CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Jéssica Vieira Soares

FERRAMENTAS DE SUPORTE A AUTOMAÇÃO DE TESTE FUNCIONAL: LEVANTAMENTO BIBLIOGRÁFICO

Timóteo

2017

Jéssica Vieira Soares

FERRAMENTAS DE SUPORTE A AUTOMAÇÃO DE TESTE FUNCIONAL: LEVANTAMENTO BIBLIOGRÁFICO

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

Orientador: Marcelo de Sousa Balbino
Coorientadora: Deisymar Botega Tavares

Timóteo
2017

Jéssica Vieira Soares

Ferramentas de Suporte a Automação de Teste Funcional: Levantamento Bibliográfico

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

Orientador: Marcelo de Sousa Balbino
Coorientadora: Deisymar Botega Tavares

Marcelo de Sousa Balbino

Marcelo de Sousa Balbino
Orientador

Deisymar Botega Tavares

Deisymar Botega Tavares
Profª. Coorientadora

Márcia Valéria Rodrigues Ferreira

Márcia Valéria Rodrigues Ferreira
Profª. Convidada

Aléssio Miranda

Aléssio Miranda Júnior
Prof. Convidado

Timóteo, 05 de abril de 2017

Dedico aos meus pais,
que sempre me apoiaram e acreditaram em mim da forma mais pura e verdadeira.

Agradecimentos

Agradeço primeiramente Deus, por ter me dado forças nos momentos difíceis e pelo privilégio de crescer com uma família maravilhosa que sempre torceu por mim e nunca deixou de ditar palavras de incentivo quando me encontrava perante a um obstáculo.

Aos meus pais, Eni e Geraldo, que mesmo não compreendendo completamente o verdadeiro motivo de tantos estresses e preocupações, de passar aquelas noites em claro tentando finalizar aqueles trabalhos difíceis que pareciam não ter solução, sempre procuravam uma maneira de tentar resolver meus problemas, me dando forças, me apoiando e acreditando no meu sucesso da forma mais sincera.

Aos meus irmãos, Daiana e Joatan e, é claro, meu sobrinho Kayque, que mesmo com as desavenças do dia-a-dia nunca deixaram de me apoiar e me ajudar nos momentos que eu mais precisei.

Aos colegas de caminhada, que compartilharam comigo as mesmas preocupações. Devo a eles o companheirismo, a amizade, os momentos de risada e é claro a disposição para ajudar.

Aos meus professores por todo o conhecimento transmitido, especialmente a professora Márcia por ter me proporcionado novas oportunidades de aprendendizado.

E não posso deixar de agradecer o meu orientador Marcelo e minha coorientadora Deisymar que me auxiliaram até os últimos momentos deste trabalho, sempre me dando dicas e conselhos para melhorar.

Enfim, a todos os que estiveram comigo nessa longa jornada, o meu sincero Obrigado!

"O sucesso nasce do querer, da determinação e persistência em se chegar a um objetivo. Mesmo não atingindo o alvo, quem busca e vence obstáculos, no mínimo fará coisas admiráveis".

José de Alencar

Resumo

Devido a crescente evolução dos meios tecnológicos, é notável a demanda por softwares cada vez mais amplos e consequentemente mais propício a erros, e quanto mais prática e amigável a interface do usuário for, maior será sua aceitação. Em contrapartida, caso seja mal projetada ou apresentar defeitos, pode vir a ser um ponto crucial de rejeição. Dessa forma, os requisitos do sistema devem cumprir com as expectativas esperadas. Sendo assim, os testes funcionais, responsáveis por verificar se o sistema está de acordo com as especificações do cliente, são muito importantes para validação do sistema já que as exigências estão cada vez maiores. Técnicas e ferramentas de testes funcionais surgem no mercado para auxiliar na execução desses testes, de forma mais rápida e eficiente. Essas ferramentas, geralmente, simulam as ações de um usuário no sistema em teste e as transformam em *scripts* de teste que podem ser salvos e reexecutados sempre que necessário. Um problema enfrentado é o alto custo para a geração dos inúmeros casos de teste de entrada, necessários para se testar todas as funcionalidades do sistema. Embora os testes de software sejam grande aliados para a obtenção da qualidade, são poucos os documentos técnicos-científicos disponíveis acerca do tema. Geralmente, os livros de Engenharia de Software trazem uma breve descrição em uma seção ou capítulo de uma forma mais conceitual. Por essa razão, o presente trabalho destina-se a um levantamento biliográfico nas principais bases de pesquisa na área tecnológica, ACM, Science Direct e IEEE, e na revista Engenharia de Software Magazine, com o intuito de verificar as principais técnicas e ferramentas de automação de testes funcionais utilizadas, visando contribuir com a área de desenvolvimento de teste de software e fornecendo um apoio para fontes de consultas acadêmicas. A revisão proporcionou um levantamento de diversas estratégias e técnicas utilizadas atualmente, incluindo tanto as ferramentas de execução automatizada de testes funcionais, como o Abbot Framework, Marathon e Selenium IDE, quanto a geração de casos de testes para auxiliar no uso dessas ferramentas, como o AutoBlackTest, LBTest, e MoMuT::UML, dentre outras.

Palavras-chave: testes funcionais, teste de caixa-preta, ferramentas de automação de teste, geração de casos de teste.

Abstract

Due to the increasing evolution of the technological means, the demand for increasingly broader software and consequently more error-prone is remarkable, and the more practical and user friendly the user interface, the greater its acceptance. On the other hand, if it is poorly designed or defective, it can be a crucial point of rejection. That way, system requirements must meet the expected expectations. Therefore, the functional tests, responsible for verifying that the system conforms to customer specifications, are very important for validation of the system since the requirements are increasing. Functional testing tools and techniques appear in the market to help in the execution of these tests, more quickly and efficiently. These tools generally simulate a user's actions on the system being tested and transform them into test scripts that can be saved and replayed whenever necessary. A problem faced is the high cost for generating the numerous input test cases required to test all system functionalities. Although software testing is a great ally for getting quality, few technical-scientific documents are available on the subject. Generally, the Software Engineering books bring a brief description in a section or chapter in a more conceptual way. The present work is intended for a biliographic survey in the main research bases in the technological area, ACM, Science Direct and IEEE, and in the magazine Engenharia de Software Magazine, in order to verify the main techniques and tools of automation of functional tests used , Aiming to expand the research focused on functional tests and contribute to the area of software testing development, providing support for sources of academic queries. For this reason, the review provided a survey of the main strategies and techniques currently used, including both the automated execution of functional tests tools, like the Abbot Framework, Marathon and Selenium IDE and the generation of test cases to assist in the use of these tools, like AutoBlackTest, LBTest, e MoMuT::UML, among others.

Keywords: functional testing, black box testing, test automation tools, test cases generation.

Lista de ilustrações

Figura 1 – Estratégia de teste 23
Figura 2 – Regra 10 de Myers 24
Figura 3 – Exemplo de dados de entrada contínuos – particionamento de equivalência 28
Figura 4 – Exemplo de entrada de dados simples - particionamento de equivalência 28
Figura 5 – Exemplo de dados de entrada contínuos – análise do valor limite 29
Figura 6 – Casos de teste NEEOCP para variáveis únicas. 37
Figura 7 – Casos de teste REEOCP para variáveis únicas. 37
Figura 8 – Diagrama da visão geral do Framework. 38
Figura 9 – Exemplo de organização de dados de teste. 40
Figura 10 – Exemplo de definição dos dados de teste. 41
Figura 11 – Interface Gráfica do LBTest. 42
Figura 12 – Arquitetura do MoMuT::UML. 43
Figura 13 – Etapas do Teste de Caixa-Preta baseado em Rede Neural (NNBBBT). 45
Figura 14 – Atribuição de peso. 46
Figura 15 – Estratégia de seleção de teste. 47
Figura 16 – Um schema XML e as instâncias XML derivadas de TAXI. 48
Figura 17 – Tela inicial do Costello - Abbot 1.0.2. 50
Figura 18 – Página de configuração do script. 51
Figura 19 – Ações capturadas. 52
Figura 20 – Assert incluído. 53
Figura 21 – Modificação realizada no código fonte. 53
Figura 22 – Tela do sistema sendo executada automaticamente. 54
Figura 23 – Erro no Script. 54
Figura 24 – Configurações básicas do projeto. 55
Figura 25 – Configurações principais do Sistema a ser testado. 56
Figura 26 – Tela de Class Path, adicionando libs. 56
Figura 27 – Tela principal do Marathon. 57
Figura 28 – Listagem 1 - Exemplo simples de navegação. 58
Figura 29 – Tela principal. 58
Figura 30 – Tela de Import. 59
Figura 31 – Listagem 2 - Mapeando os dados. 59
Figura 32 – Tela de Mapping. 60
Figura 33 – Tela de Export. 60
Figura 34 – Listagem 3 - Exportando os dados. 60
Figura 35 – Arquitetura do MobileTest. 61
Figura 36 – Exemplo de procedures no QFTest. 63

Figura 37 – Exemplo de uma Package no QF-Test. 64
Figura 38 – Exemplo de chamada de uma Procedure no QF-Test. 65
Figura 39 – Exemplo de lista de variáveis que foram atribuídas na execução em QFTest. 65
Figura 40 – Exemplo de variáveis em QF-Test. 66
Figura 41 – Exemplo de estrutura de controle Loop no QF-Test. 66
Figura 42 – Exemplo de Relatório gerado pelo QF-Test. 67
Figura 43 – Plugin Selenium IDE no Firefox. 68
Figura 44 – Tela inicial do Selenium IDE, com opção de gravação ativada. 69
Figura 45 – Roteiro de teste Selenium IDE 1.0.5. 69
Figura 46 – Roteiro de teste com valores inválidos. 70
Figura 47 – Roteiro de teste com valores válidos. 70
Figura 48 – Reprodução de teste com falha. 71
Figura 49 – Tela inicial ferramenta Hudson. 72
Figura 50 – Interface principal da ferramenta SoapUI. 73
Figura 51 – Criando um novo projeto no SoapUI. 73
Figura 52 – Exemplo de requisição. 74
Figura 53 – Criação do TestSuite. 75
Figura 54 – Asserts e configurações do caso de teste. 75
Figura 55 – Erro após execução do teste. 76
Figura 56 – Caso de teste executado com sucesso. 76
Figura 57 – Tela de detalhes do usuário. 77
Figura 58 – Tela de criação de novo projeto. 78
Figura 59 – Tela de Nova Baseline. 78
Figura 60 – Tela de novo plano de teste. 79
Figura 61 – Tela de especificação de requisitos. 79

Lista de Quadros

2.1 Fontes consultadas 18
2.2 String de busca - IEEE 19
2.3 Definição dos critérios de notas 21
3.1 Exemplo de uma tabela de decisão 30
3.2 Ferramentas de Automação de Testes de Software. 33
5.1 Resultado da busca “Black-box testing + Functional software testing”. 80
5.2 Quantidade de artigos por critério de notas nas bases científicas 81
5.3 Busca realizada na revista Engenharia de Software Magazine. 82
5.4 Técnicas e ferramentas para geração de casos de teste. 85
5.5 Ferramentas de Automação de Testes de Software. 86

Lista de siglas

CA – Celular Automata – Autômatos Celulares

CP – Category Partition – Partição de Categoria

EEOCP – Equivalence Even Odd Class Partitioning – Particionamento de Classe de Equivalência Par e Ímpar.

ERP – Enterprise Resource Planning – Planejamento de Recursos Empresariais.

GUI – Graphical User Interface – Interface Gráfica do Usuário.

LBT – Learning-Based Testing – Teste Baseado em Aprendizagem.

MBT – Model-Based Testing – Modelo Baseado em Teste.

MBMT – Model-Based Mutation Testing – Teste de Mutação Baseado em Modelo.

MoMuT – MOdel-based MUtation Testing (Ferramenta de Automação)

MSD – Module State Diagram – Diagrama de Estado do Módulo.

NNBBBT – Neural Network Based Black Box Testing – Teste de Caixa-Preta baseado em Rede Neural.

OOAO – Object Oriented Action System – Sistema de ação orientado a objetos.

PLTL – Propositional Linear Temporal Logic – Lógica Temporal Linear Proposicional.

QA – Quality Assurance – Garantia da Qualidade.

QTP – Quick Test Professional (Ferramenta de Automação).

RFT – Rational Functional Tester (Ferramenta de Automação).

SUT – System Under Test – Sistema em Teste.

TCG – Test Case Generation – Geração de Casos de Teste.

UAT – User Acceptance Tests – Testes de Aceitação do Usuário.

Sumário

1 INTRODUÇÃO 14
1.1 Objetivos 16
1.1.1 Questões de pesquisa 16
1.2 Organização das próximas seções 17

2 MATERIAIS E MÉTODOS 18
2.1 Levantamento bibliográfico 18
2.1.1 Processo de pesquisa 18
2.1.2 Critérios de inclusão e exclusão 19
2.1.3 Filtros de pesquisa 20

3 FUNDAMENTAÇÃO TEÓRICA 22
3.1 Teste de software 22
3.1.1 Importância do teste de software 23
3.2 Teste funcional 24
3.3 Técnicas e métodos de teste funcional 25
3.3.1 Particionamento de equivalência 27
3.3.2 Análise do valor limite 29
3.3.3 Tabela de decisão 29
3.4 Automatização de teste de software 30
3.4.1 Por que automatizar? 31
3.4.2 Técnicas de automatização de testes 31
3.4.3 Ferramentas de automatização de testes 32

4 TÉCNICAS E FERRAMENTAS DE APOIO AO TESTE FUNCIONAL 34
4.1 Técnicas e ferramentas para a geração de casos de teste 34
4.1.1 AutoBlackTest 35
4.1.2 EEOCP 36
4.1.3 Framework para geração de dados de teste a partir da especificação de negócios 38
4.1.4 LBTest 41
4.1.5 MoMuT::UML 43
4.1.6 NNBBBT 44
4.1.7 TAXI 45
4.1.8 Teste baseado em modelo para aplicação WEB 47
4.2 Ferramentas para Execução de Teste Funcional 49

4.2.1 Abbot Framework 49
4.2.2 Marathon 55
4.2.3 MobileTest 61
4.2.4 QF-Test 63
4.2.5 Selenium IDE 67
4.2.6 SoapUI 72
4.2.7 TestLink 76

5 ANÁLISE E DISCUSSÃO DOS RESULTADOS 80
5.1 Resultados da pesquisa 80
5.2 Análise das pesquisas 83
5.3 Descrição das técnicas e ferramentas abordadas 85

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS 87

REFERÊNCIAS 89

APÊNDICE A – ARTIGOS DESCARTADOS APÓS FILTROS DE SELEÇÃO 92

APÊNDICE B – ARTIGOS DESCARTADOS APÓS CRITÉRIO DE NOTAS 100

1 Introdução

O teste de software torna-se uma prática cada vez mais comum e necessária, tendo em vista o aumento do nível de complexidade e eficiência dos softwares. A partir das constantes mudanças tecnológicas, surgem novas perspectivas acerca da maneira como os usuários passam a visualizar a interface dos sistemas e a interagir com elas. Um aparelho celular, por exemplo, que antes precisava-se acionar várias teclas numéricas para realizar uma única operação, hoje com a tecnologia de *touch screen* essa interação passou a ser mais rápida e prática, com operações que podem ser acionadas com apenas um toque ou até mesmo por um comando de voz.

Dessa forma, os desenvolvedores devem se esforçar para cumprir com as novas exigências do mercado em um curto espaço de tempo e com um custo satisfatório, mesmo às vezes não tendo em mãos a tecnologia necessária. É por isso que os sistemas atuais são muito propícios a erros, que se identificados de forma tardia podem custar um preço muito alto e quase inacessível para manutenção, além de causar a insatisfação dos clientes e então a falta de confiança dos mesmos com a empresa desenvolvedora.

Atualmente, existem inúmeras técnicas de teste de software para auxiliar na obtenção de um sistema com qualidade, que podem ser aplicadas desde o momento inicial de desenvolvimento até as etapas finais. Uma maneira de verificar defeitos do software é por meio da realização de testes de aceitação, que irá avaliar se o software está de acordo com os requisitos propostos e pronto para ser lançado para o ambiente de produção do usuário final (PRESSMAN, 2011). Uma das técnicas utilizadas nessa etapa de teste são os testes de caixa-preta, essa técnica não se preocupa com o código-fonte implementado, apenas com as funcionalidades do sistema, (NETO, 2010). De acordo com Pressman (2011), para que a validação tenha sucesso o sistema deverá funcionar de acordo com a expectativa do cliente. Para isso, é necessário o uso das especificações de requisitos do software, que descreve todas as funcionalidades existentes.

Entretanto, de acordo BENDER (1996 apud PINTO, 2013), o processo de testes é complexo e caro, podendo chegar a 50% de seu custo total de desenvolvimento quando realizado da forma tradicional. E se o mesmo for realizado de forma manual, o custo poderá ser ainda maior, pelo fato de ser tedioso e ineficiente (BENDER, 1996 apud PINTO, 2013). É a partir daí que começam a surgir os chamados testes automatizados, que segundo Caetano (2008) podem ser descritos como a aplicação de estratégias e ferramentas tendo em vista a redução do envolvimento humano em atividades manuais repetitivas. Sommerville (2011) explica que os testes automatizados são geralmente mais rápidos, principalmente, quando se trata de testes de regressão responsáveis por reexecutar testes já realizados após novas alterações no programa.

A automação de testes torna-se tão eficiente e prática, que é notório seu crescimento nos últimos anos (CAETANO, 2008). Novas ferramentas de automatização de testes surgem no mercado para facilitar a vida dos programadores. Inúmeros trabalhos que envolvem a automatização de testes de unidade ou testes de caixa-branca, aqueles que abordam a estrutura interna do programa, já vem sendo desenvolvidos ao longo dos anos. Até a década de 90, por exemplo, uma atividade comum e praticada era a criação de uma função de teste em cada módulo ou classe de um sistema com algumas simulações do uso da unidade (BERNARDO; KON, 2008). Embora pouco eficiente, já se tinha em mente a necessidade de se realizar testes automatizados para as funcionalidades internas do sistema. Já as pesquisas focadas na automatização de testes funcionais, embora, estejam em constante crescimento nos últimos anos, é bem mais recente que as pesquisas referentes aos testes de unidade. Segundo o trecho abaixo escrito por Crosby próximo a década de 80, citado por Pinto (2013), naquela época já existiam as automações de teste, mas poucos tinham foco direcionado a automatização de testes funcionais.

