Durante o estudo, foram descobertas outras fontes de pesquisa como o Núcleo de Apoio à Pesquisa em Software (NAPSoL) e o *Intenational Workshop Seach-Based Software Testing* (SBST). Então, é interessante como trabalhos futuros o complemento desse projeto utilizando essas bases.

Referências

ABREU, T. C. L. de; MOTA, L. da S.; ARAúJO, M. A. P. Seis sigma + cmmi: Métricas de software. *Engenharia de Software Magazine*, p. 50–55, 2010. ISSN 1983127-7. Citado nas páginas 37 e 38.

ACHARYA, S. *Mockito for Spring*. 1. ed. Birmingham: Packt Publishing, 2015. 166 p. ISBN 978-1-78398-378-0. Citado na página 24.

AHMED, Z.; ZAHOOR, M.; YOUNAS, I. Mutation operators for object-oriented systems: A survey. In: *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*. [S.l.: s.n.], 2010. v. 2, p. 614–618. Citado na página 30.

ARAúJO, M. A. P. Teste de software: Testes com objetos mock. *Engenharia de Software Magazine*, p. 42–47, 2008. ISSN 1983127-7. Citado nas páginas 41, 42, 43 e 74.

BERNARDO, P. C.; KON, F. Melhoria de processos de software com uso de análise causual de defeitos: A importância dos testes automatizados. *Engenharia de Software Magazine*, p. 54–57, 2008. ISSN 1983127-7. Citado nas páginas 23, 24, 25 e 26.

COTA, T. T. *Projeto de jogos móveis para idosos: um estudo com foco na motivação para jogar*. 2014. 25-28 p. Pontifícia Universidade Católica de Minas Gerais. Citado na página 15.

DAKA, E.; FRASER, G. A survey on unit testing practices and problems. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2014. p. 201–211. ISSN 1071-9458. Citado nas páginas 23 e 30.

DEZINGER, J. *Software Testing*. 2002. Citado na página 38.

DOMINGUES, A. L. dos S. *Avaliação de critérios e ferramentas de teste para programas OO*. 2002. Universidade de São Paulo - São Carlos. Citado na página 19.

FRASER, G.; ARCURI, A. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 24, n. 2, p. 8:1–8:42, dez. 2014. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/2685612>. Acesso em: 11 de novembro de 2016. Citado nas páginas 27, 28 e 29.

FRASER, G.; ARCURI, A. Evosuite at the sbst 2015 tool competition. In: *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*. [S.l.: s.n.], 2015. p. 25–27. Citado na página 27.

FRASER, G. et al. Does automated white-box test generation really help software testers? In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2013. (ISSTA 2013), p. 291–301. ISBN 978-1-4503-2159-4. Disponível em: <http://doi.acm.org/10.1145/2483760.2483774>. Acesso em: 11 de novembro de 2016. Citado nas páginas 26 e 27.

GAFFNEY, C.; TREFFTZ, C.; JORGENSEN, P. Tools for coverage testing: Necessary but not sufficient. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, USA, v. 20, n. 1, p. 27–33, out. 2004. ISSN 1937-4771. Disponível em: <http://dl.acm.org/citation.cfm?id=1040231.1040235>. Acesso em: 11 de novembro de 2016. Citado nas páginas 22, 38 e 39.

HARROLD, M. J. Testing: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 61–72. ISBN 1-58113-253-0. Disponível em: <http://doi.acm.org/10.1145/336512.336532>. Acesso em: 11 de novembro de 2016. Citado na página 21.

JORGENSEN, P. C. *Software Testing: A Craftsman's Approach*. 2. ed. Boca Raton: CRC Press LLC, 2002. ISBN 0-8493-0809-7. Citado na página 39.

KONG, L.; YIN, Z. The extension of the unit testing tool junit for special testings. In: *Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on*. [S.l.: s.n.], 2006. v. 2, p. 410–415. Citado na página 41.

LAGES, D. S. Gestão de ti: Reportando de forma simples os resultados dos testes. *Engenharia de Software Magazine*, p. 40–48, 2010. ISSN 1983127-7. Citado na página 23.

LEITNER, A. et al. Reconciling manual and automated testing: The autotest experience. In: *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. [S.l.: s.n.], 2007. p. 261a–261a. ISSN 1530-1605. Citado nas páginas 26 e 27.

LUFT, C. C. *Teste de software: Uma necessidade das empresas*. 2012. Universidade Regional Do Noroeste Do Estado Do Rio Grande do Sul. Citado na página 12.

MALDONADO, J. C. et al. *INTRODUÇÃO AO TESTE DE SOFTWARE*. 2004. Instituto de Ciências Matemáticas e de Computação - São Carlos. Citado nas páginas 18, 20 e 21.

MYERS, G. J. *The Art of Software Testing*. 2. ed. Hoboken: John WileyeSons Inc., 2004. ISBN 0-471-46912-2. Citado nas páginas 12, 18 e 19.

PFLEEGER, S. L. *Engenharia de Software: teoria e prática*. 2. ed. São Paulo: Prentice Hall, 2004. Citado na página 18.

PRASETYA, I. S. W. B. T3: Benchmarking at third unit testing tool contest. In: *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*. [S.l.: s.n.], 2015. p. 44–47. Citado na página 27.

PRESSMAN, R. S. *Software engineering – a practitioner's approach*. 5. ed. [S.l.]: McGraw-Hill, 2000. Citado na página 19.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: Uma abordagem profissional*. 8. ed. Porto Alegre: AMGHl, 2016. 940 p. Citado nas páginas 12 e 18.

RAFI, D. M. et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: *Automation of Software Test (AST), 2012 7th International Workshop on*. [S.l.: s.n.], 2012. p. 36–42. Citado nas páginas 15, 21 e 22.

RAMLER, R.; KASPAR, T. Applicability and benefits of mutation analysis as an aid for unit testing. In: *Computing and Convergence Technology (ICCCT), 2012 7th International Conference on*. [S.l.: s.n.], 2012. p. 920–925. Citado nas páginas 33 e 34.

RAMLER, R.; WOLFMAIER, K. Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2006. (AST '06), p. 85–91. ISBN 1-59593-408-1. Disponível em: <http://doi.acm.org/10.1145/1138929.1138946>. Acesso em: 11 de novembro de 2016. Citado na página 21.

RIBEIRO, V. V.; ARAúJO, M. A. Maturidade + evolução: Testes de mutação com o plug-in muclipse. *Engenharia de Software Magazine*, p. 44–51, 2008. ISSN 1983127-7. Citado nas páginas 34, 35, 36, 68, 69, 70 e 71.

RIBEIRO, V. V.; ARAúJO, M. A. P. Gerência de tempo: Uma abordagem para escolha de operadores de mutação. *Engenharia de Software Magazine*, p. 46–55, 2012. ISSN 1983127-7. Citado nas páginas 30, 31, 32 e 33.

ROJAS, J. M.; FRASER, G.; ARCURI, A. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2015. (ISSTA 2015), p. 338–349. ISBN 978-1-4503-3620-8. Disponível em: <http://doi.acm.org/10.1145/2771783.2771801>. Acesso em: 11 de novembro 2016. Citado na página 27.

SILVA, D. M. da; SIQUEIRA, R. D. de; PAGLIARES, R. M. Reuso na prática: Aplicando teste unitário com dublês de teste. *Engenharia de Software Magazine*, p. 40–47, 2013. ISSN 1983127-7. Citado na página 41.

SMITH, B. H.; WILLIAMS, L. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, v. 82, n. 11, p. 1819 – 1832, 2009. ISSN 0164-1212. SI: {TAIC} {PART} 2007 and {MUTATION} 2007. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121209001368>. Acesso em: 11 de novembro de 2016. Citado na página 30.

SMITH, B. H.; WILLIANS, L. *An Empirical Evaluation of the MuJava Mutation Operators*. 2007. Citado na página 30.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo: Pearson Addison Wesley, 2009. Citado nas páginas 18 e 19.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. São Paulo: Pearson Prentice Hall„ 2011. 529 p. Citado na página 18.