> "Diversos trabalhos, [...] têm realizado esta automação usando diferentes abordagens, tentando contribuir para a redução dos custos de desenvolvimento e a melhoria da qualidade do software construído. Poucos, porém, se concentram na verificação da adequação do software em relação aos seus requisitos, que é uma das principais formas de atestar sua qualidade (CROSBY, 1979). Fazendo uso dos requisitos, pode-se diminuir a explosão combinatória de possíveis caminhos percorríveis no uso do software, diminuindo o tempo e os recursos necessários para a geração dos testes, além de se concentrar nos casos importantes para seu maior interessado, o usuário."(PINTO, 2013)

Essa preocupação em relação à automação de testes funcionais, se dá a constante evolução dos softwares, que estão cada vez mais interativos e mais robustos. A interface está cada vez mais evoluída para facilitar seu uso por parte do usuário e assim a quantidade de funcionalidades tendem a aumentar. Por isso, é importante se obter ferramentas que proporcionem a realização de testes eficazes para garantir o funcionamento correto da aplicação final e obter a aprovação do cliente. Entretanto, de acordo com Matieli e Araújo (2015), são poucos os documentos técnicos-científicos voltados para os testes de softwares, fragilizando às organizações e profissionais que atuam nessa área.

Segundo Neto (2008) nos livros tradicionais de Engenharia de Software, costuma ser encontrado um capítulo ou seção com uma breve descrição sobre os testes de software, sempre de forma básica e conceitual. São apresentadas informações como os diferentes tipos de testes existentes, as técnicas de teste que podem ser aplicadas ou quais os critérios para a seleção dos testes (ex.: particionamento de classe de equivalência, análise do valor limite ou tabela de decisão). Mas, quando se trata do desenvolvimento de uma aplicação real, essas informações apresentadas não são o suficiente para serem aplicadas em um ambiente complexo de teste e que envolve grandes cenários. Por isso, é sempre importante

se ter um documento de apoio a futuros profissionais da área, com informações relevantes ao tema.

1.1 Objetivos

Tendo em vista os pontos acima levantados, deseja-se, com este trabalho, realizar um levantamento bibliográfico para verificar a existência de ferramentas de teste funcional ou teste de caixa-preta. E a partir dos resultados adquiridos, fazer uma análise dessas ferramentas, com o intuito de ampliar a pesquisa voltada para o mercado de testes funcionais e contribuir com a área de desenvolvimento de teste de software, fornecendo um apoio para fontes de consultas acadêmicas.

Como objetivos específicos, o trabalho se propõem a fazer uma análise sobre:

- Automação de testes funcionais;
- Técnicas de testes funcionais;
- Ferramentas de automatização para testes funcionais.

1.1.1 Questões de pesquisa

A partir do propósito deste trabalho, foi feito um levantamento com as principais questões de pesquisa a serem respondidas, de acordo com a perspectiva acadêmica. As mesmas são apresentadas a seguir:

- QP1: Como os testes funcionais podem contribuir para obtenção de um software com qualidade?
- QP2: Como o teste funcional pode ser executado?
- QP3: Quais são as ferramentas de automação existentes para esse tipo de teste e quais suas características?
- QP4: Quais são as vantagens de se utilizar tais ferramentas de automação?

A QP1 procura saber de que forma os testes funcionais podem auxiliar o desenvolvedor a obter um software com qualidade, contribuindo para a detecção de falhas. A QP2 visa obter o conhecimento sobre como os testes funcionais podem ser realizados. Se existe alguma técnica automatizada ou se geralmente é feito manualmente. A QP3 objetiva mostrar quais ferramentas automatizadas existem, caso sejam identificadas pela QP2. Por fim, na QP4, deseja-se saber as vantagens do uso dessas ferramentas para auxiliar o desenvolvedor.

1.2 Organização das próximas seções

Os capítulos seguintes estão divididos da seguinte forma: No capítulo 2 “Materiais e métodos”, é apresentada a metodologia abordada para a realização das pesquisas. No capítulo 3 “Fundamentação teórica”, são apresentados os principais conceitos referentes ao teste de software e algumas técnicas utilizadas nos testes funcionais, para auxiliar no entendimento do trabalho. No capítulo 4 “Técnicas e ferramentas de apoio ao teste funcional”, são descritas com mais detalhes, as ferramentas e técnicas de automação de teste encontradas nas pesquisas. No capítulo 5 “Análise dos resultados”, são apresentados os resultados obtidos pelas buscas realizadas e uma breve análise sobre as ferramentas encontradas no capítulo 4. E por fim, no capítulo 6 “Considerações finais e trabalhos futuros”, são apresentados as conclusões finais do trabalho e os trabalhos futuros.

2 Materiais e Métodos

Este trabalho consiste na realização de uma pesquisa exploratória, por meio de um estudo abrangente em três importantes bases de pesquisas científicas ACM (ACM Digital Li-brary), IEEE (IEEE Xplore Digital Library) e Science Direct e na revista Engenharia de Software Magazine. A revisão sistemática abordada foi baseada nos métodos utilizados pelo trabalho desenvolvido por Cota (2014). Na seção 2.1, é mostrado como foi estruturado o método de levantamento bibliográfico utilizado neste trabalho para se fazer uma revisão sistemática da literatura abordada.

2.1 Levantamento bibliográfico

A abordagem principal desse estudo foi baseada na coleta de dados de trabalhos científicos sobre o tema proposto. As informações obtidas foram de grande relevância para o enriquecimento do estudo, pois engloba uma gama de resultados científicos sobre o tema, expressando diferentes opiniões e conclusões.

2.1.1 Processo de pesquisa

As bases de dados ACM, IEEE e Science Direct foram escolhidas por serem da área das ciências exatas e tecnológica e por serem de grande referência na área de computação. Além dessas bases, foi utilizada como fonte de busca uma revista da área de engenharia de software brasileira, por conter um conteúdo mais técnico proporcionando obter informações mais detalhadas das ferramentas. Estavam disponíveis para análise as revistas de 1ª a 80ª edição. Portando, as edições posteriores não participaram da pesquisa. No quadro 2.1 são descritas as bases e revista consultadas.

Quadro 2.1 – Fontes consultadas

Fonte	**Acrônimo**
ACM Digital Library	ACM
IEEE Explorer	IEEE
Science Direct	SC
Engenharia de Software Magazine	-

Fonte: Elaborada pelo autor.

Inicialmente foram escolhidas as palavras-chave que seriam utilizadas nas buscas, de acordo com o objetivo e foco principais do trabalho. Como as bases são desenvolvidas no idioma inglês, as palavras-chave determinadas foram: *Black-Box testing*, *Functional*

Software Testing, *Functional Testing Automation*, *Open Source Testing Tools*, *Functional Testing Tools*, *Black-Box Testing Tools*. As mesmas foram utilizadas de forma dividida e não sistematizada, formando *strings* de buscas. Como foi obtido um resultado muito amplo, foi escolhida a *String* que obteve resultados mais relevantes ao tema para a seleção dos artigos.

Nas bases foi utilizado o recurso de "Pesquisa avançada", disponibilizado pelas mesmas, com o intuito de delimitar os campos de busca e obter resultados mais diretos e consistentes. Para isso, as *strings* de busca foram limitadas por título, palavras-chave, resumo e o ano de 2000 – atual. O ano 2000 foi escolhido como ano inicial, pois foi a partir deste ano que o número de artigos sobre a temática abordada começou a surgir com maior intensidade. Na base ACM, por exemplo, tem um campo sobre Refinar por Ano de Publicação, onde é possível visualizar um gráfico de linha que mostra esse crescimento. Já na base IEEE os primeiros artigos encontrados, se iniciaram a partir deste exato ano. Um exemplo de uma das strings de busca utilizadas na base IEEE, é mostrado no quadro 2.2 com as palavras-chave *Black-box testing* e *Functional software testing*.

Quadro 2.2 – String de busca - IEEE

((((("Document Title":"Black-box testing") OR "Abstract":"Black-box testing") OR "Author Keywords":"Black-box testing") OR "Document Title":"Functional software testing") OR"Abstract":"Functional software testing") OR "Author Keywords":"Functional software testing")

Fonte: Elaborada pelo autor.

As palavras-chave utilizadas na revista foram diferenciadas, pois não há uma ferramenta de busca avançada assim como nas bases científicas. Além disso, a revista Engenharia de Software Magazine é elaborada pelo idioma português. Assim, foram utilizadas as palavras-chave: "automação de testes funcionais", "teste de caixa-preta", "testes funcionais" e "ferramentas de testes de caixa-preta", de forma individual.

2.1.2 Critérios de inclusão e exclusão

Com o intuito de melhorar a eficiência da busca, reduzindo o número de resultados que fogem ao tema proposto, foram realizados os seguintes critérios de inclusão e exclusão:

Inclusão:

- Publicações a partir de 2000, quando a pesquisa por teste de software começou a se intensificar.

- Publicações em que os textos estejam disponíveis em sua íntegra e que possuam resumos para facilitar o filtro de pesquisa.

Exclusão:

- Estudos que não tenham relevância ao tema proposto.
- Estudos que tenham apenas fundamento didático.
- Estudos não disponibilizados gratuitamente ou não acessíveis via assinatura do portal Periódicos da Capes feita pelo CEFET-MG.

2.1.3 Filtros de pesquisa

Após a realização das consultas iniciais e obtenção dos resultados, foi aplicada uma técnica para a seleção dos artigos mais importantes e próximos ao tema abordado. Essa técnica baseia-se na aplicação dos três filtros a seguir:

- 1º filtro: fazer a leitura do título e/ou palavra chave de cada estudo retornado na busca inicial e verificar a sua aderência ao tema deste trabalho;
- 2º filtro: fazer a leitura do resumo de cada estudo selecionado no 1º filtro e verificar a sua aderência ao tema deste trabalho;
- 3º filtro: fazer a leitura da introdução e conclusão de cada estudo selecionado no 2º filtro e verificar sua aderência ao tema deste trabalho. Estes foram selecionados para leitura na íntegra.

A partir dos artigos selecionados pela etapa anterior, foi feita a leitura dos mesmos e uma análise da importância e qualidade de cada um. Isso gerou uma classificação com notas de 1 a 5, sendo de menor importância à maior importância, respectivamente. Essa classificação permitiu uma filtragem mais centrada dos resultados, com foco principal nos assuntos de maior relevância para o estudo. Com base nas leituras realizadas e visto o foco dos temas abordados pelos artigos, foi dada uma importância maior aqueles que tratam de ferramentas e técnicas voltadas para a geração de casos de teste e ferramentas de automação de testes funcionais. O quadro 2.3 mostra a definição de cada critério de nota.

O resultado obtido após a aplicação dos métodos apresentados, pode ser visualizado no capítulo 5.

Quadro 2.3 – Definição dos critérios de notas

Notas	**Definição**
1	Apresentam conteúdos pouco relevantes ou que fogem do foco principal do trabalho.
2	Apresentam algum conteúdo relacionado ao tema proposto, mas com pouco detalhe.
3	Apresentam conteúdos que abordam o tema, mas possuem foco pouco direcionado a ferramentas de automação.
4	Apresentam técnicas ou ferramentas de automação sobre o tema, mas com pouco detalhe ou que abrange outras áreas.
5	Apresentam técnicas ou ferramentas de automação relacionadas ao tema proposto e seu funcionamento.

Fonte: Elaborada pelo autor.

As ferramentas selecionadas a partir dos artigos das revistas, que eram gratuitas e de fácil acesso para download e instalação, foram testadas para um melhor entendimento do funcionamento das mesmas.

3 Fundamentação Teórica

Este capítulo aborda conceitos fundamentais sobre os testes de software, enfatizando os testes funcionais e a automatização de teste, para um melhor entendimento sobre os demais capítulos que compõem este trabalho.

3.1 Teste de software

O teste de software é um processo de detecção de riscos que pode ser executado em várias etapas do processo de desenvolvimento de um software (SOUZA; GASPAROTTO, 2013). De acordo com Pressman (2011), o teste de software faz parte de uma abordagem mais ampla da gestão de qualidade, que abrange um conjunto de estudos e práticas responsáveis por garantir a qualidade do software, não sendo o único método de garantir a segurança e qualidade de um sistema, mas essencial para se ter sucesso. Ele está incluso nos processos de verificação ("Estamos criando o produto corretamente?") e validação ("Estamos criando o produto certo?") BOEHM (1979 apud PRESSMAN, 2011), que compõem a garantia de qualidade de um software, assim como outras diversas atividades.

A verificação irá garantir que o software atenda as especificações dos requisitos corretamente, e a validação garantirá que o mesmo atenda as expectativas do cliente, de forma a estabelecer que o produto poderá ser entregue de acordo com as conformidades (SOMMERVILLE, 2011).

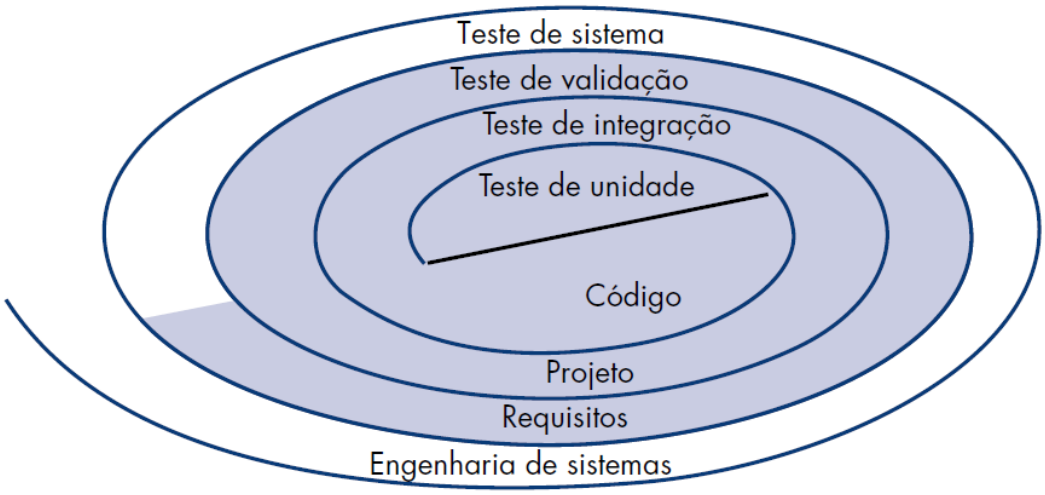
Antes de iniciar a fase de testes, é preciso planejar com antecedência um modelo de teste baseado nos requisitos e então executá-los.

> "Uma estratégia de teste de software deve acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem como testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente."(PRESSMAN, 2011, p.402)

Segundo Pressman (2011), muitas estratégias de teste já foram propostas na literatura. Uma estratégia é mostrada na Figura 1.

O modelo apresentado, conhecido como modelo espiral, é iniciado pelo centro, começando pelo teste de unidade, até a extremidade final, com o teste de sistema. O teste de unidade é responsável por garantir a funcionalidade de cada componente interno do sistema, cada componente será testado individualmente. O teste de integração irá proceder o teste de unidade, sendo que os componentes testados individualmente serão testados em conjunto. Em ambas as etapas é necessário que o testador tenha acesso e conhecimento do código fonte. Sendo geralmente executados pelos próprios desenvolvedores. O teste de validação está mais focado na parte funcional do software, seu objetivo é garantir que os

Figura 1 – Estratégia de teste



Fonte: Pressman (2011)

requisitos propostos estão de acordo com as expectativas do cliente. Nessa etapa não é necessário o conhecimento interno do código, podendo ser executada por um grupo independente de testadores. Por último a etapa de testes de sistemas compreende um contexto mais amplo, pois engloba não só a parte de software, mas também outros elementos do sistema como todo, hardware, banco de dados, capacidade de usuários, etc (PRESSMAN, 2011).

Todas as etapas do processo de testes são importantes para a obtenção de um sistema eficiente e que adquira a confiança do cliente. E cada sistema deve apresentar uma estratégia de teste que atenda ao seu propósito específico, uma estratégia elaborada para um grande projeto de software, por exemplo, não caberá a um projeto mais simples, com um número bem menor de funcionalidades.

3.1.1 Importância do teste de software

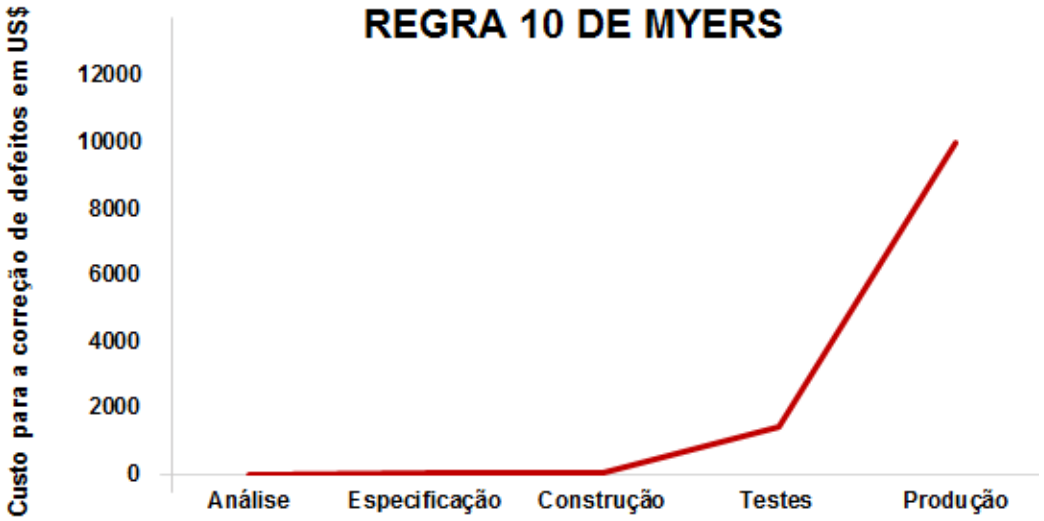
Todo produto criado para ser inserido no mercado precisa ser testado antes de ser entregue ao público-alvo final. Desde um simples brinquedo destinado ao público infantil, que deve passar por testes que garantam o cumprimento dos requisitos de segurança, até um automóvel que será destinado a um adulto, que precisa ter desempenho e segurança garantidos (GOMES, 2013).

Com sistemas de computadores não pode ser diferente. Por exemplo, se um sistema em tempo real do controle de voo falhasse ou um sistema de controle de insulina, que controla o nível de glicose no sangue, liberasse doses alteradas, os danos causados por essas falhas seriam enormes e a vida de pessoas correria risco. Todos os sistemas possuem sua finalidade e espera-se que ela seja atendida da melhor maneira possível. Não testar implica em supor que o sistema está ausente de erros e funciona de acordo com as especificações

propostas pelo cliente, mas isso não é o que acontece. Infelizmente, como os sistemas são abstratos e conceituais, falhas humanas podem ocorrer sendo os erros inevitáveis (WILLY ROBSON, 2008). Por isso, a execução de testes precisa ser realizada com o intuito de detectar essas possíveis falhas que possam comprometer o funcionamento do sistema.