TILLMANN, N.; SCHULTE, W. Mock-object generation with behavior. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. [S.l.: s.n.], 2006. p. 365–368. ISSN 1938-4300. Citado na página 41.

TOLEDO, J. V. et al. Gestão de ti: Teste unitário e de cobertura para java script com jsunit e jscoverage. *Engenharia de Software Magazine*, p. 49–56, 2010. ISSN 1983127-7. Citado nas páginas 39, 40, 41, 72 e 73.

WAHID, M.; ALMALAISE, A. Junit framework: An interactive approach for basic unit testing learning in software engineering. In: *Engineering Education (ICEED), 2011 3rd International Congress on*. [S.l.: s.n.], 2011. p. 159–164. Citado nas páginas 13, 23 e 24.

WANG, S.; OFFUTT, J. Comparison of unit-level automated test generation tools. In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*. [S.l.: s.n.], 2009. p. 210–219. Citado nas páginas 13, 23 e 34.

WIKLUND, K. et al. Impediments for automated testing – an empirical analysis of a user support discussion board. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. [S.l.: s.n.], 2014. p. 113–122. ISSN 2159-4848. Citado na página 21.

YANG, Q.; LI, J. J.; WEISS, D. A survey of coverage based testing tools. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2006. (AST '06), p. 99–103. ISBN 1-59593-408-1. Disponível em: <http://doi.acm.org/10.1145/1138929.1138949>. Acesso em: 11 de novembro de 2016. Citado nas páginas 22 e 38.

ZHU, H. Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In: *Trustworthy Systems and Their Applications (TSA), 2015 Second International Conference on*. [S.l.: s.n.], 2015. p. 8–15. Citado na página 26.

APÊNDICE A – ARTIGOS DESCARTADOS

Este apêndice apresenta os 147 artigos não repetidos das bases científicas e da revista Engenharia de Software Magazine que foram descartados a partir do segundo critério de seleção. As referências estão no formato disponibilizado pelas bases de pesquisa científicas, enquanto que nas revistas estão no formato padrão.

- A. Bansal, M. Muli and K. Patil, "Taming complexity while gaining efficiency: Requirements for the next generation of test automation tools,"2013 IEEE AUTOTESTCON, Schaumburg, IL, 2013, pp. 1-6. doi: 10.1109/AUTEST.2013.6645055;
- A. J. H. Simons and C. D. Thomson, "Lazy Systematic Unit Testing: JWalk versus JUnit,"Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007, Windsor, 2007, pp. 138-138. doi: 10.1109/TAIC.PART.2007.14;
- A. J. Thomson, "Benchmarking Effectiveness for Object-Oriented Unit Testing,"Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on, Lillehammer, 2008, pp. 375-379. doi: 10.1109/ICSTW.2008.10;
- A. K. Jha, "Development of test automation framework for testing avionics systems,"Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th, Salt Lake City, UT, 2010, pp. 6.E.5-1-6.E.5-11. doi: 10.1109/DASC.2010.5655445;
- A. M. Fard, A. Mesbah and E. Wohlstadter, "Generating Fixtures for JavaScript Unit Testing (T),"Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, Lincoln, NE, 2015, pp. 190-200. doi: 10.1109/ASE.2015.26;
- A. Mansoor, "Analytical survey on automated software test data evaluation,"New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on, Gyeongju, 2010, pp. 580-585;
- A. Méndez-Porras, J. Alfaro-Velásco, M. Jenkins, A. M. Porras and A. Méndez-Porras, "Automated testing framework for mobile applications based in user-interaction features and historical bug information,"Computing Conference (CLEI), 2015 Latin American, Arequipa, 2015, pp. 1-8. doi: 10.1109/CLEI.2015.7359996;
- A. Panichella, F. M. Kifetew and P. Tonella, "Results for EvoSuite – MOSA at the Third Unit Testing Tool Competition,"Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on, Florence, 2015, pp. 28-31.doi: 10.1109/SBST.2015.14

- A. Sakti, G. Pesant and Y. G. Guéhéneuc, "JTExpert at the Third Unit Testing Tool Competition,"
  Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on, Florence, 2015, pp. 52-55.doi: 10.1109/SBST.2015.20

- A. Z. Javed, P. A. Strooper and G. N. Watson, "Automated Generation of Test Cases Using Model-Driven Architecture,"Automation of Software Test , 2007. AST '07. Second International Workshop on, Minneapolis, MN, 2007, pp. 3-3. doi: 10.1109/AST.2007.2;

- Abdulhadi Celenlioglu, Birsen G. Ozdemir, Effective Testing with JSFUnit for Educational Applications, Procedia - Social and Behavioral Sciences, Volume 47, 2012, Pages 2031-2035, ISSN 1877-0428, http://dx.doi.org/10.1016/j.sbspro.2012.06.944;

- Alex Gerdes, John Hughes, Nick Smallbone, and Meng Wang. 2015. Linking unit tests and properties. In Proceedings of the 14th ACM SIGPLAN Workshop on Erlang (Erlang 2015). ACM, New York, NY, USA, 19-26. DOI=http://dx.doi.org/10.1145/2804295.2804298

- Andrew Patterson, Michael Kölling, and John Rosenberg. 2003. Introducing unit testing with BlueJ. In Proceedings of the 8th annual conference on Innovation and technology in computer science education (ITiCSE '03), David Finkel (Ed.). ACM, New York, NY, USA, 11-15. DOI=http://dx.doi.org/10.1145/961511.961518;

- Arindam Chakrabarti and Patrice Godefroid. 2006. Software partitioning for effective automated unit testing. In Proceedings of the 6th ACM e IEEE International conference on Embedded software (EMSOFT '06). ACM, New York, NY, USA, 262-271.
  DOI=http://dx.doi.org/10.1145/1176887.1176925;

- Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2colic testing. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, 37-47.
  DOI=http://dx.doi.org/10.1145/2491411.2491453;

- B. Daniel et al., "ReAssert: a tool for repairing broken unit tests,"2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, 2011, pp. 1010-1012. doi: 10.1145/1985793.1985978;

- BRUNO, E. A.; SILVA, P. C. B. da; ALVES, T. S. Testes funcionais de software: Kanban desafios e a receita para o sucesso. Engenharia de Software Magazine, p. 37–41, 2012. ISSN 1983127-7.

- C. D. Nguyen, B. Mendelson, D. Citron, O. Shehory, T. E. J. Vos and N. Condori-Fernández, "Evaluating the FITTEST Automated Testing Tools: An Industrial Case Study,"2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, 2013, pp. 332-339. doi: 10.1109/ESEM.2013.61;

- CAETANO, C. Gestão de defeitos: Qualidade de software. Engenharia de Software Magazine, Edição Especial, p. 60–67, 2007. ISSN 1983127-7.