Poucas empresas no Brasil costumam realizar algum tipo de teste. Segundo (MOLINARI, 2008), citado por (BERLATTO, 2012), de acordo com uma pesquisa realizada pelo Ministério da Ciência e Tecnologia, apenas cerca de 10% das empresas brasileiras realizam algum tipo de teste. Geralmente a barreira encontrada é que a realização de testes possui um custo muito alto, mas já é de conhecimento que um erro descoberto em produção torna-se muito maior do que se descoberto nas fases de implementação (GOMES, 2013). Como pode ser analisado, por exemplo, pela "Regra 10 de Myers", criada pelo cientista da computação Glenford Myers. Essa regra estabelece o conceito de que o custo de um erro cresce de forma exponencial conforme em qual fase é encontrado (SOUZA; GASPAROTTO, 2013). Na figura 2 é mostrado o custo médio em dólar de erro encontrado em cada etapa de desenvolvimento, de acordo com a regra de Myers.

Figura 2 – Regra 10 de Myers



Fonte: Adaptado de (SOUZA; GASPAROTTO, 2013)

A partir dessas observações, é possível supor que o investimento em testes é fundamental e não causa perda nos custos, pois futuramente ele será compensado.

3.2 Teste funcional

Os testes funcionais são também conhecidos como teste de caixa-preta, por se preocupar apenas com a estrutura externa do software, onde funciona como uma caixa

que recebe valores de entradas e retorna uma saída. Não é necessário ter o conhecimento do código-fonte para executá-lo (NETO, 2010).

Esse é o teste mais importante na etapa de validação do software, pois além de poder ser utilizado em quase todas as fases de desenvolvimento, é ele que dirá se toda a estrutura externa está funcionando corretamente, ou seja, se atende as exigências que foram definidas pelo usuário. Além disso, ele é independente do tipo de programação utilizada, que muitas vezes não está disponível para quem irá executá-lo. Se trata de verificar se a parte funcional, aquela que o usuário do sistema irá operar, satisfaz todos os requisitos pré-estabelecidos. Os erros que o teste funcional procura detectar estão incluídos em cinco categorias: (1) funções incorretas ou faltando, (2) erros de interface, (3) erros em estruturas de dados ou acesso a bases de dados externas, (4) erros de comportamento ou de desempenho, e (5) erros de inicialização e término (PRESSMAN, 2011).

O teste de caixa-preta nunca deve substituir o teste de caixa branca (testa a estrutura interna do software), ou os demais testes, pois cada um tem sua importância. Mas tem a finalidade de descobrir uma classe diferente dos erros referentes aos demais testes (WILLY ROBSON, 2008). Essa etapa seria como um complemento dos testes realizados dentro do código-fonte implementado, pois não basta apenas saber se as funcionalidades internas do código estão funcionando corretamente se a parte que o usuário realmente irá interagir não cumpre com suas exigências.

As controvérsias existentes a respeito desse tipo de teste é que, como seu conceito fundamental é realizar acesso a todas as funções existentes no programa, a tendência é testar cada tarefa possível que será executada pelo usuário e isso torna esse tipo teste muito caro e exaustivo para ser executado manualmente (NAUTIYAL; GUPTA; DIMRI, 2016). Existem, atualmente, algumas técnicas utilizadas para a redução da geração de casos de testes de entrada. Pois, com a crescente evolução dos softwares, o número de casos de testes possíveis está cada vez mais amplo, várias pesquisas para a geração de casos de teste automático vêm sendo propostas de forma a agilizar esse processo, reduzindo o nível de complexidade e o custo. Algumas dessas propostas serão apresentadas, na seção 4.1 sobre ferramentas e metodologias para a geração de casos de testes. Na seção 3.3, será apresentado com mais detalhes, como é o funcionamento do teste funcional, como ele é aplicado e quais os critérios existentes para reduzir o número de casos de testes.

3.3 Técnicas e métodos de teste funcional

Os testes funcionais desejam verificar a eficiência do produto final que será entregue ao cliente, da forma como ele verá e utilizará o sistema. Por isso a especificação de requisitos deve ser bem definida, pois ela será o principal meio para geração de requisitos dos testes (PRESSMAN, 2011).

Esse documento deve ser bem elaborado pela equipe de desenvolvimento, pois ele conduzirá o trabalho que será executado pelos testes funcionais. Muitas empresas falham ao realizar esse tipo de teste, justamente pela falta de uma documentação bem elaborada.

> "Se foi desenvolvida uma Especificação de Requisitos de Software, ela descreve todos os atributos do software visíveis ao usuário e contém uma seção denominada Critério de Validação que forma a base para uma abordagem de teste de validação." (PRESSMAN, 2011)

Pressman (2011) explica que para que os testes tenham sucesso é preciso elaborar um plano de teste que será seguido como um guia. Nele deve conter as classes e procedimentos de teste que definem os casos de testes gerados a partir dos requisitos funcionais, de modo que todos sejam devidamente satisfeitos. Existem dois resultados possíveis após a execução de cada caso de teste: 1) as funcionalidades estão de acordo com a especificação e é aceita; 2) existe algum desacordo com a especificação, então uma lista de deficiências é elaborada.

A seguir são mostrados os passos necessários para a execução dos testes funcionais, segundo o trabalho desenvolvido pela NAPSOL (2016) (Núcleo de Apoio à Pesquisa de Software Livre) – um projeto voltado para a automatização de testes de software, sem fins lucrativos, apoiado pela Pró-Reitoria de Pesquisa da Universidade de São Paulo e em particular, pelo CCSL - Centro de Competência em Software Livre:

- Analisar a especificação de requisitos;
- Escolher entradas válidas e inválidas de teste e verificar seu comportamento, de acordo com as especificações de requisitos;
- Determinar as saídas esperadas de acordo com as entradas escolhidas;
- Construir os casos de testes possíveis;
- Executar o conjunto de casos de testes;
- Comparar a saída esperada com a saída obtida;
- Criar um relatório com os resultados.

As etapas acima compõem toda a estrutura do teste funcional, do início ao fim. A dificuldade encontra-se nos inúmeros casos de teste existentes em um software, que pode gerar um alto nível de complexidade. Para facilitar a geração dos casos de teste de forma que não se torne tão exaustiva, existem algumas técnicas muito comuns e frequentes que são normalmente utilizadas. A escolha da técnica deve levar em conta as características do software que mais se adequam a ela. Dentre os critérios existentes estão (NAPSOL, 2016):

- Testes com base em grafos;
- Particionamento de equivalência;
- Análise do valor limite;
- Tabela de decisão;
- Testes de todos os pares;
- Testes de casos de uso.

Para um melhor entendimento de como funciona essa escolha dos casos de testes, serão apresentados nas seções 3.3.1, 3.3.2 e 3.3.3 os critérios de particionamento de equivalência, análise do valor limite e tabela de decisão e os seus respectivos funcionamentos.

3.3.1 Particionamento de equivalência

Esse tipo de critério divide determinados casos de testes de entrada em classes de testes, denominadas classes de equivalência. Isso porque, em determinadas situações, um caso de teste específico é suficiente para cobrir uma classe de erros. Por exemplo, supondo que a idade para contratação de um funcionário em uma empresa seja de 18 a 40 anos, para saber se um valor de teste de entrada é válido para este caso, basta inserir um valor que esteja dentro deste intervalo. Não é necessário realizar o teste para todos os valores de 18 a 40. Assim, ao invés de um dado caso de teste dentro de um intervalo ser executado para todos os valores possíveis, basta executá-lo apenas uma vez. Isso irá reduzir bastante o número de casos de teste. Essa é uma das técnicas mais comuns, que pode ser executada de forma intuitiva até mesmo sem o conhecimento prévio de seu funcionamento (NAPSOL, 2016).

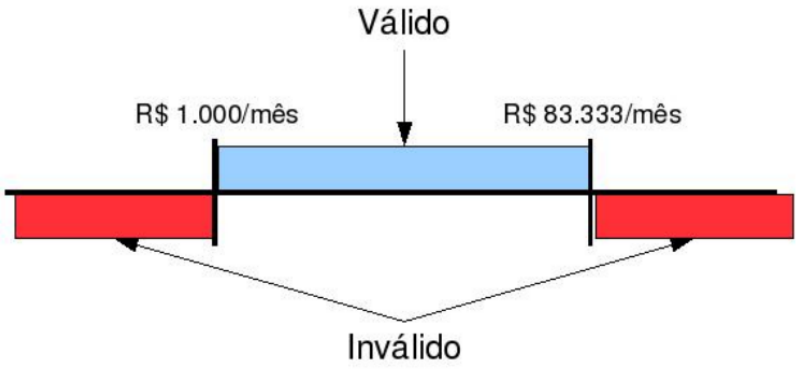
O particionamento das classes de equivalência depende muito de como são os dados de entrada, que pode ser um número, um intervalo de valores, dados verdadeiro ou falso ou um conjunto de valores relacionados (PRESSMAN, 2011).

1. Se uma condição de entrada especifica um intervalo, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
2. Se uma condição de entrada requer um valor específico, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
3. Se uma condição de entrada especifica um membro de um conjunto, são definidas uma classe de equivalência válida e uma classe de equivalência inválida.
4. Se uma condição de entrada for booleana, são definidas uma classe válida e uma inválida.

Os exemplos apresentados abaixo foram retirados da NAPSOL (2016). Por meio deles é possível perceber como o particionamento de equivalência é realizado de acordo com alguns conjuntos de dados de entrada.

- Intervalo de dados contínuos (renda para hipoteca de R$1.000 a 83.333/mês)

Figura 3 – Exemplo de dados de entrada contínuos – particionamento de equivalência

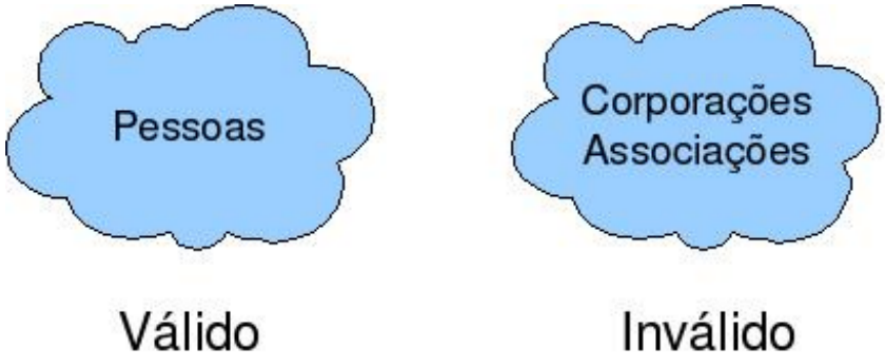


Fonte: (COPELAND, 2004 apud NAPSOL, 2016)

Neste caso são definidas duas classes inválidas e uma válida. Para a classe válida poderia ser escolhido R$1.342/mês. E para as classes inválidas poderia ser: R$123/mês e R$90.000/mês.

- Intervalo de dados simples (somente hipoteca para pessoas é permitido)

Figura 4 – Exemplo de entrada de dados simples - particionamento de equivalência



Fonte: (COPELAND, 2004 apud NAPSOL, 2016)

Neste caso são definidas uma classe inválida e uma válida. Para a classe válida poderia ser escolhida uma pessoa qualquer. Para a classe inválida deve ser escolhida uma companhia ou associação, como mostrado na figura 4.

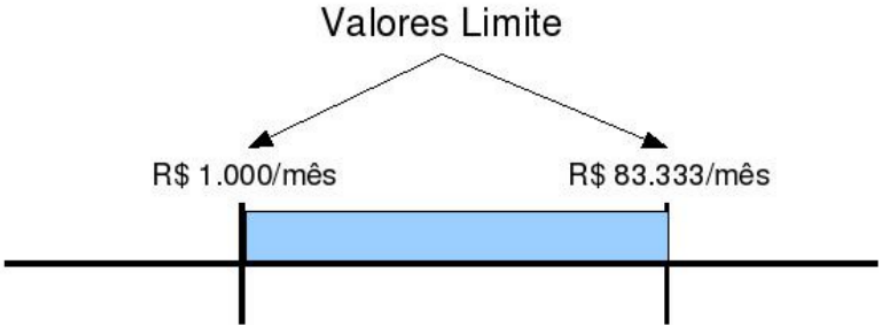
3.3.2 Análise do valor limite

Esta técnica se baseia em valores limites de entradas. Funciona como um complemento ao particionamento de equivalência, se diferencia por selecionar valores específicos próximos às extremidades de um intervalo, em vez de um valor qualquer. Foi criado para conter os erros ocorridos nas fronteiras, onde costumam aparecer com mais frequência. Seu uso é mais adequado para entradas que possuem valores contínuos.

Segundo a NAPSOL (2016), para a execução dessa técnica deve-se identificar as classes de equivalência, por meio da técnica anterior, e a seguir os limites de cada classe. O próximo passo é a criação de casos de teste para esses limites, que podem ser um valor anterior ao limite, o próprio limite e um valor após o limite. Dependendo dos recursos disponíveis podem ser criados outros casos de testes.

Para explicar como a seleção dos casos de teste funciona pela análise do valor limite, na Figura 5 é apresentado o mesmo exemplo utilizado acima para dados contínuos.

Figura 5 – Exemplo de dados de entrada contínuos – análise do valor limite



Fonte: (COPELAND, 2004 apud NAPSOL, 2016)

Dados de teste para o limite inferior: $999, $1.000, $1.001. Dados de teste para o limite superior: $83.332, $83.333, $83.334 (NAPSOL, 2016)

3.3.3 Tabela de decisão

A tabela de decisão define um conjunto de ações que serão executadas de acordo com um conjunto de decisões. Esse tipo de técnica é mais viável para geração de casos de testes mais complexos, que demandam um tempo muito grande de execução. Ela armazena as regras de negócio complexas que serão implementadas pelo sistema (NAPSOL, 2016).

Um exemplo de seu funcionamento é apresentado no Quadro 3.1.

O sistema deverá verificar se a pessoa informada é ou não maior de idade. Neste caso, a validação da entrada teria sete cenários possíveis.

Até o momento foram apresentadas algumas definições e métodos existentes dos testes de software com uma ênfase maior para os testes funcionais, que são o foco dessa

Quadro 3.1 – Exemplo de uma tabela de decisão

Condição	Cenário 1	Cenário 2	Cenário 3	Cenário 4	Cenário 5	Cenário 6	Cenário 7
Idade	<18	Igual 18	>18	0	Vazio	Negativo	Acima do valor No limite
Resultado Esperado	Menor	Maior	Maior	Inválido	Inválido	Inválido	Inválido

Fonte: (MALLLMANN, 2015).

pesquisa. A partir dos tópicos seguintes, serão apresentadas as técnicas de automação de testes existentes para agilizar os processos apresentados anteriormente.

3.4 Automatização de teste de software

Como a complexidade dos softwares aumenta de forma rápida, as técnicas manuais para execução de testes tornam-se muito trabalhosas e cada vez menos viáveis. Surge então, a necessidade de se criar meios que substituam o esforço manual e repetitivo do desenvolvedor. É por isso que a automatização de teste de softwares vem ganhando forças nos últimos anos. Segundo Caetano (2008), a automação de testes pode ser resumidamente descrita como a aplicação de estratégias e ferramentas tendo em vista a redução do envolvimento humano em atividades manuais repetitivas. As empresas estão cada vez mais adotando a prática de automação pelo menos em alguma etapa do processo de software. “Uma pesquisa realizada em 2006 pelo Forrester Research Inc, revela que 9% das empresas entrevistadas (empresas dos Estados Unidos e Europa) utilizam testes automatizados em todos os esforços de testes e 39% das empresas responderam que utilizam testes automatizados em alguns esforços de testes. ” Forrester Research Inc (2006 apud CAETANO, 2008).

Basicamente o que uma ferramenta de automação de testes faz, é transformar uma rotina de testes manuais em *Scripts* automáticos. Os testes de regressão são candidatos fortes a serem automatizados, pois são responsáveis por verificar se alterações realizadas em um sistema causaram algum efeito colateral negativo. Toda vez que uma modificação é feita, um conjunto de testes já executados e aprovados é reexecutado. A partir de uma ferramenta de automação é possível gravar esse conjunto de testes já realizado em forma de *scripts*, e quando necessário é só executá-lo novamente.

Nas próximas seções serão apresentadas as vantagens e desvantagens de se utilizar automatização de testes, assim como os tipos e algumas ferramentas existentes.

3.4.1 Por que automatizar?

Realizar tarefas exaustivas e repetitivas de forma manual demanda muito esforço e tempo de trabalho, além de ser suscetível a erro. Imagina ter que criar 10 mil tipos de séries de testes possíveis para testar as funcionalidades de um programa. Ou toda vez que seja feita uma alteração no sistema, o mesmo conjunto de testes já executado tenha de ser refeito para assegurar que as alterações não tenham causado algum tipo de erro. É para evitar esses tipos de esforços, que entram as ferramentas de automação de software. Dentre as principais vantagens dessas ferramentas estão (SOMMERVILLE, 2011):

- Execução de testes de forma mais rápida;
- Maior cobertura de testes;
- Menor índice de erros;
- Maior garantia de qualidade;
- Menor custo.

O papel da automatização de testes é executar de forma mais rápida e eficiente os processos de testes manuais, propondo uma maior cobertura de testes. Mas apesar dessas vantagens, é importante destacar que nem sempre seu uso leva ao sucesso. A automatização de testes também pode falhar se for utilizada sem o conhecimento prévio sobre funcionamento das ferramentas ou dos casos de uso corretos. É preciso conhecer as funcionalidades do software e saber qual tipo de teste será mais adequado ao propósito específico do que se deseja testar. Um planejamento deve ser feito com antecedência para saber o quê, como e por quê será automatizado (COLLINS; LOBAO, 2010a). Às vezes, pode ser preciso modificar algum *script* de teste que não foi gerado corretamente pela ferramenta, para que ele possa então ser reexecutado. Além disso, deve se levar em conta que os testes automatizados não cobrem todos os tipos de teste existentes. Nunca se deve descartar por completo o teste manual, alguns casos de teste só podem ser avaliados corretamente por pessoas, como detalhes visuais de interface gráfica ou testar se um programa não tenha efeito colateral. (NEGRINI, 2013).

3.4.2 Técnicas de automatização de testes

A execução de casos de testes automatizados pode ser dividida em duas abordagens distintas, apesar de não ser limitada a elas: baseadas na interface gráfica e baseadas na lógica de negócios (COLLINS; LOBAO, 2010a).

A primeira está focada na interação com a interface gráfica, simulando um usuário. Essa técnica utiliza o recurso de gravação e execução de casos de testes, conhecido

como *Record and Playback* ou *rec-and-play*. Ela irá gravar os casos de testes, que seriam as ações a serem executadas por um usuário, em um *script* que poderá ser executado posteriormente (COLLINS; LOBAO, 2010a).

As vantagens dessa técnica é que não são necessários ajustes de padronização de código para facilitar o teste (testabilidade), pois o teste é baseado apenas na interface gráfica do software. Entre as desvantagens está a alta dependência de uma interface gráfica estável, pois caso ela seja alterada o *script* de teste não funcionará mais (CAETANO, 2008).

Na segunda abordagem, baseada em lógica de negócios, o teste não irá interagir com a interface gráfica. Por meio dela poderão ser testadas partes pequenas ou grandes do código, como funções, métodos, componentes, dentre outros. (COLLINS; LOBAO, 2010a). As vantagens dessa técnica é a possibilidade de se focar na camada onde existe um maior índice de erros, agilizar testes que necessitam de centenas de repetição ou cálculos complexos, dentre outros. Como desvantagem está a necessidade de realizar alterações no código para padronizá-lo de forma a facilitar o teste e profissionais de programação especializados (CAETANO, 2008).

3.4.3 Ferramentas de automatização de testes

Atualmente, existem diversas ferramentas comerciais e *Open Source* para a realização de testes automáticos em várias fases de implementação, que estão disponibilizadas para uso. Desde os testes unitários, que visam o teste de cada unidade do sistema, até a geração de testes funcionais, que testam as funcionalidades do sistema. No quadro 3.2, são apresentadas algumas dessas ferramentas e as funcionalidades principais de cada uma delas.

As ferramentas Abbot Framework, Marathon, MobileTest, MoMuT::UML, Selenium IDE, SoapUI, TestLink e QF-Test, serão discutidas com mais detalhes no capítulo 4. As demais ferramentas não foram descritas no trabalho, seja por não entrarem nos objetivos principais do tema abordado, relativo aos testes funcionais, ou por não terem sido enfatizadas nos artigos selecionados de forma abrangente para o estudo.

Quadro 3.2 – Ferramentas de Automação de Testes de Software.

Ferramenta	**Licença**	**Técnica Utilizada**	**Funcionalidade**
FitNesse	*Open Source*	Lógica de Negócio	Teste de aceitação
JUnity	*Open Source*	Lógica de Negócio	Teste unitário Java
MbUnit	*Open Source*	Lógica de Negócio	Teste unitário .NET
Abbot Framework	*Open Source*	Interface gráfica	Teste de unidades GUI e testes funcionais
actiWATE	*Open Source*	Interface gráfica	Teste funcional para linguagem para aplicações WEB
Apadora	*Open Source*	Interface gráfica	Teste funcional para linguagem para aplicações WEB
BadBoy	*Open Source*	Interface Gráfica.	Teste funcional e performance para aplicações WEB
Canoo WEBTest	*Open Source*	Interface Gráfica.	Teste funcional para linguagem para aplicações WEB
IBM Rational Functional Tester	*Open Source*	Interface Gráfica.	Teste funcional e regressão
Mantis	*Open Source*	Interface Gráfica.	Registro e controle de defeitos
Marathon	*Open Source*	Interface Gráfica.	Teste funcional
MobileTest	Comercial	Interface Gráfica.	Teste funcional para aplicativos móveis
MoMuT::UML	*Open Source*	Interface Gráfica.	Geração de casos de teste
QF-Test	Comercial	Interface Gráfica.	Teste de regressão, carga e Aplicações WEB e Java
QuickTest Professional	Comercial	Interface Gráfica.	Teste funcional e regressão
Robot Framework	*Open Source*	Interface Gráfica.	Teste funcional
Selenium IDE	*Open Source*	Interface Gráfica.	Teste funcional para aplicações WEB - Ferramenta para navegadores
Selenium WebDriver	*Open Source*	Interface Gráfica.	Testes funcionais e de regressão para aplicações WEB - Framework
SoapUI	Versão *Open Source* e comercial	Interface Gráfica.	Testes funcionais, carga, segurança, performance e simulação de serviços em APIs.
TestLink	*Open Source*	Interface Gráfica.	Gestão de casos de teste
TestMaster	*Open Source*	Interface Gráfica.	Gestão de casos de teste
Watir	*Open Source*	Interface Gráfica.	Teste funcional para linguagem para aplicações WEB - gera scripts em linguagem RUBY
WinRanner	Comercial	Interface Gráfica.	Teste funcional

Fonte: Elaborada pelo autor.

4 Técnicas e ferramentas de apoio ao teste funcional

Até o momento foram apresentadas técnicas e ferramentas de automação de software existentes, de forma geral. As seções 4.1 e 4.2, visam explorar as ferramentas de automatização específicas para testes funcionais, que foram identificadas pelas pesquisas. Nem todas estão disponíveis para utilização, pois fazem parte do resultado de trabalhos acadêmicos e não possui fins comerciais.

Embora nas revistas científicas tenham sido encontrados diversos artigos sobre ferramentas de automação de testes funcionais, a automatização para execução de testes funcionais é pouco abordada pelos trabalhos encontrados nas bases científicas. Os artigos, encontrados nas bases, eram mais focados em ferramentas de automação e técnicas para facilitar a geração de casos de testes. Por essa razão, esta parte do trabalho foi dividida em duas seções, a seção 4.1 para relatar sobre as técnicas e ferramentas de geração de casos de teste encontradas e a 4.2 sobre o uso de ferramentas para a execução de teste funcional.

4.1 Técnicas e ferramentas para a geração de casos de teste

As ferramentas de automação de testes funcionais existentes, geralmente executam *scripts* de teste que representam as ações que seriam executadas pelo usuário ao utilizar o sistema (COLLINS; LOBAO, 2010a). Uma tarefa muito importante e que deve ser bem elaborada antes de utilizar essas ferramentas, é a geração dos casos de teste mais relevantes para o teste. O caso de teste nada mais é do que uma sequência de ações que representam uma funcionalidade específica do software, que compõem o *script* de teste. Acontece que devido ao aumento da complexidade dos softwares, com inúmeros cenários existentes, a elaboração manual dos casos de teste torna-se cada vez mais tediosa e demorada, como apontado pelos trabalhos a seguir.

Segundo Nautiyal, Gupta e Dimri (2016), a tendência do teste funcional é testar cada tarefa possível do software o que faz com que essa fase de teste seja muito cara. Deseja-se, então, planejar o menor número de casos de testes possíveis, de forma a cobrir todas as tarefas pedidas pelo usuário e obter um teste bem sucedido.

Torsel (2011) explica que a automação de testes de caixa-preta de aplicações web usando ferramentas de automação de teste comercial ou *OpenSource* como *selenium* ou *Canoo Webtest*, por exemplo, tem sofrido avanços significativos. Segundo o autor, essas ferramentas executam os casos de teste automaticamente, conduzido pela implementação

de *script*s que descreve a simulação da interação do usuário e a verificação de etapas. Um grande problema que continua não resolvido é como criar e manter estes casos de testes de uma forma eficiente, sendo que isso geralmente é feito manualmente pelo engenheiro de teste. O trabalho manual para geração de caso de teste pode não ser viável em cenários onde o modelo está sujeito a mudanças frequentes como nos processos de desenvolvimento ágeis, por exemplo. Por isso, surge a necessidade de empregar métodos automatizados para a seleção e geração dos melhores casos de teste.

De acordo com Wang, Yang e Zhu (2009), várias ferramentas são desenvolvidas para auxiliar os testadores como o Quick Test Professional (QTP), WinRunner, Rational Functional Tester (RFT) e Robot. No entanto, essas ferramentas estão focadas principalmente no design de etapa de teste na execução do teste, a geração automática dos dados de teste ainda está fora do alcance destas ferramentas. A geração de dados de teste seguindo determinados critérios, como o particionamento de equivalência e tabela de decisão, é uma tarefa demorada e muito difícil. Uma maneira para aliviar este processo é a geração automática de dados de teste.

Tendo em vista as observações levantadas acima sobre a necessidade da criação de métodos para geração de casos de testes automática, foi feito um levantamento a partir dos artigos selecionados pelas bases científicas, de algumas técnicas e ferramentas em fase de pesquisa, com o intuito de superar esses desafios.

4.1.1 AutoBlackTest

AutoBlackTest é uma ferramenta para a geração automática de casos de teste em aplicações interativas. O objetivo da ferramenta é gerar casos de testes que exerçam as funcionalidades do aplicativo em teste, interagindo com sua GUI (Interface Gráfica do Usuário). Para isso, o AutoBlackTest explora o ambiente desconhecido do aplicativo por meio de um agente, que "aprende"a melhor forma de agir, baseado na técnica de aprendizado de reforço. O agente pretende não apenas explorar as várias características do aplicativo em teste, como também identificar suas características mais significativas. (MARIANI et al., 2011).

O AutoBlackTest utiliza o *Q-learning*[1] *Agent*, um algortimo de aprendizado por reforço para construir de forma incremental um modelo de sequência dos eventos, que representa os passos que podem ser seguidos pelo aplicativo. Esse modelo constitui um conjunto de estados[2], que representam os estados da aplicação e um conjunto de transições

[1] "O algoritmo *Q-Learning* irá decidir entre explorar ou explotar através de uma política que permite escolher entre agir baseado na melhor informação de que o agente dispõe no momento ou agir para obter novas informações sobre o problema que possam permitir melhor desempenho no futuro. Neste contexto, o agente "aprende"quando consegue executar de forma mais eficiente determinada tarefa em função de experiências obtidas. (JÚNIOR, 2009)

[2] Os estados correspondem à forma como o agente percebe determinado aspecto do problema, enquanto

de estado, que serão acionadas mediante uma ação (MARIANI et al., 2011).

O *Q-learning*, utiliza uma estratégia para a escolha da melhor ação a ser executada através do maior valor de utilidade de estado, denominado por *Q-value* (JÚNIOR, 2009). A primeira vez que um agente executa uma ação em um dado estado, o *Q-value* do par (ação, estado) é o valor da recompensa retornada pela execução da ação. Enquanto o agente continua a interagir com o aplicativo executando novas ações ou respondendo ações conhecidas, os *Q-values* são atualizados iterativamente de acordo com a experiência adquirida durante o processo de aprendizagem. Esta característica do *Q-learning* é extremamente útil para orientar o agente para a re-execução e explorar profundamente os cenários mais relevantes (MARIANI et al., 2011).

O protótipo AutoBlackTest é integrado com as ferramentas IBM Rational Functional Tester (RFT) e TeachingBox. A primeira, é uma ferramenta automática de captura e repetição para testes de regressão. O AutoBlackTest usa a RFT para extrair a lista de *widgets* presentes em uma determinada GUI, para poder acessar o estado dos *widgets* e interagir com eles. A segunda ferramenta, TeachingBox, implementa o núcleo do algoritmo *Q-learning* que pode ser adaptado às necessidades específicas, implementando as funções de abstração de estado, as ações e a função de recompensa. AutoBlackTest também fornece a capacidade de gerar um conjunto de teste que pode ser automaticamente re-executado com a RFT a partir do modelo obtido como resultado da fase de aprendizagem (MARIANI et al., 2011).

AutoBlackTest foi testado no aplicativo *Twitthere*, usado para postar, modificar, excluir e ler *tweets* no *Twitter* de forma rápida. A fase de aprendizagem produziu um modelo com 462 estados e 1176 transições e foram descobertas 2 falhas no aplicativo. A ferramenta cobriu 71% das declarações, mostrando assim uma boa capacidade de execução do aplicativo. O conjunto de testes de regressão que o AutoBlackTest sintetizou a partir do modelo foi executado com sucesso e cobriu muitos cenários relevantes do ponto de vista do perfil de uso, incluindo numerosas combinações de múltiplas operações, como postar vários *tweets*, postar e excluir um *tweet*, etc (MARIANI et al., 2011).

4.1.2 EEOCP

Em Nautiyal, Gupta e Dimri (2016) foi proposta uma abordagem para reduzir o número de casos de teste utilizando o particionamento de classe de equivalência, uma das técnicas utilizadas para gerar casos de testes, (seção 3.3.1). A abordagem utiliza entradas numéricas pares e ímpares nomeada por EEOCP (Equivalence Even Odd Class Partitioning), sua função é minimizar o número de dados de entrada. São analisados conjuntos válidos e inválidos de valores de entrada para um caso de teste que pode ser

as ações correspondem as escolhas disponíveis ao agente, de forma que, a escolha de uma ação acarreta na mudança de um estado.(JÚNIOR, 2009)

par ou ímpar, um intervalo de valores numéricos ou qualquer condição que resulte em verdadeiro ou falso. Depois de identificado todas as partições, são selecionados casos de teste de cada partição.

Foram utilizados exemplos de valores de entrada únicos e valores de entrada múltiplos, onde o primeiro tem apenas uma variável de entrada dentro de um intervalo específico e o segundo pode ter duas ou mais variáveis de entrada dentro de um intervalo específico. Essa abordagem é uma variação do particionamento de classes de equivalência. A classe de equivalência classifica a entrada de um sistema em classes equivalentes. Assume-se que todos os elementos de uma classe são tratados da mesma maneira. Assim, a partir do particionamento de classes de equivalência, foram testados apenas um elemento de cada classe (NAUTIYAL; GUPTA; DIMRI, 2016).

A abordagem proposta fornece um procedimento sistemático para avaliar a confiabilidade do produto de software. O EEOCP funciona em sistemas para os quais a variável de entrada é de tipo inteiro. O intervalo de entrada é dividido em duas classes NEEOCP e REEOCP. A primeira classe abrange os valores de entrada que estão dentro do intervalo de entrada (Figura 6) e a segunda classe está preocupada com os valores de entrada inválidos, fora desse intervalo (Figura 7) (NAUTIYAL; GUPTA; DIMRI, 2016).

Figura 6 – Casos de teste NEEOCP para variáveis únicas.

Fonte: (NAUTIYAL; GUPTA; DIMRI, 2016)

Figura 7 – Casos de teste REEOCP para variáveis únicas.

Fonte: (NAUTIYAL; GUPTA; DIMRI, 2016)

Foi utilizado como exemplo, um programa que retorna o quadrado de um número dentro do intervalo de 1 a 100. Assim, na classe NEEOCP foi testado um valor de entrada par, 6, e um valor de entrada ímpar, 97, que estão dentro do intervalo. Já na classe REEOCP, foram testados quatro valores de entrada, sendo dois valores inferiores a 1 (par e ímpar) e dois valores superiores a 100 (par e ímpar), que estão fora do intervalo (NAUTIYAL; GUPTA; DIMRI, 2016).

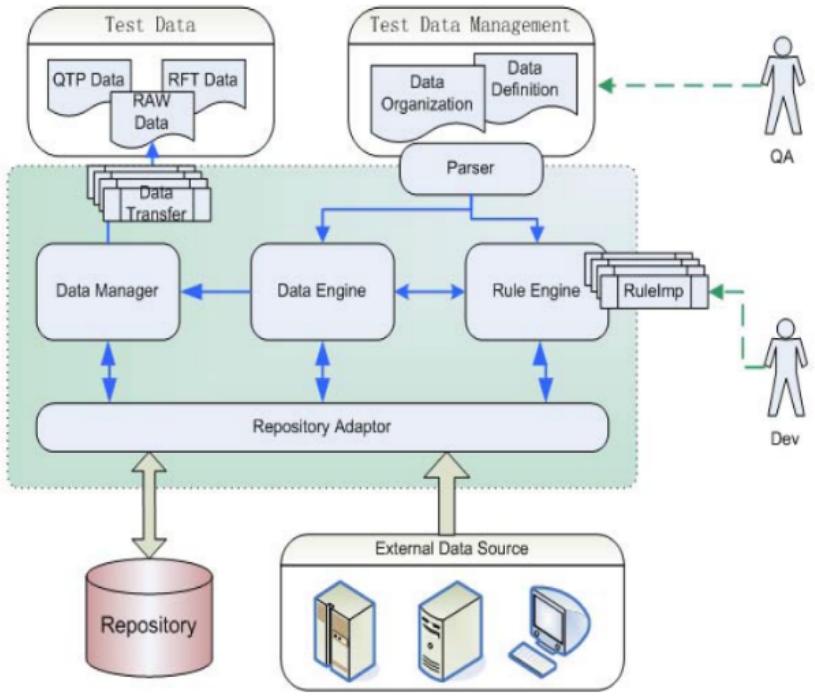
O método proposto foi testado e gerou dados de teste que cobrem quase todo o intervalo de entrada com menor número de casos de teste possível, sendo assim concluiu-se

que essa metodologia é eficaz e pode ser útil para minimizar o esforço ao criar esses casos de teste (NAUTIYAL; GUPTA; DIMRI, 2016).

4.1.3 Framework para geração de dados de teste a partir da especificação de negócios

No trabalho desenvolvido por Wang, Yang e Zhu (2009) é apresentado um Framework para a geração de dados de teste para teste de caixa-preta, a partir de uma especificação de negócios. Essa estrutura permite separar a definição de dados do programa em teste de sua implementação, concentrando o processo de teste apenas na parte externa do software. O núcleo do aplicativo apresentado é dividido em quatro componentes principais: mecanismo de dados (Data Engine), mecanismo de regras (Rule Engine), gerenciador de dados (Data Manager) e adaptador de repositório (Repository Adaptor), Figura 8.

Figura 8 – Diagrama da visão geral do Framework.



Fonte: (WANG; YANG; ZHU, 2009)

Essa estrutura permite ao usuário definir a lógica de negócios do sistema, que será introduzida para a geração dos dados de teste, e manter a organização dos dados armazenada para reuso. Ela funciona basicamente da seguinte forma: O mecanismo de dados é responsável por controlar o fluxo de trabalho principal da geração de dados de teste. Ele recebe as informações sobre a organização dos dados, e as informações sobre a construção dos dados são enviadas para o mecanismo de regras. Os dados de teste são

gerados pelo mecanismo de dados seguindo a instrução do mecanismo de regras e depois são passados ao gerenciador de dados (WANG; YANG; ZHU, 2009).

Quando o mecanismo de regra abstrai as definições de dados, ele as armazena no repositório juntamente com suas implementações. Ele atua como uma "base de regra"que pode ser estendida adicionando um novo *RuleImp* que implementa a lógica de negócios especificada. O *RuleImp* define um grupo de interface de programação que pode ser estendido pelo usuário (WANG; YANG; ZHU, 2009).

Todos os dados de teste gerados e as lógicas de negócios são armazenados no repositório de recursos eventualmente. O adaptador de repositório define formas comuns de acessar diferentes repositórios e fontes de dados externas. Através das interações entre os quatro componentes do framework, a lógica de negócios pode ser analisada e os dados de teste poderão ser gerados pela ”palavra-chave” correspondente (WANG; YANG; ZHU, 2009).

Segue abaixo os passos executados pelo Framework (WANG; YANG; ZHU, 2009):

- **Organização dos dados de teste:** Os dados de entrada são organizados pelo framework por uma hierarquia de árvore. A estrutura de organização padrão definida no framework possui três camadas, sendo a primeira a camada de aplicação, seguida da camada de suíte de teste e logo depois a camada de casos de teste. Para organizar bem esses dados de teste e evitar conflitos de nomes, cada dado é mapeado para um parâmetro.

- **Definição dos dados de teste:** Os dados de teste devem ser definidos de forma regulada para facilitar a análise por parte dos testadores e analistas de negócios. Para isso é usado o termo *"palavra-chave"*, que representa um dado de teste, como uma abordagem de *tradeoff*. Um parâmetro é usado para representar o dado de teste. Ele deve ser atribuído a um tipo para identificar sua representação de negócios. E algum valor de condição é usado como argumento de entrada do tipo de palavra-chave. O contexto é usado para especificar a restrição externa desse parâmetro. Portanto, a definição de uma entidade de dados de teste pode ser descrita como um vetor de quatro dimensões <Parâmetro, tipo, condição, contexto>.