- CAETANO, C. Testes ágeis: Teste de software. Engenharia de Software Magazine, p. 10–15, 2012. ISSN 1983127-7.
- Chang Liu. 2000. Platform-independent and tool-neutral test descriptions for automated software testing. In Proceedings of the 22nd international conference on Software engineering (ICSE '00). ACM, New York, NY, USA, 713-715. DOI=http://dx.doi.org/10.1145/337180.337598;
- Charles D. Allison. 2007. The simplest unit test tool that could possibly work. J. Comput. Sci. Coll.23, 1 (October 2007), 183-189.
- Chin-Yu Huang, Chung-Sheng Chen, Chia-En Lai, Evaluation and analysis of incorporating Fuzzy Expert System approach into test suite reduction, Information and Software Technology, Volume 79, November 2016, Pages 79-105, ISSN 0950-5849, http://dx.doi.org/10.1016/j.infsof.2016.07.005;
- COLLINS, E.; LOBãO, L. Experiência em automação de testes: Mps.br. Engenharia de Software Magazine, p. 05–10, 2010. ISSN 1983127-7.
- COLLINS, E.; LOBãO, L. Processo e automação de testes de software: Automação de testes. Engenharia de Software Magazine, p. 26–33, 2010. ISSN 1983127-7.
- D. Shao, S. Khurshid and D. E. Perry, "A Case for White-box Testing Using Declarative Specifications Poster Abstract,"Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007, Windsor, 2007, pp. 137-137. doi: 10.1109/TAIC.PART.2007.36;
- D. Tao, Z. Lin and C. Lu, "Cloud platform based automated security testing system for mobile internet,"in Tsinghua Science and Technology, vol. 20, no. 6, pp. 537-544, December 2015. doi: 10.1109/TST.2015.7349926;
- Darko Marinov and Wolfram Schulte. 2008. Workshop on state-space exploration for automated testing (SSEAT 2008). In Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08). ACM, New York, NY, USA, 315-316. DOI=http://dx.doi.org/10.1145/1390630.1390672;
- David Saff and Michael D. Ernst. 2004. Mock object creation for test factoring. In Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering(PASTE '04). ACM, New York, NY, USA, 49-51. DOI=http://dx.doi.org/10.1145/996821.996838;
- David Saff. 2007. From developer's head to developer tests: characterization, theories, and preventing one more bug. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07). ACM, New York, NY, USA, 811-812. DOI=http://dx.doi.org/10.1145/1297846.1297900;
- Dirk Riehle. 2008. JUnit 3.8 documented using collaborations. SIGSOFT Softw. Eng. Notes 33, 2, Article 5 (March 2008), 28 pages. DOI=http://dx.doi.org/10.1145/1350802.1350812;

- Don Blaheta. 2015. Unci: a C++-based Unit-testing Framework for Intro Students. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 475-480;
- E. Alégroth, M. Nass and H. H. Olsson, "JAutomate: A Tool for System- and Acceptance-test Automation,"2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luembourg, 2013, pp. 439-446. doi: 10.1109/ICST.2013.61;
- E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems,"2014 IEEE 25th International Symposium on Software Reliability Engineering, Naples, 2014, pp. 201-211. doi: 10.1109/ISSRE.2014.11;
- E. Diaz, J. Tuya and R. Blanco, "Automated software testing using a metaheuristic technique based on Tabu search,"Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, 2003, pp. 310-313. doi: 10.1109/ASE.2003.1240327;
- Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. 2002. An experience in integrating automated unit testing practices in an introductory programming course. SIGCSE Bull. 34, 4 (December 2002), 125-128. DOI=http://dx.doi.org/10.1145/820127.820183;
- Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. 2008. A test generation solution to automate software testing. In Proceedings of the 3rd international workshop on Automation of software test (AST '08). ACM, New York, NY, USA, 45-48. DOI=http://dx.doi.org/10.1145/1370042.1370052;
- FURTADO, G. C. F.; PINTO, A.; ALBUQUERQUE, R. Customização e integração de ferramentas open-source: Seis sigma + cmmi. Engenharia de Software Magazine, p. 43–49, 2010. ISSN 1983127-7.
- G. K. Thiruvathukal, K. Laufer and B. Gonzalez, "Unit Testing Considered Useful,"in Computing in Science e Engineering, vol. 8, no. 6, pp. 76-87, Nov.-Dec. 2006. doi: 10.1109/MCSE.2006.124;
- G. S. D. Oliveira and A. Duarte, "A Framework for Automated Software Testing on the Cloud,"2013 International Conference on Parallel and Distributed Computing, Applications and Technologies, Taipei, 2013, pp. 344-349. doi: 10.1109/PDCAT.2013.61;
- Gábor Szeder. 2009. Unit testing for multi-threaded Java programs. In Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '09). ACM, New York, NY, USA, , Article 4 , 8 pages. DOI=http://dx.doi.org/10.1145/1639622.1639626;
- Gordon Fraser and Andreas Zeller. 2011. Generating parameterized unit tests. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, New York, NY, USA, 364-374. DOI=http://dx.doi.org/10.1145/2001420.2001464;

- Gordon Fraser, Franz Wotawa, Paul Ammann, Issues in using model checkers for test case generation, Journal of Systems and Software, Volume 82, Issue 9, September 2009, Pages 1403-1418, ISSN 0164-1212, http://dx.doi.org/10.1016/j.jss.2009.05.016;
- Guoqing Xu, Zongyuan Yang, Haitao Huang, Qian Chen, Ling Chen and Fengbin Xu, "JAOUT: automated generation of aspect-oriented unit test,"Software Engineering Conference, 2004. 11th Asia-Pacific, 2004, pp. 374-381.doi: 10.1109/APSEC.2004.63
- Guy Collins Ndem, Abbas Tahir, Andreas Ulrich, and Helmut Goetz. 2011. Test data to reduce the complexity of unit test automation. In Proceedings of the 6th International Workshop on Automation of Software Test (AST '11). ACM, New York, NY, USA, 105-106. DOI=http://dx.doi.org/10.1145/1982595.1982618;
- H. Yu, Y. Lan and H. Ren, "The Research about an Automated Software Testing System RunTool,"Intelligent Systems and Applications (ISA), 2011 3rd International Workshop on, Wuhan, 2011, pp. 1-4. doi: 10.1109/ISA.2011.5873331;
- HABIB, E. Testes ágeis: Especificação de requisitos. Engenharia de Software Magazine, p. 07–13, 2010. ISSN 1983127-7.
- Hong Zhu, Joseph R. Horgan, S. C. Cheung, and J. Jenny Li. 2006. The first international workshop on automation of software test. In Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 1028-1029. DOI=http://dx.doi.org/10.1145/1134285.1134487;
- J. H. Andrews, "A case study of coverage-checked random data structure testing, "Automated Software Engineering, 2004. Proceedings. 19th International Conference on, Linz, 2004, pp. 316-319. doi: 10.1109/ASE.2004.1342755;
- J. Takahashi and Y. Kakuda, "Effective automated testing: a solution of graphical object verification,"Test Symposium, 2002. (ATS '02). Proceedings of the 11th Asian, Guam, USA, 2002, pp. 284-291. doi: 10.1109/ATS.2002.1181725;
- J. Wloka, B. G. Ryder and F. Tip, "JUnitMX - A change-aware unit testing tool,"2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, 2009, pp. 567-570. doi: 10.1109/ICSE.2009.5070557;
- J. Zhan, H. Zhang, B. Zou and X. Li, "Research on Automated Testing of the Trusted Platform Model,"Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for, Hunan, 2008, pp. 2335-2339. doi: 10.1109/ICYCS.2008.533;
- James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. 2006. Tool support for randomized unit testing. In Proceedings of the 1st international workshop on Random testing (RT '06). ACM, New York, NY, USA, 36-45. DOI=http://dx.doi.org/10.1145/1145735.1145741;
- Javier Andrade, Juan Ares, María-Aurora Martínez, Juan Pazos, Santiago Rodríguez, Julio Romera, Sonia Suárez, An architectural model for software testing lesson learned

systems, Information and Software Technology, Volume 55, Issue 1, January 2013, Pages 18-34, ISSN 0950-5849, http://dx.doi.org/10.1016/j.infsof.2012.03.003;

- Jochen Schimmel, Korbinian Molitorisz, Ali Jannesari, and Walter F. Tichy. 2015. Combining unit tests for data race detection. In Proceedings of the 10th International Workshop on Automation of Software Test (AST '15). IEEE Press, Piscataway, NJ, USA, 43-47;

- Johannes Link, Chapter 2 - Automating Unit Tests, In The Morgan Kaufmann Series in Software Engineering and Programming, Morgan Kaufmann, San Francisco, 2003, Pages 23-38, Unit Testing in Java, ISBN 9781558608689, http://dx.doi.org/10.1016/B978-155860868-9/50004-3.