- **Implementação da lógica de geração de dados:** Para implementar a lógica de geração de dados, é definido um conjunto de interfaces de programação denominado *RuleImp* .

- **Manutenção de dados de teste:** Os dados de teste gerados são armazenados no repositório pelo gerenciador de dados, para que possam ser reutilizados posteriormente. Como diferentes ferramentas de teste têm diferentes formatos e organização de dados de teste, o QTP e o RFT, por exemplo, têm seu próprio repositório de dados para suportar os testes de dados, é preciso que os dados de teste gerados

tenham o mesmo formato aceito por essas ferramentas. Para isso, o componente de dados (Data Transfer), figura diagramaframework, irá transferir os dados gerados para o formato de dados especificado. Novas transferências de dados podem ser adicionadas ao gerenciador de dados para que esta estrutura possa suportar diferentes ferramentas de teste.

Foi desenvolvido um projeto de implementação para experimento prático, por meio de um sistema de negociação de ações. Como o ambiente de teste é acoplado ao ambiente do produto, o banco de dados do aplicativo, que é a fonte dos dados de teste, é alterado por um processo diário. Portanto, os dados de teste precisam ser atualizados adequadamente.

Foi usado um arquivo *XML* como entrada para manter a hierarquia de dados de teste (Aplicação, suíte de teste e casos de teste). Na figura 9 é apresentado um exemplo da organização dos dados de teste desse exemplo. A entrada <BusiLogic> especifica o caminho do arquivo no qual a lógica dos dados de teste é definida.

Figura 9 – Exemplo de organização de dados de teste.

```
- <Application>
    <ID>StockTrading</ID>
  - <TestSuitList>
    - <TestSuit>
        <ID>TS01</ID>
      - <TestCaseList>
        - <TestCase>
            <ID>TC01</ID>
            <BusiLogic>OrderConstraint1.xml</BusiLogic>
          </TestCase>
        - <TestCase>
            <ID>TC02</ID>
            <BusiLogic>OrderConstraint2.xml</BusiLogic>
          </TestCase>
        + <TestCase>
        + <TestCase>
        </TestCaseList>
      </TestSuit>
    + <TestSuit>
    + <TestSuit>
    + <TestSuit>
    </TestSuitList>
  </Application>
```

Fonte: (WANG; YANG; ZHU, 2009)

Na Figura 10 é apresentado um exemplo da definição dos dados de teste com três parâmetros, e suas respectivas entradas: nome, tipo, condição e restrição.

A partir do framework, foi possível concluir que são necessários 8 dias por pessoa para desenvolver as palavras-chave e o design de dados de teste. Requer 1 dia por pessoa para atualizar manualmente os dados de teste em cada ciclo. Assim, o profissional QA(*Quality Assurance*) se beneficiará do uso desse framework, após 8 ciclos a partir da perspectiva de economia de esforço. O framework apresentado funcionaria bem com projetos de ciclo múltiplo, cuja lógica de negócios raramente é alterada. No entanto, a estabilidade da especificação do negócio e a complexidade do Sistema em Teste são considerações fundamentais antes de usar o framework (WANG; YANG; ZHU, 2009).

Figura 10 – Exemplo de definição dos dados de teste.

```
- <TestData>
  + <Init>
  - <ParameterList>
    - <Parameter>
        <Name>Symbol</Name>
        <Type>Equity.Stock</Type>
        <Context>Equity.MarketOrder</Context>
        <Condition>US</Condition>
      </Parameter>
    - <Parameter>
        <Name>Qty</Name>
        <Type>Math.Integer</Type>
        <Context>Equity.MarketOrder</Context>
        <Condition>"100-200"</Condition>
      </Parameter>
    - <Parameter>
        <Name>Account</Name>
        <Type>Equity.Account</Type>
        <Context>Equity.MarketOrder</Context>
        <Condition>TR</Condition>
      </Parameter>
    </ParameterList>
  </TestData>
```

Fonte: (WANG; YANG; ZHU, 2009)

4.1.4 LBTest

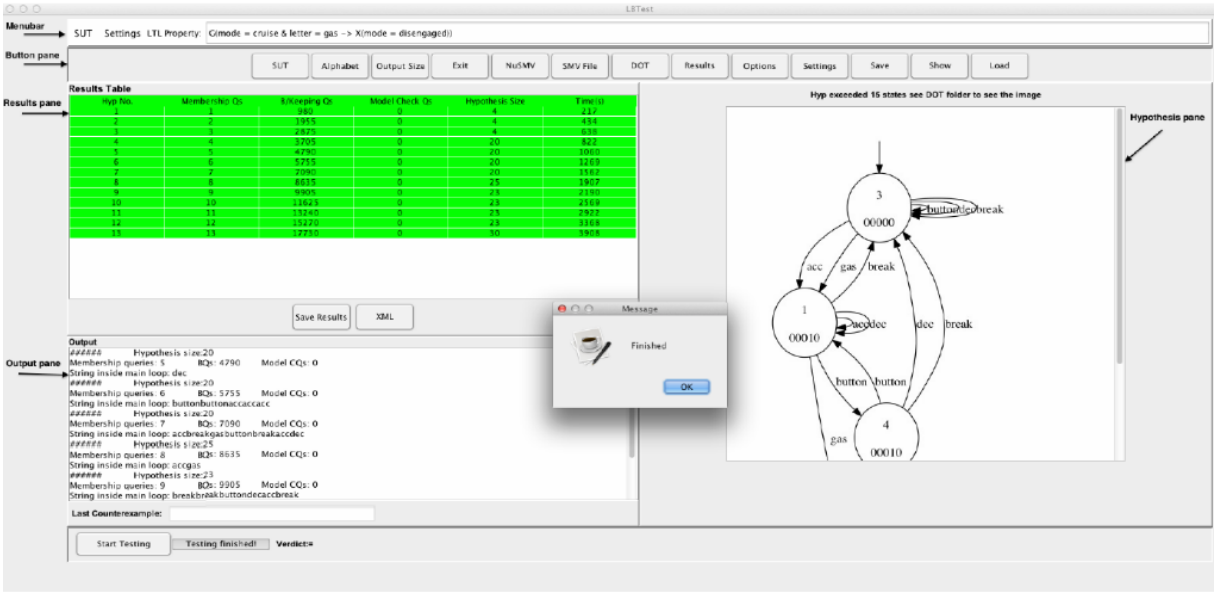
O LBTest é uma ferramenta baseada em algoritmos de aprendizagem para teste de sistemas reativos. As suas funcionalidades incluem a geração de casos de teste, a execução dos testes e a construção de oráculos de teste. A ideia básica da ferramenta é gerar automaticamente um grande número de casos de teste de alta qualidade combinando um algoritmo de verificação de modelo e um algoritmo de aprendizagem. Os dois algoritmos são integrados ao SUT (sistema em teste) em loop de realimentação iterativo, que irá otimizar a construção dos casos de teste, com base nos resultados anteriores. Os oráculos de testes são então construídos através da comparação entre as sequências de saídas previstas e das sequências de saídas observadas (MEINKE; SINDHU, 2013).

Um algoritmo LBT (Learning-based testing) é composto por três componentes:

1. Um sistema em teste de caixa-preta;
2. uma especificação formal de requerimento do usuário;
3. Um modelo de aprendizagem

(1) e (2) referem-se a entrada básica do algoritmo e também da ferramenta LBTest, e (3) trata-se da saída obtida por um algoritmo de aprendizagem. O teste baseado em aprendizagem é um método heurístico iterativo para gerar automaticamente uma seqüência de casos de teste. A interface gráfica da ferramenta pode ser visualizada na Figura 11 (MEINKE; SINDHU, 2013).

Figura 11 – Interface Gráfica do LBTest.



Fonte: (MEINKE; SINDHU, 2013)

O LBTest funciona, de forma genérica, a partir da seguinte sequência básica de passos (MEINKE; SINDHU, 2013):

1. **Carregar o sistema em teste**: Nessa primeira etapa, o usuário deverá carregar no LBTest o sistema a ser testado, a partir de um arquivo executável, como o arquivo .jar ou .exe, por exemplo.

2. **Definir a interface do SUT**: Nesta etapa o usuário deverá fornecer as interfaces de entrada e saída entre a ferramenta LBTest e o sistema a ser testado.

3. **Requisito de entrada**: Nesta etapa, os requisitos de entrada do sistema em teste devem ser fornecidos para o LBTest no formato PLTL (Lógica temporal linear proposicional), uma linguagem aceita pelo LBTest e que descreve o comportamento da interface do SUT.

4. **Executar**: Após as etapas anteriores e os dados informados, a ferramenta LBTest é então executada para encontrar qualquer violação que fora definida no passo 3, a partir do comportamento do SUT.

5. **Salvar/Mostrar Resultados**: Após a execução, a ferramenta LBTest irá salvar e mostrar os resultados obtidos a partir das informações dadas. Os resultados são salvos no formato *.xml* e podem ser acessados posteriormente.

Para mostrar o funcionamento da ferramenta, foi utilizado um estudo de caso pedagógico a partir de uma aplicação de controlador de cruzeiro, permitindo ao usuário

configurar e aprender a ferramenta rapidamente. Foi também fornecido um exercício de usabilidade para apoiar a avaliação da ferramenta descrita (MEINKE; SINDHU, 2013).

4.1.5 MoMuT::UML

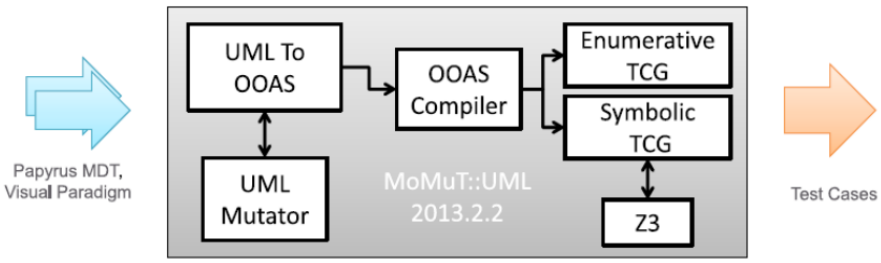
O MoMuT :: UML é uma ferramenta de automação para a geração de casos de teste, baseada na mutação de um modelo do SUT (sistema em teste) - MBMT (teste de mutação baseado em modelo) (AICHERNIG et al., 2015) .

O MoMuT::UML aborda o conjunto de duas técnicas. O teste de mutação, que tem o objetivo de injetar falhas no sistema em teste e verificar se os casos de teste conseguem identificá-lo e a modelagem do SUT, onde os casos de teste são gerados a partir de um modelo que descreve as caractéristicas do sistema de acordo com sua especificação.

A partir do conjunto dessas técnicas, o MBMT procura gerar automaticamente casos de teste que sejam capazes de revelar se qualquer falha modelada foi implementada. Para isto, o MBMT tomará o modelo original do SUT, aplicará um operador de mutação em um determinado local, derivando um mutante, comparará o comportamento do modelo original com o do mutante, e uma vez que uma diferença for encontrada escreve um caso de teste que dirige o SUT para esta diferença (AICHERNIG et al., 2015).

O MoMuT:: UML utiliza como entrada para a geração de casos de teste o diagrama de estado UML, o diagrama de classes e o diagrama de instâncias. A ferramenta também utiliza um pequeno perfil UML do SUT para saber quais eventos são provenientes do ambiente de teste e que são fornecidos pelo SUT em resposta. A Figura 12 apresenta a arquitetura da ferramenta (AICHERNIG et al., 2015).

Figura 12 – Arquitetura do MoMuT::UML.



Fonte: (AICHERNIG et al., 2015)

Essa arquitetura funciona basicamente da seguinte forma: Primeiro o MoMuT::UML irá interpretar os gráficos de estado UML e converter o modelo UML para um sistema de ação orientado a objetos OOAS (sistema de ação orientado a objeto). Em seguida ele irá preparar versões mutadas do modelo UML gerado, aplicando operadores de mutação selecionados pelo usuário para um conjunto de espaços de nome também selecionados pelo usuário. Os mutantes do modelo UML resultantes também são convertidos em OOAS.

Com os objetos OOAS gerados pelo modelo original e pelos mutantes, o MoMuT::UML estará pronto para gerar os casos de teste - TCG (Geração de casos de teste).

Como resultado, a ferramenta consegue uma cobertura de falha do modelo e o conjunto de testes resultante descobrindo se alguma das falhas modeladas foi implementada. Até o momento essa é única ferrramenta capaz de gerar casos de testes a partir de modelos de uma máquina de estado UML. O MoMuT é capaz e tem demonstrado produzir conjuntos de teste de alta qualidade e tem sido aplicado a uma série de estudos de casos de crescente complexidade. A ferramenta é disponibilizada gratuitamente, através do link <https://momut.org/?page_id=80,comaúltimavers~aoatualizada> (AICHERNIG et al., 2015).

4.1.6 NNBBBT

O trabalho proposto por Bhasin e Khanna (2014) tem como objetivo apresentar um sistema baseado em Redes Neurais que priorize os casos de teste de maior importância, tornando o teste de caixa-preta mais eficiente.

O sistema proposto, denominado Teste de Caixa-Preta Baseado em Rede Neural (NNBBBT), é dividido em cinco etapas (Figura 13).

1. **Gerador do diagrama do estado de módulo:** A primeira etapa é gerar os módulos de cada estado do sistema em teste a partir do Diagrama de Estado do Módulo (MSD), proposto em um dos trabalhos anteriores dos autores. Esse diagrama trata-se de um grafo em que cada nó representa uma função e as arestas entre os nós indicam a sequência de chamada. O MSD captura o fluxo da função de um programa e pode ser criado usando o rastreamento de pilha de um programa ou as especificações do Software.

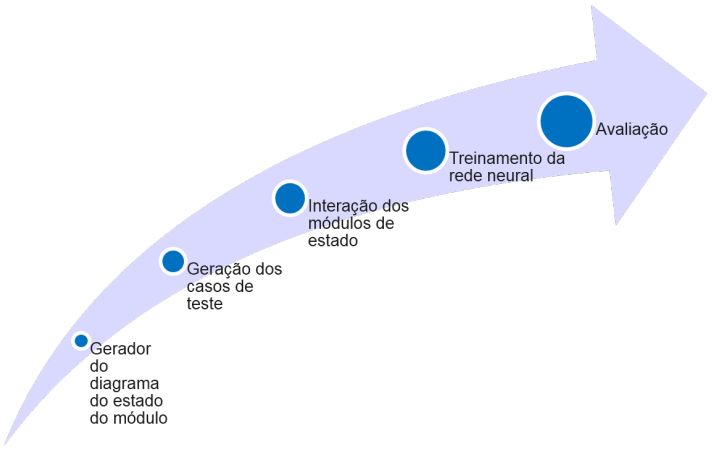
2. **Geração dos casos de teste:** A segunda etapa é a geração dos casos de teste. Isso é feito por um gerador de casos de teste baseado em autômatos celulares (CA), que também foi implementado e verificado em trabalhos anteriores dos autores.

3. **Interação dos módulos de entrada:** A terceira etapa trata-se das interações entre os módulos de estado encontradas pelo MSD. O módulo chamador passa argumentos para o módulo chamado e o módulo chamado retorna alguns dados para o módulo chamador. Essas interações de módulo são priorizadas por meio de atribuição de valores, de maior importância à menor importância, que são usadas para treinar a rede neural.

4. **Treinamento da rede neural:** A quarta etapa do sistema é o treinamento das redes neurais. As redes neurais são treinadas para priorizar as interações do módulo.

5. **Avaliação:** Na quinta e última etapa, a rede neural desenvolvida avalia a prioridade da interação com base no treinamento fornecido a ele na etapa 4.

Figura 13 – Etapas do Teste de Caixa-Preta baseado em Rede Neural (NNBBBT).



Fonte: Adaptado de ((BHASIN; KHANNA, 2014), **tradução nossa**)

Essa técnica apresentada foi testada em um sistema médio de Planejamento de Recursos Empresariais (ERP), e os resultados foram próximos às prioridades de casos de testes atribuídas manualmente. Segundo Bhasin e Khanna (2014), a técnica é um passo importante no sentido de desenvolver uma solução de teste completa para gerar casos de teste e priorizá-los.

4.1.7 TAXI

TAXI é uma ferramenta de automação para a geração de casos de teste. Essa ferramenta utiliza como entrada a linguagem em *Schema XML* e gera automaticamente um conjunto de instâncias em *XML*. Ela pode ser usada em testes de caixa-preta para aplicativos que aceitam como entrada instâncias *XML* e para *benchmarking* de sistemas de gerenciamento de banco de dados. (BERTOLINO et al., 2007).

TAXI é baseado na técnica de partição de categoria (CP), que fornece uma abordagem intuitiva, passo a passo, para testes funcionais da seguinte forma: 1) identificar os parâmetros de entrada relevantes; 2) Definir as condições ambientais; 3) Combinar seus valores significativos em um suíte de testes eficazes.

Para reduzir o número de combinações geradas pelo CP, TAXI também integra um conjunto de estratégias de teste ponderadas. O usuário pode fixar o número de instâncias geradas ou distribuí-las de acordo com fatores de importância definidos (pesos).

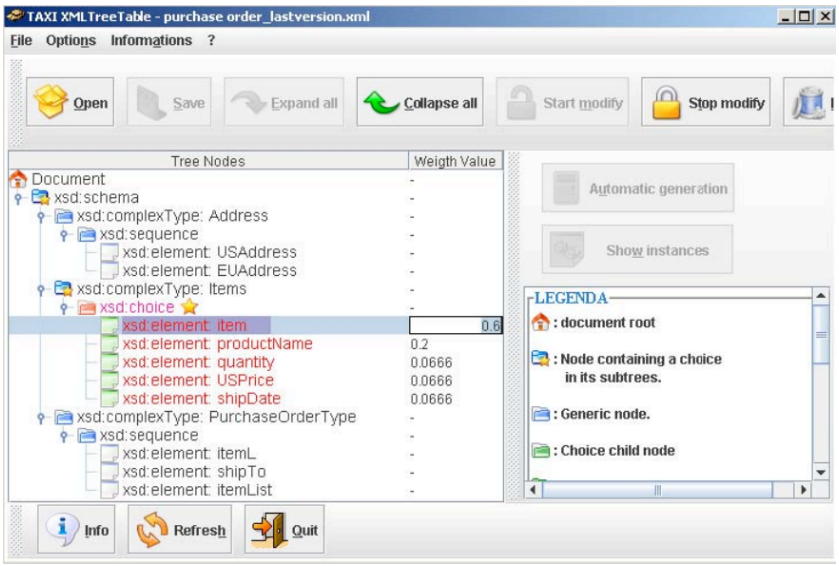
Segue abaixo os principais passos executados pelo TAXI:

- **Entrada *Schema XML*.** A interface do usuário lê a entrada em *Schema XML* e estabelece-a como uma estrutura em árvore, onde são distribuídos uniformemente

pesos padrão nos nós filhos de cada elemento de escolha. Eles pertencem ao intervalo [0,1] de modo que a soma dos pesos de cada elemento tem que ser 1.

- **Atribuição de peso.** O usuário pode modificar os pesos que foram atribuídos aos nós filhos de cada elemento de escolha, de acordo com a importância de cada um. TAXI verifica automaticamente se os pesos de cada escolha somam 1. A Figura 14 mostra a interface TAXI para a atribuição de peso.

Figura 14 – Atribuição de peso.



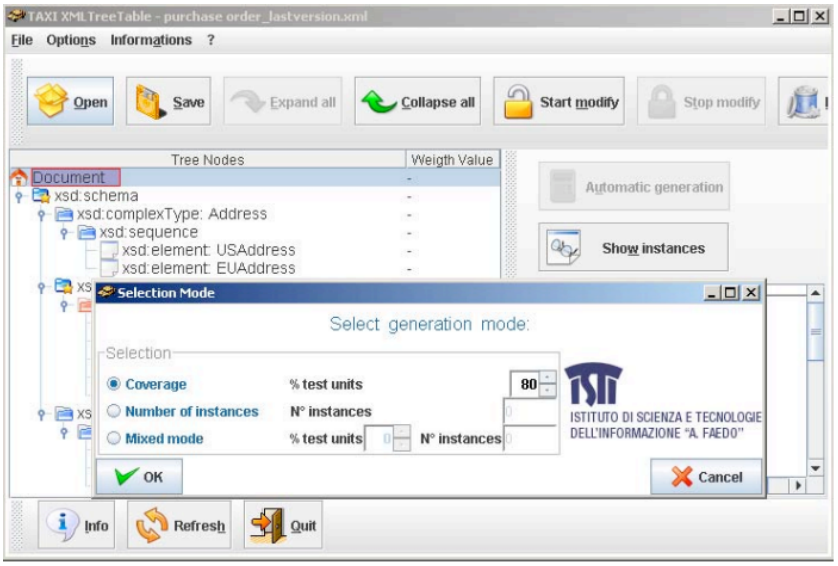
Fonte: (BERTOLINO et al., 2007)

- **População do banco de dados.** Existe um banco de dados contendo os valores para a derivação de instância, onde o usuário pode preenchê-lo de três formas: Inserindo os valores manualmente pelo TAXI, deixando o TAXI baixar esses valores de uma fonte específica ou deixando o TAXI recuperar esses valores das definições de entrada do *Schema XML*.

- **Análise de escolha.** TAXI extrai um conjunto de sub-Schemas, cada um contendo um diferente nó filho dos elementos de escolha. Ele automaticamente normaliza os pesos finais associados aos vários elementos do sub-Esquema.

- **Seleção de estratégias.** TAXI fornece três estratégias de teste: *Com um número fixo de instâncias*, que poderia ser na prática um conjunto finito de casos de teste que deve ser derivado. *Com uma cobertura funcional fixa*, quando uma determinada percentagem de cobertura de teste funcional (por exemplo, 80%) é estabelecida como um critério de saída para a geração de instâncias. *Com uma cobertura funcional fixa e número de instâncias*: as estratégias acima mencionadas são combinadas. A Figura 15 mostra a interface para a seleção da estratégia de teste.

Figura 15 – Estratégia de seleção de teste.



Fonte: (BERTOLINO et al., 2007)

- **Derivação de instância.** De acordo com a estratégia de teste adotada, TAXI gera automaticamente um conjunto de instâncias. Para isso TAXI fornece valores específicos para ocorrências de elemento e atribui valores do banco de dados aos elementos de sub-Schema. A Figura 16 mostra um Schema XML muito simples usado como entrada no TAXI e duas das instâncias que podem ser geradas pela ferramenta.

De acordo com os autores, as técnicas de teste de partição podem ser aplicadas naturalmente sobre o Schema XML, uma vez que fornece uma representação precisa do domínio de entrada em um formato adequado para processamento automatizado(BERTOLINO et al., 2007).

4.1.8 Teste baseado em modelo para aplicação WEB

Em Torsel (2011) foi apresentada uma abordagem de teste baseado em modelo para aplicações Web. O objetivo do projeto é gerar casos de teste a partir de um modelo, com um alto grau de automatização durante o processo, que possam ser executados em uma ferramenta de automação de teste existente. Para isto, os casos de testes gerados são transformados em *scripts* de teste específicos, de acordo com a linguagem aceita pela ferramenta de teste a ser utilizada, e são então executados e avaliados. O protótipo da pesquisa oferece modelos de transformação para as ferramentas Canoo Webtest e Selenium. Esse modelo baseia-se em trabalhos anteriores dos autores sobre testes baseados em modelos, com a inclusão do meta-modelo, responsável por permitir que oráculos de teste

Figura 16 – Um schema XML e as instâncias XML derivadas de TAXI.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseForm">
 <xsd:complexType>
 <xsd:all>
  <xsd:element name="Address">
   <xsd:complexType>
    <xsd:choice>
        <xsd:element name="USAddress"   type="xsd:string"/>
        <xsd:element name="EUAddress" type="xsd:string"/>
    </xsd:choice>
   </xsd:complexType>
   </xsd:element>
   <xsd:element name="PurchaseOrdre" type="PurchaseOrderType"/>
 </xsd:all>
 </xsd:complexType>
</xsd:element>
<xsd:complexType name="PurchaseOrderType">
<xsd:sequence>
 <xsd:element name="productName" type="xsd:string"/>
 <xsd:element name="quantity" type="xsd:positiveInteger"/>
 <xsd:element name="USPrice"    type="xsd:decimal"/>
 <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

a)

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseForm>
 <PurchaseOrdre>
   <productName>book</productName >
   <quantity>10</quantity>
   <USPrice>10.5</USPrice>
   <shipDate>2005-09-02</shipDate>
 </PurchaseOrdre>
 <Address>
   <USAddress>
     3269 Sherwood St. Montgomery
   </USAddress>
 </Address>
</purchaseForm>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseForm>
 <PurchaseOrdre>
   <productName>shoes</productName >
   <quantity>2</quantity>
   <USPrice>100</USPrice>
 </PurchaseOrdre>
 <Address>
   <USAddress>
     900 N Walnut Creek Dr. STE 100
   </USAddress>
 </Address>
</purchaseForm>
```

b)

Fonte: (BERTOLINO et al., 2007)

de uma página web sejam definidos automaticamente e transformados em um formato executável, durante a geração de casos de teste.

A ideia de utilizar um alto grau de automatização ao longo do processo de geração de modelos e geração de casos de testes visa reduzir o esforço de da realização de testes em aplicações Web. A abordagem proposta foi implementada em um protótipo com o auxílio das ferramentas *Xtext* e *Xpand* do Eclipse, que oferecem suporte para a edição do modelo. No estado atual apresentado, a abordagem proposta se adequa a geração de casos de teste automatizados para uma suíte de testes onde os cenários podem ser gerados a partir do modelo. Para cenários de teste mais intrincados, os suítes de teste são projetados manualmente pelo engenheiro de teste. Devido ao mecanismo de oráculos de teste, que se baseia na comparação de fragmentos de texto esperados e reais no código *html*, ele pode

ser usado para testar fluxos de trabalho que permitem tal observação do impacto do teste (TORSEL, 2011).

O protótipo oferece mais possibilidades de aprimoramento funcional. A realização de técnicas de decomposição hierárquica e horizontal para os modelos de aplicação beneficiaria a usabilidade da abordagem especialmente para modelos maiores. Além disso, no trabalho apresentado apenas os casos de teste positivos podem ser gerados a partir do modelo. Para melhorar a capacidade de encontrar bugs de aplicação, está sendo planejado o suporte para a geração automatizada de casos de teste negativos (TORSEL, 2011).

4.2 Ferramentas para Execução de Teste Funcional

Essa seção descreve as ferramentas de automação para execução dos testes funcionais, encontradas através das pesquisas, e seus respectivos funcionamentos. As ferramentas Abbot Framework, Marathon, Selenium IDE, SoapUI e TestLink foram baixadas e instaladas para teste, por estarem disponíveis gratuitamente para *download* e serem de fácil instalação. Os testes foram realizados para facilitar a compreensão sobre o funcionamento das mesmas.

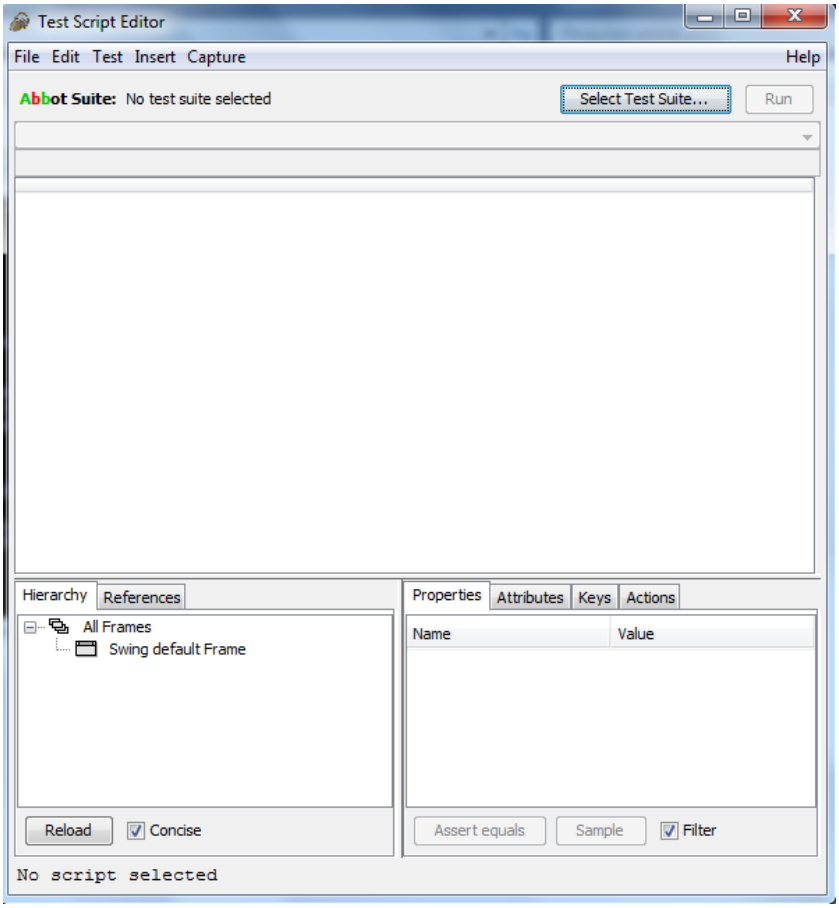
4.2.1 Abbot Framework

A ferramenta Abbot Java GUI Test Framework é capaz de realizar testes funcionais baseados em eventos. Ela nada mais é do que uma biblioteca Java para GUI (Interface Gráfica do Usuário), que abstrai componentes da GUI e realiza as ações que seriam executadas pelo usuário, fazendo comparações do resultado obtido com o resultado esperado. Nessa ferramenta, o aplicativo pode ser testado a partir do seu código Java ou por meio de *scripts* de testes que podem ser gerados de forma automática através do *Costello Script* para Abbot, um editor de *scripts* que vem incluído dentro do pacote Abbot Framework (SOUZA; VALE; ARAÚJO, 2008).

O editor Costello suporta a gravação de eventos que podem ser reproduzidos posteriormente para depuração. Os *scripts* gerados são salvos no disco em formato *XML* e são compostos por *actions*, *assertions* e referências de um componente. Os *actions* são os comandos executados pelo usuário, como clicar em um botão ou inserir um valor. Os *assertions* são comandos adicionados ao script de teste, para verificar o estado da GUI. Caso o assertion falhe, o *script* de teste irá parar a execução e notificar o erro (SOUZA; VALE; ARAÚJO, 2008). Na Figura 17 é apresentada a tela inicial do editor Costello da ferramenta Abbot Framework na versão 1.0.2.

Para iniciar um novo teste, seleciona-se a opção *file* na barra de menus do editor e cria-se um novo *script*, clicando em *New Script*. Após fazer isso, o programa cria automaticamente dois passos a serem seguidos, o *Launch*, responsável por carregar a classe e

Figura 17 – Tela inicial do Costello - Abbot 1.0.2.



Fonte: Adaptado de (SOUZA; VALE; ARAÚJO, 2008).

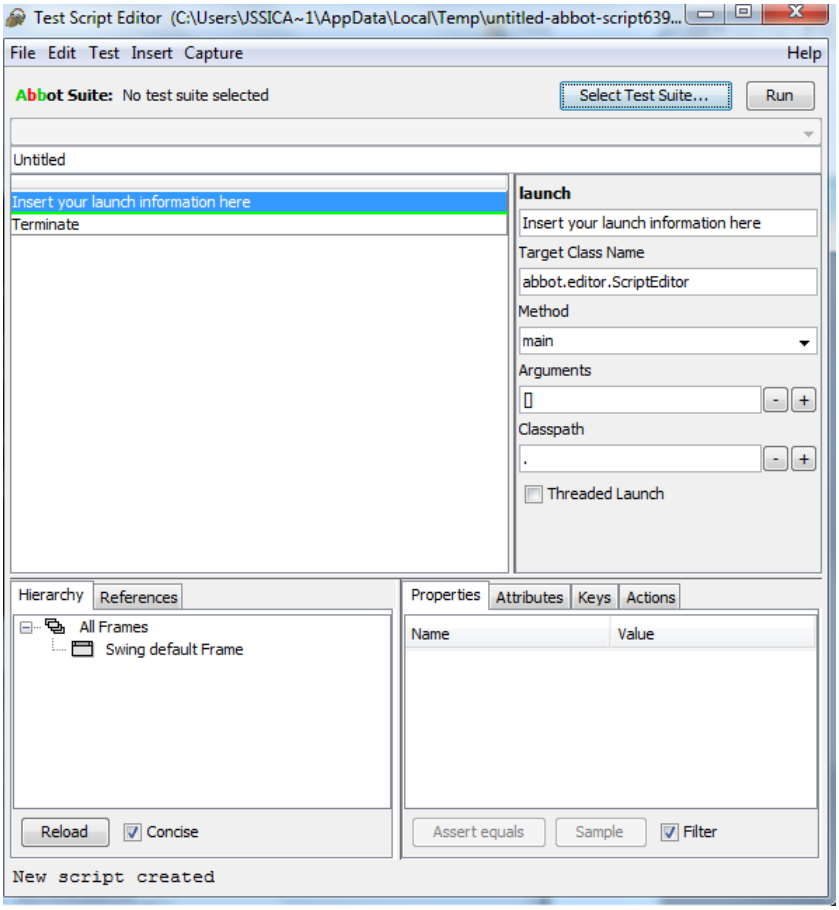
método a serem testados no programa em teste, e o *Terminate*, responsável por fechar o programa ao finalizar a execução (Figura 18). Para carregar a classe desejada, é preciso inserir o caminho onde a mesma está localizada no campo *Classpath* e inserir o nome da classe no campo *Target Class Name*. Para dar um nome ao *script* pode-se alterar o campo *Untitled*, na barra superior localizada antes de *Launch*. Após o preenchimento desses dados, o programa localiza e carrega todos os métodos existentes nessa classe (SOUZA; VALE; ARAÚJO, 2008).

O próximo passo do programa é capturar as ações do usuário ao executar o sistema, para isso basta selecionar a opção *Capture* na barra de menus e em seguida *All Actions*. Ao fazer isso, a classe será executada para que todos os estados do sistema em teste sejam gravados e posteriormente executados (Figura 19) (SOUZA; VALE; ARAÚJO, 2008).

O *script* da Figura 19 foi gerado a partir de um aplicativo Java, que verifica se um aluno foi aprovado ou reprovado, de acordo com suas notas e frequência. Esse *script* pode ser salvo e reexecutado, verificando sua integridade caso haja alguma manutenção no sistema em questão e altere a regra de negócio (SOUZA; VALE; ARAÚJO, 2008).

Essa verificação foi realizada para demonstrar a utilidade do *script*. Foram usados

Figura 18 – Página de configuração do script.

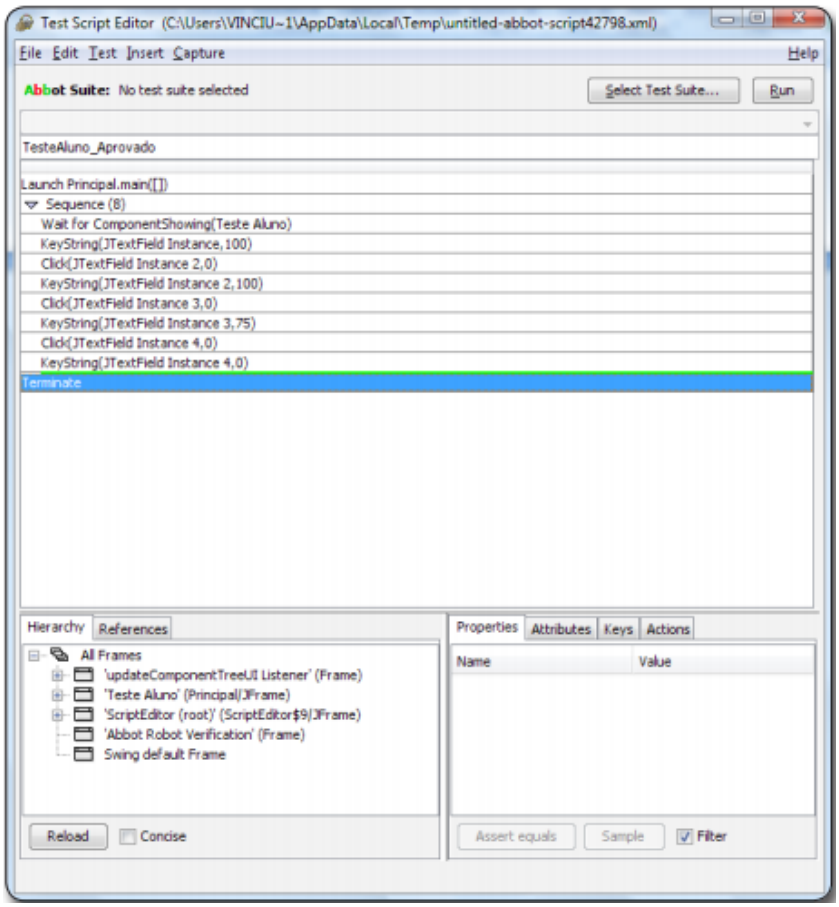


Fonte: (SOUZA; VALE; ARAÚJO, 2008).

valores de teste que gerassem como resultado a aprovação do aluno. A próxima etapa é então realizar as comparações necessárias para validação do sistema. Como o sistema é responsável por classificar a aprovação do aluno, a mensagem gerada após a inserção dos valores que será comparada. Para isso, após a captura realizada, a caixa de seleção *Concise* localizada abaixo da aba *Hierarchy*, deverá ser desmarcada para que apareça todos os componentes. Deve-se então selecionar o componente referente à área de texto que recebe a mensagem referente a aprovação (Figura 20) (SOUZA; VALE; ARAÚJO, 2008).

Após selecionado o componente, clica-se no botão *Assert Equals* para adicioná-lo ao *script* de teste. Esse *assert* criado será o responsável por comparar se a mensagem é verdadeira, caso a mensagem que aparece na área de texto seja diferente de "Aluno Aprovado", um erro será detectado. O *assert* deverá ser movido para antes do evento *Click* do botão sair, pois essa é a última ação a ser realizada. Montada as configurações necessárias, é realizada uma pequena modificação do código fonte para então ser executado novamente. Essa alteração pode ser visualizada na Figura 21. Para o aluno ser aprovado a frequência não poderia ser menor que 75%, valor exato que foi utilizado como entrada.

Figura 19 – Ações capturadas.



Fonte: (SOUZA; VALE; ARAÚJO, 2008).

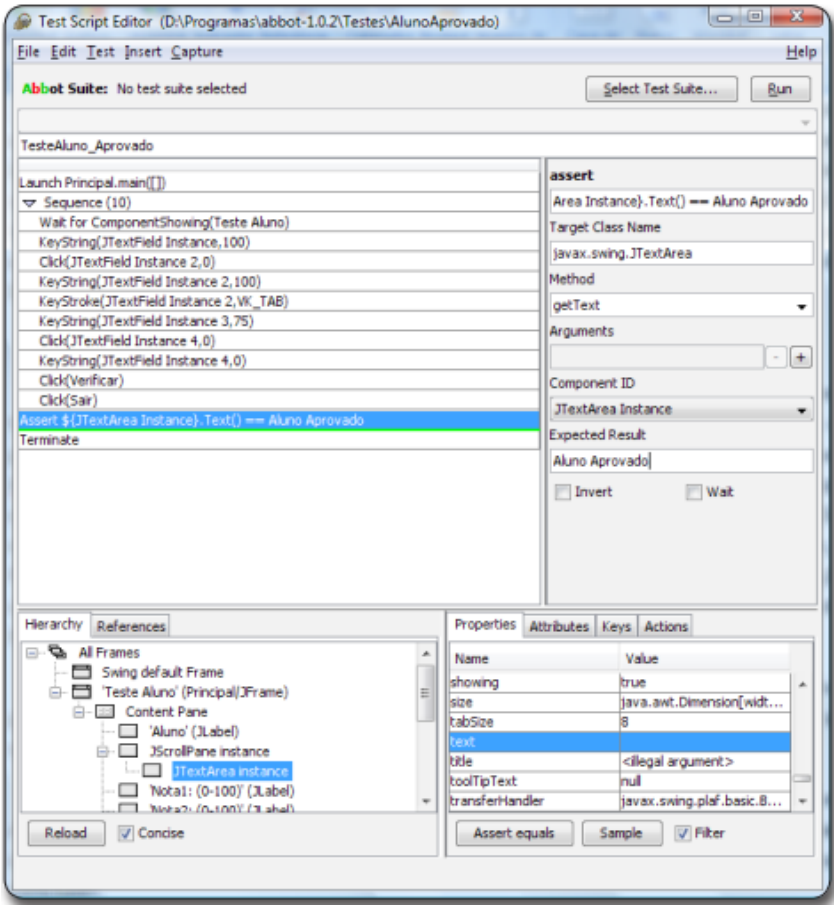
Foi então inserido o operador igual, para indicar que se a frequência for 75% o aluno não será mais aprovado e sim reprovado (SOUZA; VALE; ARAÚJO, 2008).

Após essa alteração, o *script* é então reexecutado. O resultado pode ser visualizado na Figura 22.

De acordo com a imagem, a mensagem indica que o aluno foi reprovado, contradizendo o resultado que fora obtido anteriormente. Isso fez com que o *script* gerasse um erro, ao executar a linha que compara a mensagem, como pode ser observado na Figura 23 (SOUZA; VALE; ARAÚJO, 2008).

A partir dos resultados observados após a execução dos testes no Addob Framework, pode-se notar que a ferramenta é eficaz e muito útil para teste de sistemas desktop Java. Além de ser eficiente, a ferramenta possui uma interface simples e fácil de usar. Como os *scripts* de teste podem ser obtidos automaticamente, o testador não precisa estar familiarizado com o código-fonte do sistema em teste, caso este não esteja disponível. A ferramenta é gratuita e está disponível para download no endereço <http://abbot.sourceforge.net> (SOUZA; VALE; ARAÚJO, 2008).

Figura 20 – Assert incluído.



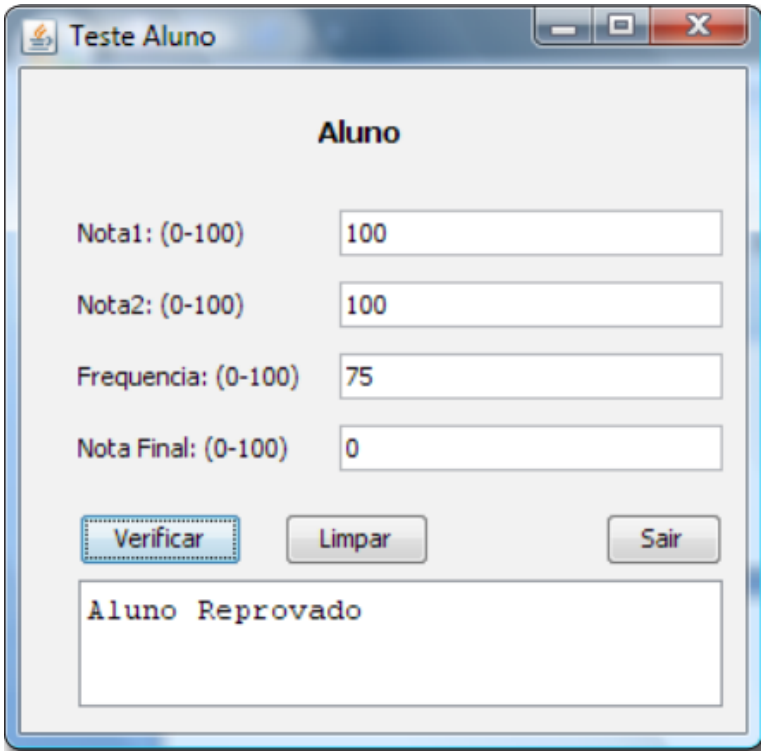
Fonte: (SOUZA; VALE; ARAÚJO, 2008).

Figura 21 – Modificação realizada no código fonte.