- José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, New York, NY, USA, 55-66. DOI=http://dx.doi.org/10.1145/2642937.2643002;

- K. Yatoh, K. Sakamoto, F. Ishikawa and S. Honiden, "ArbitCheck: A Highly Automated Property-Based Testing Tool for Java,"Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on, Cleveland, OH, 2014, pp. 405-412. doi: 10.1109/ICSTW.2014.68;

- Katarina Grolinger, Miriam A.M. Capretz, A unit test approach for database schema evolution, Information and Software Technology, Volume 53, Issue 2, February 2011, Pages 159-170, ISSN 0950-5849, http://dx.doi.org/10.1016/j.infsof.2010.10.002;

- Kevin Buffardi and Stephen H. Edwards. 2012. Exploring influences on student adherence to test-driven development. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (ITiCSE '12). ACM, New York, NY, USA, 105-110. DOI=10.1145/2325296.2325324;

- Konrad Iwanicki, Przemyslaw Horban, Piotr Glazar, and Karol Strzelecki. 2014. Bringing Modern Unit Testing Techniques to Sensornets. ACM Trans. Sen. Netw. 11, 2, Article 25 (August 2014), 41 pages. DOI=http://dx.doi.org/10.1145/2629422;

- Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13). ACM, New York, NY, USA, 263-272.
DOI=http://dx.doi.org/10.1145/1081706.1081750;

- L. Kong, H. Zhu and B. Zhou, "Automated Testing EJB Components Based on Algebraic Specifications,"Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, Beijing, 2007, pp. 717-722.
doi: 10.1109/COMPSAC.2007.82;

- L. Mukkavilli, "Smart Unit Testing Framework,"Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on, Dallas, TX, 2012, pp. 70-79. doi: 10.1109/ISSREW.2012.45;
- L. Padgham, Z. Zhang, J. Thangarajah and T. Miller, "Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems,"in IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1230-1244, Sept. 2013. doi: 10.1109/TSE.2013.10;
- L. Williams, G. Kudrjavets and N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft,"2009 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, 2009, pp. 81-89.doi: 10.1109/ISSRE.2009.32
- LAGES, D. S. Automação dos testes: um lobo na pele de cordeiro?: Automação de testes. Engenharia de Software Magazine, p. 20–25, 2010. ISSN 1983127-7.
- Lars-Ola Damm, Lars Lundberg, David Olsson, Introducing Test Automation and Test-Driven Development: An Experience Report, Electronic Notes in Theoretical Computer Science, Volume 116, 2005, Pages 3-15, ISSN 1571-0661, http://dx.doi.org/10.1016/j.entcs.2004.02.090;
- Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, , Article 33 , 11 pages. DOI=http://dx.doi.org/10.1145/2393596.2393634;
- Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2013. TestEvol: a tool for analyzing test-suite evolution. In Proceedings of the 2013 International Conference on Software Engineering(ICSE '13). IEEE Press, Piscataway, NJ, USA, 1303-1306;
- M. Catelani, L. Ciani, V. L. Scarano and A. Bacioccola, "A Novel Approach To Automated Testing To Increase Software Reliability,"Instrumentation and Measurement Technology Conference Proceedings, 2008. IMTC 2008. IEEE, Victoria, BC, 2008, pp. 1499-1502. doi: 10.1109/IMTC.2008.4547280;
- M. Kim, Y. Kim and H. Kim, "A Comparative Study of Software Model Checkers as Unit Testing Tools: An Industrial Case Study,"in IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 146-160, March-April 2011. doi: 10.1109/TSE.2010.68;
- M. Ó Cinnéide, D. Boyle and I. H. Moghadam, "Automated Refactoring for Testability, "Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, Berlin, 2011, pp. 437-443. doi: 10.1109/ICSTW.2011.23;
- M. P. Prado, E. Verbeek, M. A. Storey and A. M. R. Vincenzi, "WAP: Cognitive aspects in unit testing: The hunting game and the hunter's perspective,"Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on, Gaithersbury, MD, 2015, pp. 387-392. doi: 10.1109/ISSRE.2015.7381832;

- Marcantonio Catelani, Lorenzo Ciani, Valeria L. Scarano, Alessandro Bacioccola, Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use, Computer Standards e Interfaces, Volume 33, Issue 2, February 2011, Pages 152-158, ISSN 0920-5489, http://dx.doi.org/10.1016/j.csi.2010.06.006;

- Mark Harman. 2007. Automated Test Data Generation using Search Based Software Engineering. In Proceedings of the Second International Workshop on Automation of Software Test (AST '07). IEEE Computer Society, Washington, DC, USA, 2-. DOI=http://dx.doi.org/10.1109/AST.2007.4;

- Mark Last, Menahem Friedman, and Abraham Kandel. 2003. The data mining approach to automated software testing. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '03). ACM, New York, NY, USA, 388-396. DOI=http://dx.doi.org/10.1145/956750.956795;

- Martin Burger and Andreas Zeller. 2011. Minimizing reproduction of software failures. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, New York, NY, USA, 221-231. DOI=http://dx.doi.org/10.1145/2001420.2001447;

- Martin K. Brown. 2010. A framework for parallel unit testings: work in progress. In Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10). ACM, New York, NY, USA, , Article 110 , 4 pages. DOI=http://dx.doi.org/10.1145/1900008.1900150;

- Matt Staats. 2010. The influence of multiple artifacts on the effectiveness of software testing. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). ACM, New York, NY, USA, 517-522. DOI=10.1145/1858996.1859100;

- Michael Olan. 2003. Unit testing: test early, test often. J. Comput. Sci. Coll. 19, 2 (December 2003), 319-328;

- Michael Wick, Daniel Stevenson, and Paul Wagner. 2005. Using testing and JUnit across the curriculum. SIGCSE Bull. 37, 1 (February 2005), 236-240. DOI=http://dx.doi.org/10.1145/1047124.1047427;

- Mike Papadakis, Nicos Malevris, and Maria Kallia. 2010. Towards automating the generation of mutation tests. In Proceedings of the 5th Workshop on Automation of Software Test (AST '10). ACM, New York, NY, USA, 111-118. DOI=http://dx.doi.org/10.1145/1808266.1808283;

- Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An empirical evaluation and comparison of manual and automated test selection. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, New York, NY, USA, 361-372. DOI=http://dx.doi.org/10.1145/2642937.2643019;

- Mohamed A. Khamis, Khaled Nagi, Designing multi-agent unit tests using systematic test design patterns-(extended version), Engineering Applications of Artificial Intelligence, Volume 26, Issue 9, October 2013, Pages 2128-2142, ISSN 0952-1976, http://dx.doi.org/10.1016/j.engappai.2013.04.009.

- N. Aleb and S. Kechid, "Path coverage testing in the cloud,"Communications and Information Technology (ICCIT), 2012 International Conference on, Hammamet, 2012, pp. 118-123. doi: 10.1109/ICCITechnol.2012.6285773;

- N. H. Saad and N. S. Awang Abu Bakar, "Automated testing tools for mobile applications,"Information and Communication Technology for The Muslim World (ICT4M), 2014 The 5th International Conference on, Kuching, 2014, pp. 1-5. doi: 10.1109/ICT4M.2014.7020665;

- N. Tillmann, J. de Halleux and T. Xie, "Parameterized unit testing: theory and practice,"2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, 2010, pp. 483-484. doi: 10.1145/1810295.1810441;

- Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13). ACM, New York, NY, USA, 253-262. DOI=http://dx.doi.org/10.1145/1081706.1081749

- Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. 2010. Parameterized unit testing: theory and practice. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10), Vol. 2. ACM, New York, NY, USA, 483-484;

- Ossi Taipale and Kari Smolander. 2006. Improving software testing by observing practice. In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering(ISESE '06). ACM, New York, NY, USA, 262-271. DOI=http://dx.doi.org/10.1145/1159733.1159773;

- P. Dhavachelvan, G.V. Uma, Multi-agent-based integrated framework for intra-class testing of object-oriented software, Applied Soft Computing, Volume 5, Issue 2, January 2005, Pages 205-222, ISSN 1568-4946, http://dx.doi.org/10.1016/j.asoc.2004.04.004;