```
if (aluno.getFrequencia() < 75){
        jTextArea1.append("Aluno Reprovado");
                        .
                        .
                        .

if (aluno.getFrequencia() <= 75){
        jTextArea1.append("Aluno Reprovado");
                        .
                        .
                        .
```

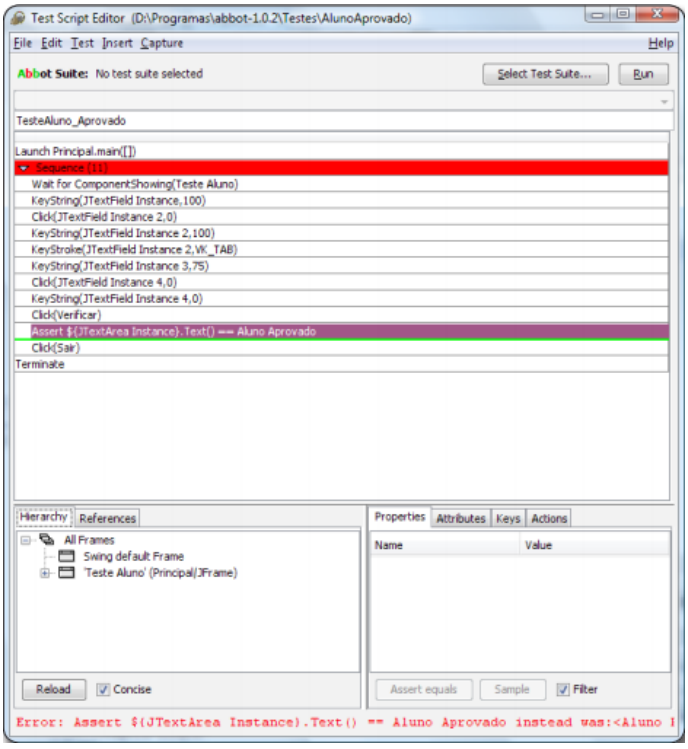
Fonte: (SOUZA; VALE; ARAÚJO, 2008).

Figura 22 – Tela do sistema sendo executada automaticamente.



Fonte: (SOUZA; VALE; ARAÚJO, 2008).

Figura 23 – Erro no Script.



Fonte: (SOUZA; VALE; ARAÚJO, 2008).

4.2.2 Marathon

O Marathon é uma ferramenta *Open Source* usada para automação de testes funcionais. Possui uma ambiente integrado para criação e execução de *scripts* de teste, para aplicações desenvolvidos na plataforma *Java Swing*, que podem ser gravados em *Jython* e *JRuby*. Assim como na ferramenta, Abbot Framework, descrita na seção 4.2.1, o Marathon cria testes automatizados por meio da captura das ações realizadas pelo usuário através da interface do sistema em teste (COLLINS; LOBAO, 2010b).

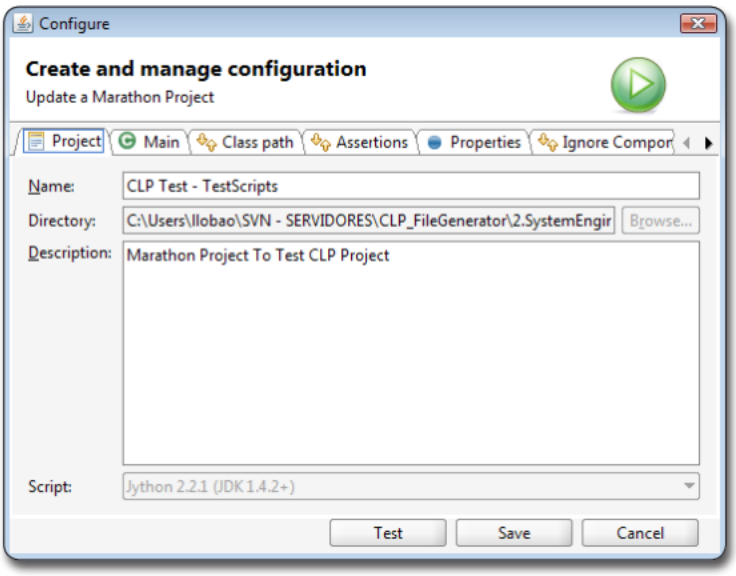
Para exemplificar o uso da ferramenta Marathon foi utilizado um exemplo chamado ProjetoSW, responsável por ler várias planilhas vindas de área de logística, e a partir delas gerar arquivos no formato *.txt*, que possam ser carregados em um outro sistema para a realização da análises dos dados (COLLINS; LOBAO, 2010b).

As configurações realizadas no Maraton que são descritas a seguir, fazem parte de um processo de teste utilizando a integração com outras ferramentas abertas de automação, TestLink e Mantis. Esse processo tem o objetivo de cobrir um ambiente de programação exploratória, com a ausência de uma especificação detalhada do sistema em teste e com alterações constantes de escopo ao longo das entregas das versões para o cliente (COLLINS; LOBAO, 2010b).

Para esse projeto foram utilizadas as abas de configuração do Marathon a seguir (COLLINS; LOBAO, 2010b):

- *Project*: Aba onde são configurados o nome, o diretório, a descrição e o tipo de *script* do projeto a ser testado, Figura 24.

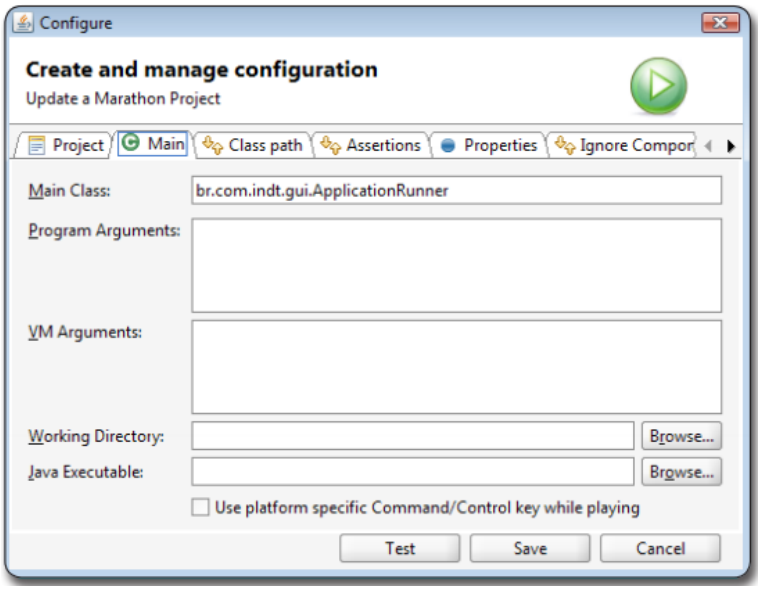
Figura 24 – Configurações básicas do projeto.



Fonte: (COLLINS; LOBAO, 2010b)

- *Main*: Aba onde são informadas as principais configurações do projeto a ser testado, Figura 25.

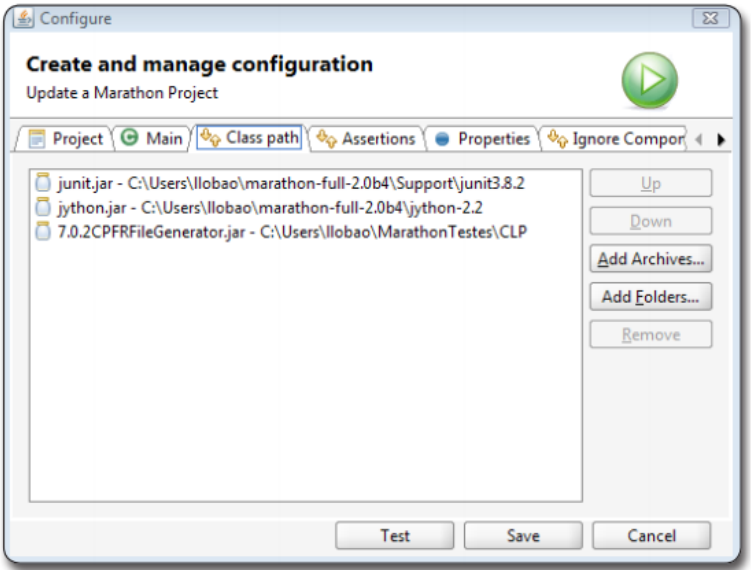
Figura 25 – Configurações principais do Sistema a ser testado.



Fonte: (COLLINS; LOBAO, 2010b)

- *Class Path*: Aba onde são acrescentadas as *libs* e *jar* necessários para a execução da aplicação, Figura 26. É a partir do*classpath* que a ferramenta Marathon busca a classe principal informada na aba *Main*.

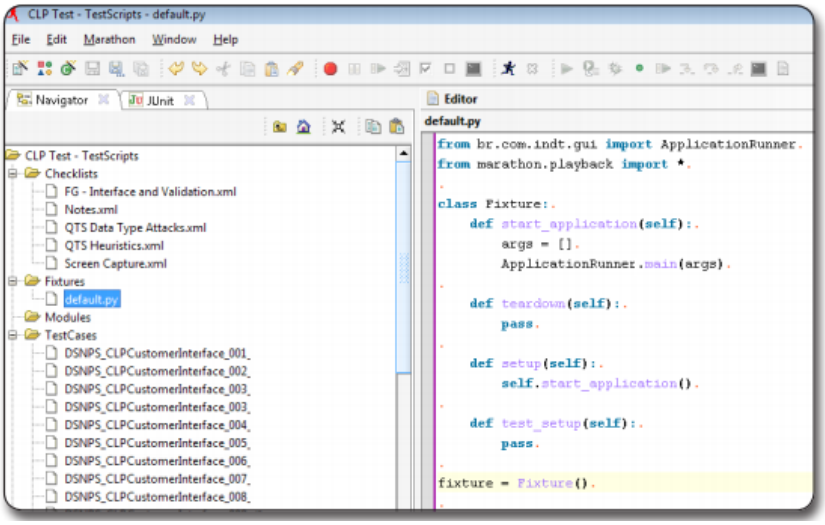
Figura 26 – Tela de Class Path, adicionando libs.



Fonte: (COLLINS; LOBAO, 2010b)

Após preencher as configurações apresentadas nas abas acima, o projeto de teste é salvo e a tela principal do Marathon será exibida, como na Figura 27. O arquivo *defaul.py* é então gerado pelo Marathon. Este arquivo trata-se de um *fixture* reponsável por fornecer uma interface onde o testador irá executar as ações de teste como abrir um arquivo, por exemplo (COLLINS; LOBAO, 2010b).

Figura 27 – Tela principal do Marathon.



Fonte: (COLLINS; LOBAO, 2010b)

Um exemplo de uma navegação realizada pela interface do aplicativo em teste, no caso o ProjetoSW, é mostrado na Listagem 1 (Figura 28). Foi utilizada a API Java no código do caso de teste feito no Marathon (COLLINS; LOBAO, 2010b).

Na figura 29 é possível observar o exemplo da tela principal do ProjetoSW. Voltando no *script* da Listagem 1 (Figura 28), nas linhas 20 a 25, é realizada a ação referente à seleção de um cliente no *Combobox*, para que as próximas ações, *Import*, *Mapping*, *Export*, *Backup* e *Generate XLS*, possam ser realizadas. Ao selecionar o cliente "Operadora1", o sistema é redirecionado para a tela de *Import*, como pode ser visualizado na Figura 30 (COLLINS; LOBAO, 2010b).

Essa tela do ProjetoSW, mostrada na Figura 30, é responsável por por realizar o *upload* das planilhas referentes ao cliente selecionado, nesse caso a "Operadora 1", para que posteriormente as informações possam ser processadas e exportadas no formato *.txt* (COLLINS; LOBAO, 2010b).

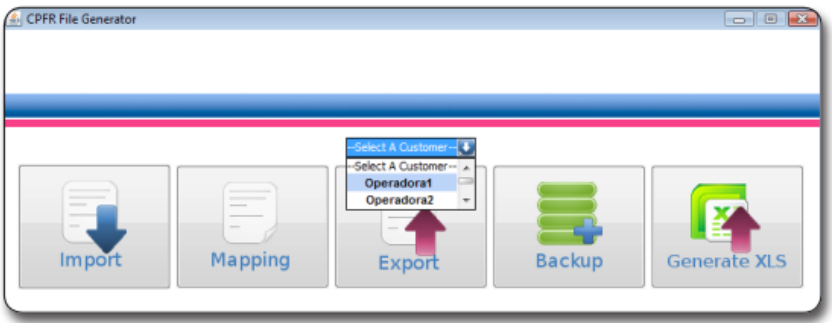
Após a importação dos arquivos realizada, o ciclo referente ao *Import* é finalizado e o próximo passo é mapear e exportar os dados em arquivos de texto, seguindo a especificação do sistema. Um exemplo desse mapeamento é mostrado na Listagem 2, da Figura 31. Na Figura 32, é mostrada a tela de mapeamento com alguns dados que já foram importados e o mapeamento realizado (COLLINS; LOBAO, 2010b).

Figura 28 – Listagem 1 - Exemplo simples de navegação.