- P. Louridas, "JUnit: unit testing and coiling in tandem,"in IEEE Software, vol. 22, no. 4, pp. 12-15, July-Aug. 2005. doi: 10.1109/MS.2005.100;

- P. Runeson, "A survey of unit testing practices,"in IEEE Software, vol. 23, no. 4, pp. 22-29, July-Aug. 2006. doi: 10.1109/MS.2006.91;

- P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann and D. Lo, "Understanding the Test Automation Culture of App Developers,"2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, 2015, pp. 1-10. doi: 10.1109/ICST.2015.7102609;

- Pandimurugan, M. parvathi and A. jenila, "A survey of software testing in refactoring based software models,"Nanoscience, Engineering and Technology (ICONSET), 2011 International Conference on, Chennai, 2011, pp. 571-573.doi: 10.1109/ICONSET.2011.6168034
- Patricia Lutsky, Information extraction from documents for automating software testing, Artificial Intelligence in Engineering, Volume 14, Issue 1, January 2000, Pages 63-69, ISSN 0954-1810, http://dx.doi.org/10.1016/S0954-1810(99)00024-2;
- PATUCI, G. de O. Ferramentas de teste de software: Agilidade. Engenharia de Software Magazine, p. 19–26, 2011. ISSN 1983127-7.
- Percy Pari Salas, Padmanabhan Krishnan, Automated Software Testing of Asynchronous Systems, Electronic Notes in Theoretical Computer Science, Volume 253, Issue 2, 2009, Pages 3-19, ISSN 1571-0661, http://dx.doi.org/10.1016/j.entcs.2009.09.048;
- Peter Sommerlad and Emanuel Graf. 2007. CUTE: C++ unit testing easier. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07). ACM, New York, NY, USA, 783-784. DOI=http://dx.doi.org/10.1145/1297846.1297886;
- R. B. Wen, "URL-driven automated testing,"Quality Software, 2001. Proceedings.Second Asia-Pacific Conference on, Hong Kong, 2001, pp. 268-272.
doi: 10.1109/APAQS.2001.990029;
- R. P. Tan and S. Edwards, "Evaluating Automated Unit Testing in Sulu,"2008 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, 2008, pp. 62-71. doi: 10.1109/ICST.2008.59;
- R. Ramler, K. Wolfmaier and T. Kopetzky, "A Replicated Study on Random Test Case Generation and Manual Unit Testing: How Many Bugs Do Professional Developers Find?, "Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual, Kyoto, 2013, pp. 484-491. doi: 10.1109/COMPSAC.2013.82;
- Rainer Gerlich, Ralf Gerlich, and Thomas Boll. 2007. Random testing: from the classical approach to a global view and full test automation. In Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007) (RT '07). ACM, New York, NY, USA, 30-37. DOI=http://dx.doi.org/10.1145/1292414.1292424;
- René Just and Franz Schweiggert. 2010. Automating software tests with partial oracles in integrated environments. In Proceedings of the 5th Workshop on Automation of Software Test(AST '10). ACM, New York, NY, USA, 91-94.
DOI=http://dx.doi.org/10.1145/1808266.1808280;
- RIBEIRO, V. V.; ALMEIDA, F. N. de; ARAúJO, M. A. P. Integração contínua com hudson, maven2, testng e subversion: Seis sigma + cmmi. Engenharia de Software Magazine, p. 56–62, 2010. ISSN 1983127-7.

- Richard Carlsson and Mickaël Rémond. 2006. EUnit: a lightweight unit testing framework for Erlang. In Proceedings of the 2006 ACM SIGPLAN workshop on Erlang (ERLANG '06). ACM, New York, NY, USA, 1-1. DOI=http://dx.doi.org/10.1145/1159789.1159791;

- ROSáRIO, J. C. do; SANTOS, I. R. dos; MARCHI, J. D. Aumente a qualidade de seu software com testes: Computação em nuvem. Engenharia de Software Magazine, p. 51–54, 2012. ISSN 1983127-7.

- S. A. Jolly, V. Garousi and M. M. Eskandar, "Automated Unit Testing of a SCADA Control Software: An Industrial Case Study Based on Action Research,"2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, 2012, pp. 400-409. doi: 10.1109/ICST.2012.120;

- S. Bauersfeld, T. E. J. Vos, K. Lakhotia, S. Poulding and N. Condori, "Unit Testing Tool Competition,"Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, Luxembourg, 2013, pp. 414-420.
doi: 10.1109/ICSTW.2013.55;

- S. H. Kuk and H. S. Kim, "Automatic Generation of Testing Environments for Web Applications,"Computer Science and Software Engineering, 2008 International Conference on, Wuhan, Hubei, 2008, pp. 694-697. doi: 10.1109/CSSE.2008.1026;

- S. L. Eddins, "Automated Software Testing for Matlab,"in Computing in Science e Engineering, vol. 11, no. 6, pp. 48-55, Nov.-Dec. 2009. doi: 10.1109/MCSE.2009.186;

- S. R. Choudhary, A. Gorla and A. Orso, "Automated Test Input Generation for Android: Are We There Yet? (E),"Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, Lincoln, NE, 2015, pp. 429-440. doi: 10.1109/ASE.2015.89;

- S. R. Shahamiri, W. M. N. W. Kadir and S. Z. Mohd-Hashim, "A Comparative Study on Automated Software Test Oracle Methods,"Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on, Porto, 2009, pp. 140-145. doi: 10.1109/ICSEA.2009.29;

- S. Steenbuck and G. Fraser, "Generating Unit Tests for Concurrent Classes,"2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luembourg, 2013, pp. 144-153. doi: 10.1109/ICST.2013.33;

- Sai Zhang. 2011. Palus: a hybrid automated test generation tool for java. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 1182-1184. DOI=http://dx.doi.org/10.1145/1985793.1986036;

- Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, Hong Zhu, An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software, Volume 86, Issue 8, August 2013, Pages 1978-2001, ISSN 0164-1212, http://dx.doi.org/10.1016/j.jss.2013.02.061;

- SENE, R. P. de. Processo, qualidade e métricas de desenvolvimento de software: Fatores de sucesso. Engenharia de Software Magazine, p. 44–48, 2012. ISSN 1983127-7.
- Sergio Segura, Robert M. Hierons, David Benavides, Antonio Ruiz-Cortés, Mutation testing on an object-oriented framework: An experience report, Information and Software Technology, Volume 53, Issue 10, October 2011, Pages 1124-1136, ISSN 0950-5849, http://dx.doi.org/10.1016/j.infsof.2011.03.006;
- Shivanand M. Handigund, A Suigeneris Automated Software Testing Methodology, Procedia Computer Science, Volume 62, 2015, Pages 21-22, ISSN 1877-0509, http://dx.doi.org/10.1016/j.procs.2015.08.404;
- Sina Shamshiri. 2015. Automated unit test generation for evolving software. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 1038-1041;
- SOUZA, V. R. de; VALE, R. C.; ARAúJO, M. A. P. Teste funcional utilizando o abbot framework: Motivação + engenharia de software. Engenharia de Software Magazine, p. 51–56, 2008. ISSN 1983127-7.
- SPíNOLA, R. O. Geração de casos de testes: Cmmi. Engenharia de Software Magazine, p. 26–32, 2012. ISSN 1983127-7.
- SPíNOLA, R. O.; ARAúJO, M. A. P.; SPíNOLA, E. O. Editorial: Automação de testes. Engenharia de Software Magazine, p. 03, 2010. ISSN 1983127-7.
- T. D. Cao, P. Felix and R. Castanet, "WSOTF: An Automatic Testing Tool for Web Services Composition,"Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on, Barcelona, 2010, pp. 7-12. doi: 10.1109/ICIW.2010.9;
- T. Repasi, "Software testing - State of the art and current research challanges,"Applied Computational Intelligence and Informatics, 2009. SACI '09. 5th International Symposium on, Timisoara, 2009, pp. 47-50. doi: 10.1109/SACI.2009.5136289;
- T. Xie, "Improving Effectiveness of Automated Software Testing in the Absence of Specifications,"2006 22nd IEEE International Conference on Software Maintenance, Philadelphia, PA, 2006, pp. 355-359. doi: 10.1109/ICSM.2006.31;
- T. Xie, K. Taneja, S. Kale and D. Marinov, "Towards a Framework for Differential Unit Testing of Object-Oriented Programs,"Automation of Software Test , 2007. AST '07. Second International Workshop on, Minneapolis, MN, 2007, pp. 5-5. doi: 10.1109/AST.2007.15;
- T. Xie, N. Tillmann, J. de Halleux and W. Schulte, "Mutation Analysis of Parameterized Unit Tests,"Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on, Denver, CO, 2009, pp. 177-181. doi: 10.1109/ICSTW.2009.43;
- TAVARES, J. F.; ARAúJO, M. A. P. Nunit: Testes unitários: Produtividade de times ágeis. Engenharia de Software Magazine, p. 33–41, 2011. ISSN 1983127-7.

- Thomas White. 2015. Increasing the efficiency of search-based unit test generation using parameter control. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 1042-1044;

- Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, Yue Wu, State of the art: Dynamic symbolic execution for automated test generation, Future Generation Computer Systems, Volume 29, Issue 7, September 2013, Pages 1758-1773, ISSN 0167-739X, http://dx.doi.org/10.1016/j.future.2012.02.006;

- Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. 2015. Making live programming practical by bridging the gap between trial-and-error development and unit testing. In Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2015). ACM, New York, NY, USA, 11-12.
DOI=http://dx.doi.org/10.1145/2814189.2814193;

- U. Rueda, T. E. J. Vos and I. S. W. B. Prasetya, "Unit Testing Tool Competition – Round Three,"Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on, Florence, 2015, pp. 19-24. doi: 10.1109/SBST.2015.12;

- V. Sethaput, K. Thorley, L. Zhou, B. Voldmane, A. Onart and F. Travostino, "A framework for automated unit testing of live network clouds,"Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on, Seattle, WA, 2001, pp. 579-592. doi: 10.1109/INM.2001.918067;

- Vahid Garousi Yusifoğlu, Yasaman Amannejad, Aysu Betin Can, Software test-code engineering: A systematic mapping, Information and Software Technology, Volume 58, February 2015, Pages 123-147, ISSN 0950-5849, http://dx.doi.org/10.1016/j.infsof.2014.06.009;

- Vanessa Peña Araya. 2011. Test blueprint: an effective visual support for test coverage. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 1140-1142. DOI=http://dx.doi.org/10.1145/1985793.1986022;

- Viera K. Proulx and Weston Jossey. 2009. Unit test support for Java via reflection and annotations. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09). ACM, New York, NY, USA, 49-56.
DOI=http://dx.doi.org/10.1145/1596655.1596663;

- Viera K. Proulx. 2012. Software testing (in Java) from the beginning. J. Comput. Sci. Coll. 27, 6 (June 2012), 7-9;

- William Pugh and Nathaniel Ayewah. 2007. Unit testing concurrent software. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07). ACM, New York, NY, USA, 513-516.
DOI=http://dx.doi.org/10.1145/1321631.1321722;

- Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04). ACM, New York, NY, USA, 432-448. DOI=http://dx.doi.org/10.1145/1028976.1029012;

- Y. Kim, Y. Kim, Taeksu Kim, Gunwoo Lee, Y. Jang and M. Kim, "Automated unit testing of large industrial embedded software using concolic testing,"Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, Silicon Valley, CA, 2013, pp. 519-528. doi: 10.1109/ASE.2013.6693109;

- Yong Lei and J. H. Andrews, "Minimization of randomized unit test cases,"16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), Chicago, IL, 2005, pp. 10 pp.-276. doi: 10.1109/ISSRE.2005.28;

- Z. Chaczko, R. Braun, L. Carrion and J. Dagher, "Design of unit testing using xUnit.net, "Information Technology Based Higher Education and Training (ITHET), 2014, York, 2014, pp. 1-9. doi: 10.1109/ITHET.2014.7155685;

- Z. Xu and J. Zhang, "A Test Data Generation Tool for Unit Testing of C Programs,"2006 Sixth International Conference on Quality Software (QSIC'06), Beijing, 2006, pp. 107-116. doi: 10.1109/QSIC.2006.7;

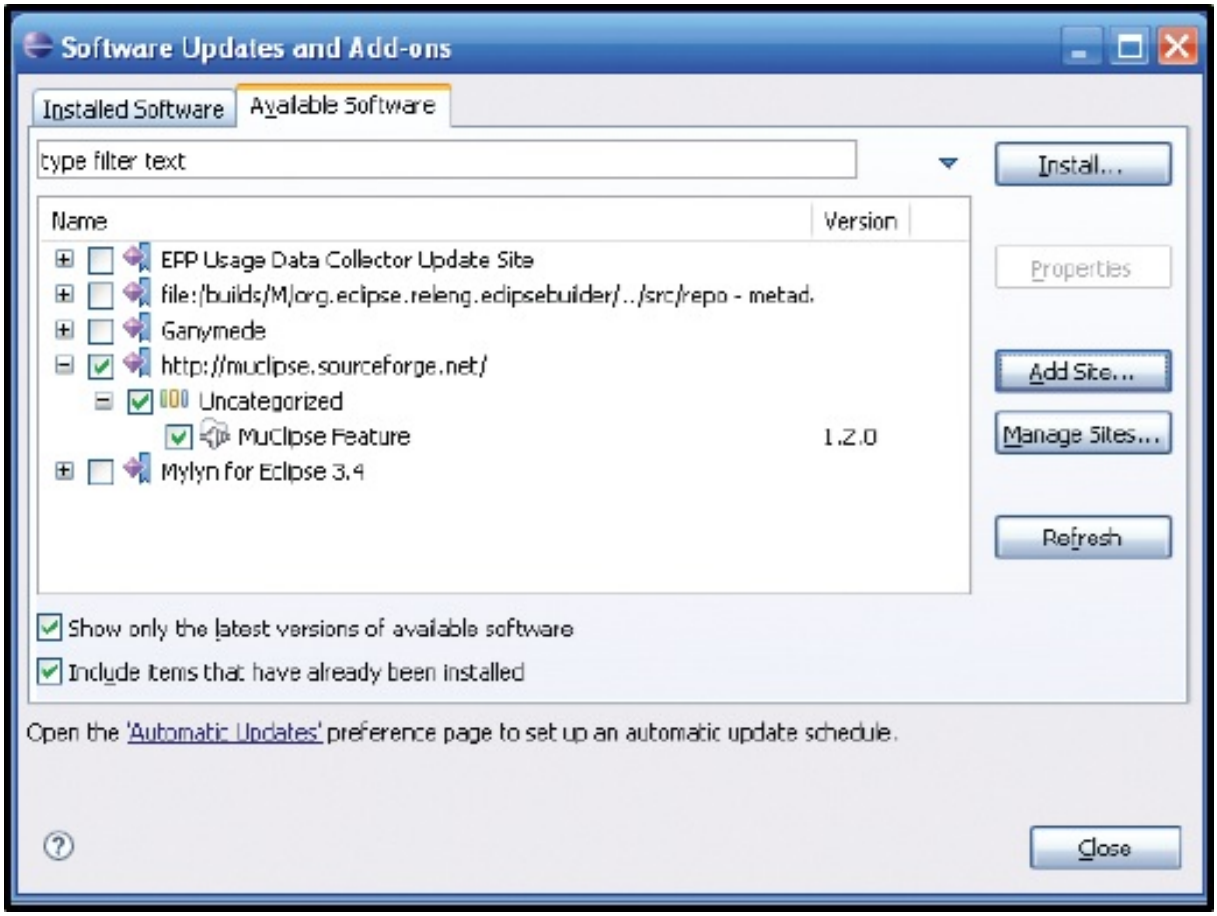
- Z. Zakaria, R. Atan, A. A. A. Ghani and N. F. M. Sani, "Unit Testing Approaches for BPEL: A Systematic Review,"2009 16th Asia-Pacific Software Engineering Conference, Penang, 2009, pp. 316-322. doi: 10.1109/APSEC.2009.72;

- Zhongjie Li, Wei Sun, Zhong Bo Jiang and Xin Zhang, "BPEL4WS unit testing: framework and implementation,"IEEE International Conference on Web Services (ICWS'05), 2005, pp. 103-110 vol.1. doi: 10.1109/ICWS.2005.31;

ANEXO A – CONFIGURAÇÃO DA FERRAMENTA MUCLIPSE

Para instala-lo é necessário abrir o Eclipse, acessar o menu *Help* / *Software Updates*, ir na aba *Available Software* e em seguida clicar no botão *Add Site*. Então é apresentada uma Janela em que se deve digitar http://muclipse.sourceforge.net/ no campo *Location* e acionar o botão *Ok*. Após isso, é necessário marcar o endereço informado no passo anterior e clicar no botão *Install*, como mostra a Figura 21 (RIBEIRO; ARAúJO, 2008).

Figura 21 – Instação MuClipse

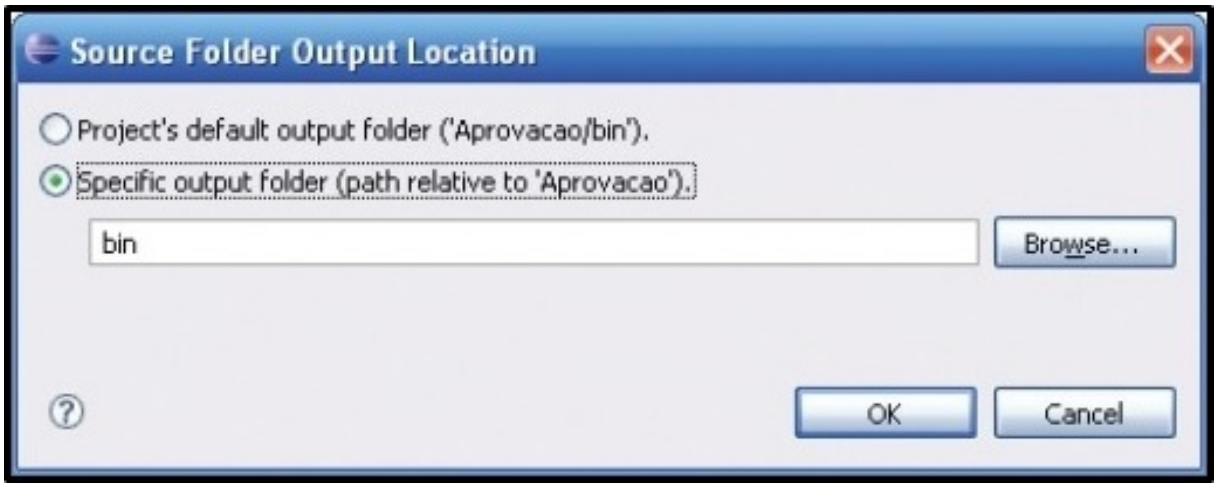


Fonte: (RIBEIRO; ARAúJO, 2008).

Para executar o teste de mutação utilizando o MuClipse, cria-se um projeto, que no exemplo tem o nome de Aprovacao, com uma estrutura composta por dois *Source Folders* chamados de src e testset, além da criação de dois *Folders* chamados de Lib e result. O MuClipse determina que o .class da aplicação não fique no mesmo diretório do .class dos testes, então é preciso configurar de onde os .class devem ser gerados. Para esse fim, clica-

se com o botão direito no src e depois selecionar a opção *Build Path* / *Configure Output Folder*. Em seguida surge uma janela em que a opção *Specific Output Folder* (path relative to 'Aprovacao') deve ser selecionado e depois digitar bin, de acordo com a Figura 22. A mesma coisa é feita para o *Source Folder* testset, porém a única diferença é o direcionamento do *Output Folder* para testset (RIBEIRO; ARAúJO, 2008).

Figura 22 – Configuração do *Output Folder*

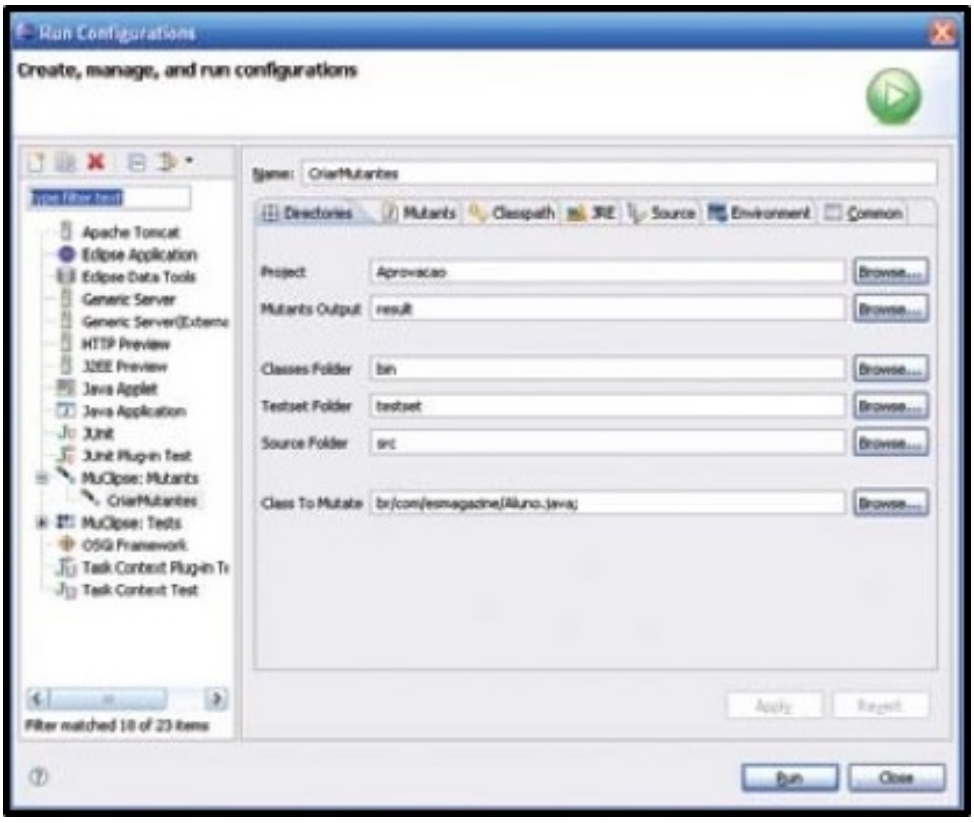


Fonte: (RIBEIRO; ARAúJO, 2008).

O MuClipse exige também a copia do arquivo extendedOJ.jar para a pasta lib e depois adiciona-lo ao *Path* da aplicação (clicando com o botão direito sobre o *Path* da aplicação e selecionando a opção *Build Path* / *Add to Build Path*). Este arquivo pode ser obtido através do *download* no site da ferramenta (RIBEIRO; ARAúJO, 2008).

Para acessar o MuClipse: *Mutants* (que proporciona a geração de mutantes) e o MuClipse: *Tests* (que executa os mutantes com os casos de teste), basta clicar com o botão direito no projeto e selecionando a opção *Run As* / *Run Configurations* é aberta uma janela de configurações do projeto, onde estão essas opções. Ao selecionar o MuClipse: *Mutants* e depois a opção *New* como mostra Figura 23, os mutantes podem ser configurados (RIBEIRO; ARAúJO, 2008).

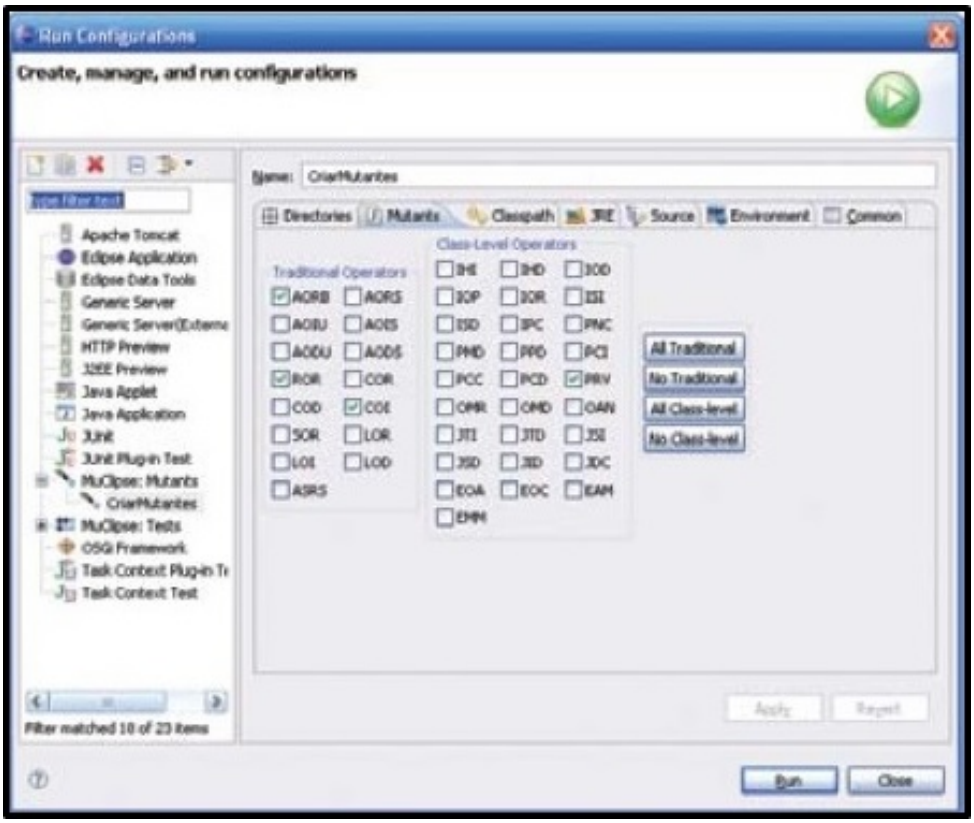
Figura 23 – Configuração dos mutantes



Fonte: (RIBEIRO; ARAúJO, 2008).

As opções a serem configuradas são, o nome do campo *Name*, o campo *Project* que deve ser preenchido com o nome do projeto, o campo *Mutants Output* é preenchido obrigatoriamente com o nome *result* para que o diretório onde os mutantes gerados possua esse nome e que não esteja no *Path* da aplicação. Os campos Classes *Folder*, *Testset Folder*, *Source Folder* devem indicar respectivamente, o diretório que possui os arquivos *bytecode*, o diretório que estão os casos de teste e o diretório que está à aplicação. Depois de escolher os operadores de mutação que serão usados através da aba *Mutants*, o botão *Run* deve ser clicado para gerar os mutantes como mostra a Figura 24 (RIBEIRO; ARAúJO, 2008).

Figura 24 – Operadores de mutação



Fonte: (RIBEIRO; ARAúJO, 2008).

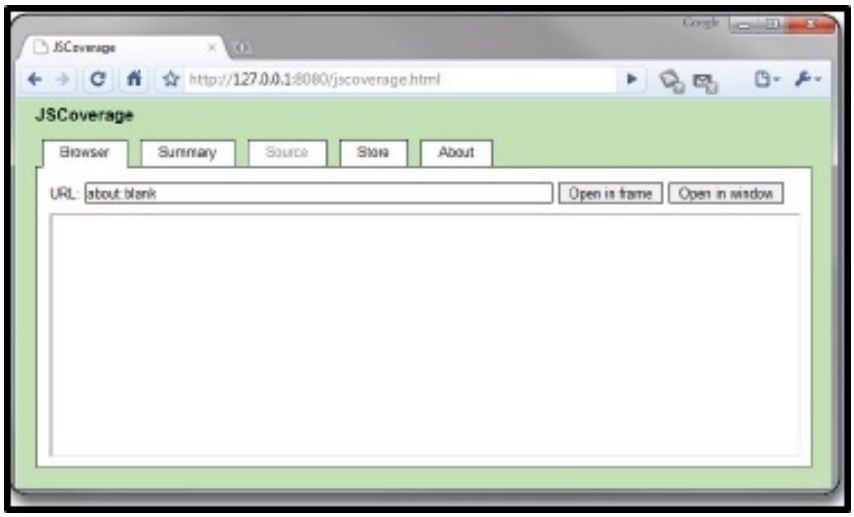
Para executar os casos de teste com os mutantes gerados é preciso configurar o MuClipse: *Tests*, preenchendo os campos *Name* e *Project*, colocando *result* no *Mutants Output*. O *Class Folder* deve-se informar o diretório bin e *Source Folder* o src, pois essas pastas possui respectivamente o arquivo *bytecode* e o código fonte da classe Aluno. O campo Testset *Folder* é preenchido com o nome do diretório onde fica os casos de teste, no caso o testset. Para o *Target Class*, informa-se o pacote e o nome da classe que será realizado os testes, já o campo *Test* é preenchido o pacote e o nome do método a ser testado. Ao realizar essas configurações e pressionando o botão *Run* inicia a execução dos mutantes e no final o *Console* apresenta além de outros assuntos o resumo do procedimento (RIBEIRO; ARAúJO, 2008).

ANEXO B – CONFIGURAÇÃO DA FERRAMENTA JSCOVERAGE

O JsCoverage está disponível em um pacote compactado (http://siliconforks.com/jscoverage/), contem a pasta principal com os executáveis jscoverage.exe e jscoverage-server.exe que são incumbidos por instrumentar as bibliotecas. E o diretório DOC, em que abriga a documentação e amostras do funcionamento da ferramenta. Ao analisar a cobertura o JsCoverage precisa instrumentar os códigos JavaScript, logo para executar essa ação existe duas formas (TOLEDO et al., 2010).

A primeira forma é usar o executável jscoverage.exe, bastando digitar o comando jscoverage.exe PASTA-FONTE PASTA-DESTINO dentro do *prompt* de comando do MS-DOS. O PASTA-FONTE é o diretório que possui todos os arquivos com extensão .js a serem instrumentados e o PASTA-DESTINO é o diretório destino dos códigos instrumentados. A segunda forma é executar o servidor *web* jscoverage-server.exe no *prompt* de comando do MS-DOS para que seja instanciado na memória. O servidor abrirá na porta 8080, seja qual for o navegador ele pode ser testado pelo endereço http://127.0.0.1:8080/jscoveragge.html. Esse link proporciona uma interface de instrumentação dos códigos JavaScript mostrada na Figura 25. Essa mesma interface é gerada na primeira forma quando se acessa o arquivo jscoverage.html na pasta destino. Para o correto funcionamento da segunda forma é necessário que o código fonte esteja no mesmo diretório do arquivo jscoverage-server (TOLEDO et al., 2010).

Figura 25 – Interface JsCoverage



Fonte: (TOLEDO et al., 2010).

Para integrar as ferramentas JsUnit e JsCoverage é necessário o *download* do JsUnit com a versão modificada para JsCoverage (http://www.jsunit.net/) e descompacta-lo em um

diretório. Após isso e com o caso de teste já criado utilizando as mesmas funções da família xUnit mas em linguagem JavaScript, basta executar o testRunner.html que está no diretório app da ferramenta (TOLEDO et al., 2010).

ANEXO C – CONFIGURAÇÃO DA FERRAMENTA EASY MOCK

Para a criação de objetos *mock* em um determinado projeto é necessário realizar o *download*, extrair os arquivos e adicionar aqueles com extensão “.jar” ao projeto. Além do mais é indispensável à instalação do JUnit com versão atualizado, pois o EasyMock depende deste *framework* para dar seguimento ao teste unitário (ARAúJO, 2008).