```
#{{{ Marathon Fixture
from default import *
#}}} Marathon Fixture

from java.awt import AWTException
from java.awt import Robot
from java.awt.event import KeyEvent

def test():
java_recorded_version = '1.6.0_10'

robot = Robot();

if window('CPFR File Generator'):
assert_p('TextField', 'Text', '')
sleep(1)
robot.keyPress(KeyEvent.VK_TAB)
sleep(1)
    # Escolher 'operadora um'
    for i in range(0, 1):
if window('CPFR File Generator'):
  robot.keyPress(KeyEvent.VK_DOWN)
  robot.keyRelease(KeyEvent.VK_DOWN)
close()
close()
sleep(1)
robot.keyPress(KeyEvent.VK_ENTER)
robot.keyRelease(KeyEvent.VK_ENTER)
sleep(1)
assert_p('ComboBox', 'Operadora1')
close()
```

Fonte: (COLLINS; LOBAO, 2010b)

Figura 29 – Tela principal.

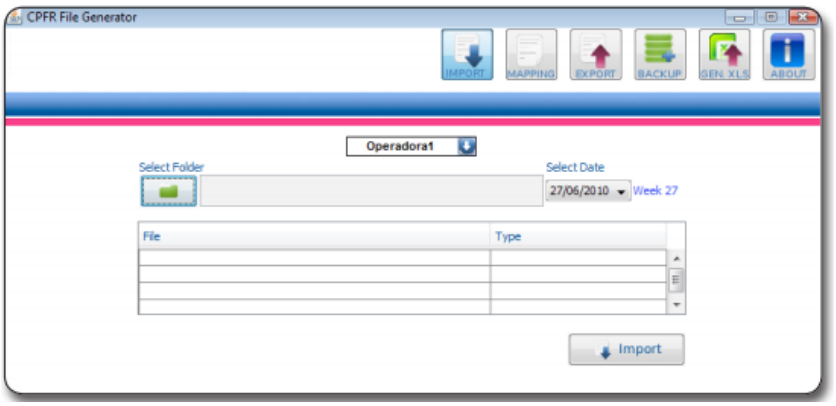


Fonte: (COLLINS; LOBAO, 2010b)

A partir dos comandos da listagem 2 (Figura 31), o Marathon acrescenta as *strings* no campo de mapeamento (Figura 32). A partir do mapeamento realizado, o ProjetoSW poderá exportar os dados no formato de arquivo de texto. A tela de *Export* é mostrada na Figura 33. Nessa tela é selecionada a pasta de destino dos arquivos que serão exportados. É possível observar os passos dessa exportação na Listagem 3 da Figura 34 (COLLINS; LOBAO, 2010b).

Após os dados exportados, termina o ciclo referente a especificação do sistema. Todos estes passos foram gerados automaticamente pelo Marathon a partir da navegação

Figura 30 – Tela de Import.



Fonte: (COLLINS; LOBAO, 2010b)

Figura 31 – Listagem 2 - Mapeando os dados.

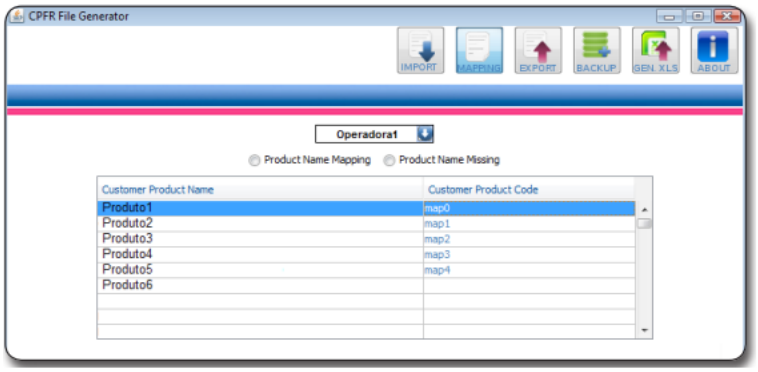
```
48. # {{{
49. #      Continuação
50. # }}}
50.
51. if window('CPFR File Generator'):
52.  select('mapping_main2','true')
53.
54      for i in range(0, 4):
55.      if window('CPFR File Generator'):
56.          doubleclick('Table'+str(i+1),'{'+ str(i) +', Customer Product Code}')
57.       sleep(1)
58.       select('Table'+str(i+1),'map','{'+ str(i) +', Customer Product Code}')
59.     close() # end if
60.     close() # end for
61.     close() # end if
62.
63. # {{{
64. #      Continua na próxima listagem
65. # }}}
```

Fonte: (COLLINS; LOBAO, 2010b)

feita na aplicação pelo testador (COLLINS; LOBAO, 2010b).

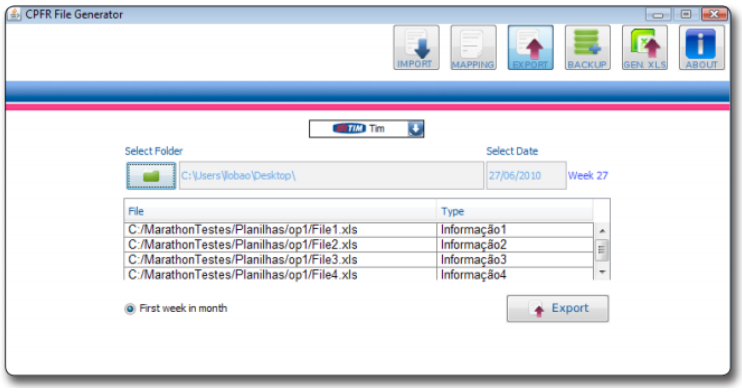
Os próximos passos são referentes ao ciclo de execução dos testes, com a integração entre as ferramentas Marathon, TestLink (apresentada na seção 4.2.7) e Mantis, que juntas definem o conjunto de atividades de teste a serem executadas a cada liberação de versão do sistema, levando em conta a programação exploratória e a ausência de um documento de especificação detalhado. Como o objetivo principal é mostrar o funcionamento da ferramenta Marathon, que foi realizada, não foi apresentado detalhes a respeito das etapas referentes a essa integração.

Figura 32 – Tela de Mapping.



Fonte: (COLLINS; LOBAO, 2010b)

Figura 33 – Tela de Export.



Fonte: (COLLINS; LOBAO, 2010b)

Figura 34 – Listagem 3 - Exportando os dados.

```
66. # {{{
67. #      Continuação
68. # }}}
69.
70. if window('CPFR File Generator'):
71. select('export_main2','true')
72. click('bot_folder_normal21')
73.
74. if window('Open'):
75.   select('FileChooser','C:/Users/llobao/Desktop')
76. close()
77.
78. click('bot_export_normal2')
79.
80. if window('Message'):
81.   click('OK')
82. close()
83. close()
84.
85. # {{{
86. #      Fim
87. # }}}
```

Fonte: (COLLINS; LOBAO, 2010b)

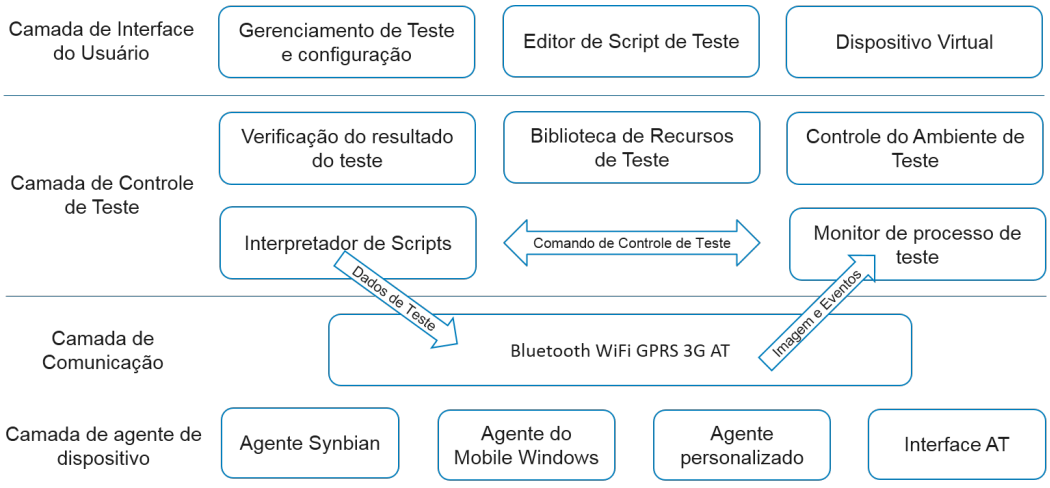
4.2.3 MobileTest

MobileTest é uma ferramenta automática de teste de caixa preta para dispositivos móveis inteligentes, baseada em eventos sensíveis ao contexto. Ela pode criar uma biblioteca de casos de teste sofisticados, sustentáveis e reutilizáveis, para teste a nível de sistema e software a nível de aplicação em uma variedade de dispositivos móveis inteligentes (BO; XIANG; XIAOPENG, 2007).

Para reduzir a complexidade do teste, o sistema usa um design em camadas (Figura 35). Cada camada fornece serviços para a camada superior com o suporte da camada inferior. Deste modo, a camada de controle de teste, responsável por controlar toda etapa referente a execução dos testes, pode ser separada das peculiaridades dos dispositivos subjacentes. Essas camadas são divididas em (BO; XIANG; XIAOPENG, 2007).:

- **Camada de interface do usuário:** Responsável por interagir com os testadores;
- **Camada de controle de teste:** Responsável por executar os *scripts* de teste, enviar operações simuladas para os dispositivos de destino, receber os *screenshots* (capturas de tela) e os eventos sensíveis dos dispositivos de destino e ainda controlar o processo de teste de acordo com os eventos sensíveis;
- **Camada de comunicação:** Responsável por conectar a camada de controle de teste à camada de agente de dispositivo;
- **Camada de agente de dispositivo:** Recebe os comandos da camada superior, executa-os e retorna o status do alvo, a partir de uma captura de tela ou um evento sensível ao contexto .

Figura 35 – Arquitetura do MobileTest.



Fonte: Adaptado de (BO; XIANG; XIAOPENG, 2007), **tradução nossa.**

A abordagem de teste baseada em eventos sensíveis ao contexto para testar operações interativas entre vários dispositivos, é um dos objetivos do MobileTest e é descrita a seguir. A mesma visa suportar o teste de volume, o teste de múltiplos estados, o teste de limite e o teste de múltiplas tarefas (BO; XIANG; XIAOPENG, 2007).

Neste caso, os eventos sensíveis são quaisquer eventos que ocorram no dispositivo em teste que estejam relacionados ao objetivo do teste atual. Esses eventos podem ser "caixa de entrada cheia", "chamada recebida", "memória baixa", "disco cheio", etc. (BO; XIANG; XIAOPENG, 2007).

- **Suporte para testes de operações interativas**. Testar a operação interativa requer que o sistema possa controlar vários dispositivos simultaneamente e fornecer um mecanismo para a sincronização entre vários dispositivos. Para conectar vários dispositivos simultaneamente, o módulo de comunicação fornece a capacidade de se conectar usando as tecnologias *Bluetooth*, *WiFi* ou 3G. Já a sincronização é realizada por um programa de um agente que é executado no dispositivo de destino, para capturar eventos como recebimento de chamada e SMS e enviá-los para serem testados na camada de controle de teste;

- **Suporte para teste de volume e teste de limite**. O teste de volume significa enviar volumes extremamente altos de dados para dispositivos de destino. Um exemplo seria criar novos rascunhos de mensagem até encher a caixa de entrada. O teste de limites tem como objetivo testar o funcionamento do dispositivo de destino, quando está no processo de uma operação longa ou durante uma transição de estado, cmomo por exemplo, receber um SMS ao esvaziar a caixa de entrada. Durante a execução de ambos os testes, o sistema precisa ser notificado quando dispositivo de destino altera um estado específico, como caixa de entrada cheia ou memória baixa. Neste caso, o programs de agente é responsável por capturar esses eventos quando eles ocorrem;

- **Suporte para teste de múltiplos estados e teste de múltiplas tarefas**. O teste de múltiplos estados é para verificar o funcionamento do dispositivo alvo, quando ele está em diferentes estados, por exemplo, rodar uma aplicação quando a bateria está baixa. O teste de múltiplas tarefas tenta verificar se vários aplicativos podem ser executados simultaneamente nos dispositivos de destino. Exemplo: Alternar entre múltiplas aplicações rapidamente ou receber um SMS quando se ouve música.

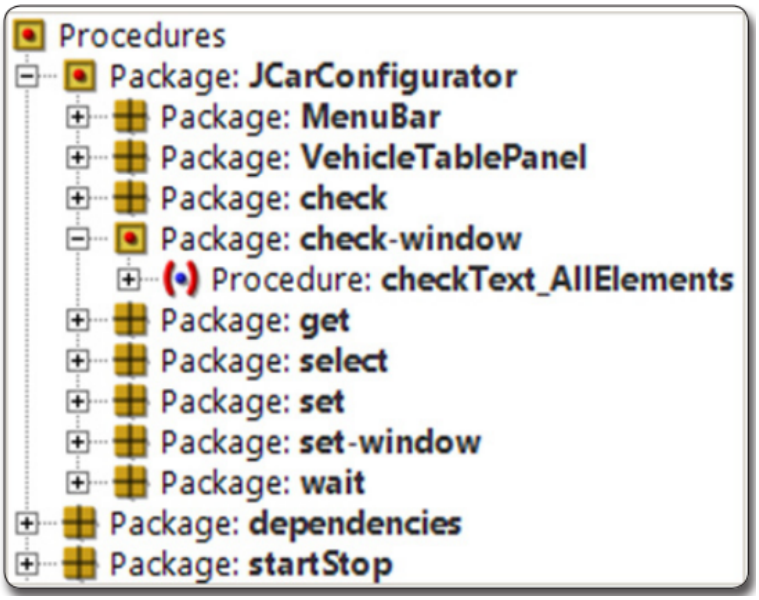
Essa abordagem apresentada, tem como objetivo capturar os eventos sensíveis do dispositivo de destino em tempo real e notificá-los à camada de controle de teste. Essa camada é responsável por tomar outras ações para alterar o processo de teste de acordo com as regras especificadas (BO; XIANG; XIAOPENG, 2007).

Um experimento real dessa ferramenta foi aplicado para três modelos de telefones celulares diferentes da mesma plataforma. A partir dos resultados foi feita uma análise de comparação a outra ferramenta de teste de dispositivos móveis existente, que mostrou a eficácia do MobileTest. A ferramenta adotou uma abordagem baseada em eventos sensíveis para simplificar a concepção de casos de teste e melhorar a eficiência e a reutilização desses casos de teste. Ele também adotou a tecnologia de agente de software para simplificar o design do sistema (BO; XIANG; XIAOPENG, 2007).

4.2.4 QF-Test

QF-Test é uma ferramenta de automação para testes de software em aplicações com Interfaces Gráficas (GUI) para *Web* ou em Java. Ela possui um recurso muito útil de *procedures*, onde é possível criar *packages* com sequências de partes de testes que podem ser reutilizadas posteriormente. O uso desses *procedures* deixa o conjunto de testes mais organizado e mais fácil para se realizar manutenções nos códigos gravados. Isso porque como cada *package* concentra partes de um componente específico, que pode ser testado isoladamente após uma manutenção realizada, por exemplo (Figura 36) (IZAC, 2015).

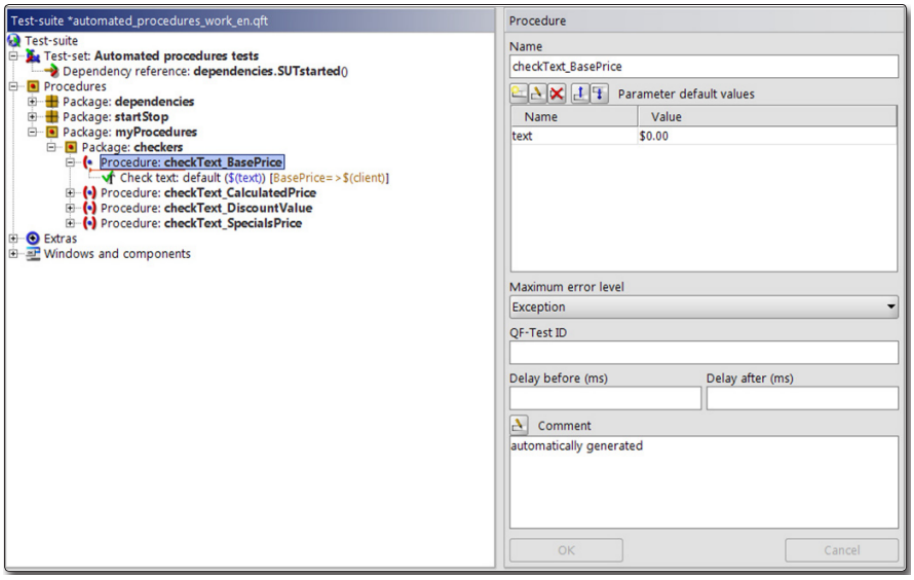
Figura 36 – Exemplo de procedures no QFTest.



Fonte: (IZAC, 2015)

Na Figura 37, é possível observar que como as *procedures* ficam dentro de *packages* é possível agrupar aquelas que possuem um mesmo assunto na mesma *package*. O *package checkers* criado, por exemplo, é utilizado para validar um campo, de acordo com valor *default* que foi atribuído à ele. No exemplo, para cada *check* criado a um campo, é criada um *procedure* no *package "checkers"*. Assim, sempre que necessário validar um campo, basta chamar a *procedure* referente a ele (IZAC, 2015).

Figura 37 – Exemplo de uma Package no QF-Test.



Fonte: (IZAC, 2015)

Além disso, é possível criar variáveis para que haja mais flexibilidade no valor a ser utilizado. Na Figura 37, por exemplo, onde foi criado o parâmetro "text" poderia ter sido criada uma variável no lugar do parâmetro 0. Assim, é possível informar um valor diferente para esse parâmetro, sempre que a *procedure* for chamada (IZAC, 2015).

Existe um campo de comentário para facilitar a consulta de testes para manutenção, seu preenchimento não é obrigatório mas é essencial. Ao gravar um teste no QF-Test é criado um arquivo com a extensão *.qft*. É importante pensar em uma melhor forma para organizar as gravações de teste nos *qfts* correspondentes. Caso se tenha um arquivo muito grande com todos os testes da aplicação gravados, seria importante separar os *qfts* por menu ou funcionalidade. Após criar as *packages* com suas respectivas *procedures*, elas serão chamadas nos testes, como no exemplo da Figura 38 (IZAC, 2015).

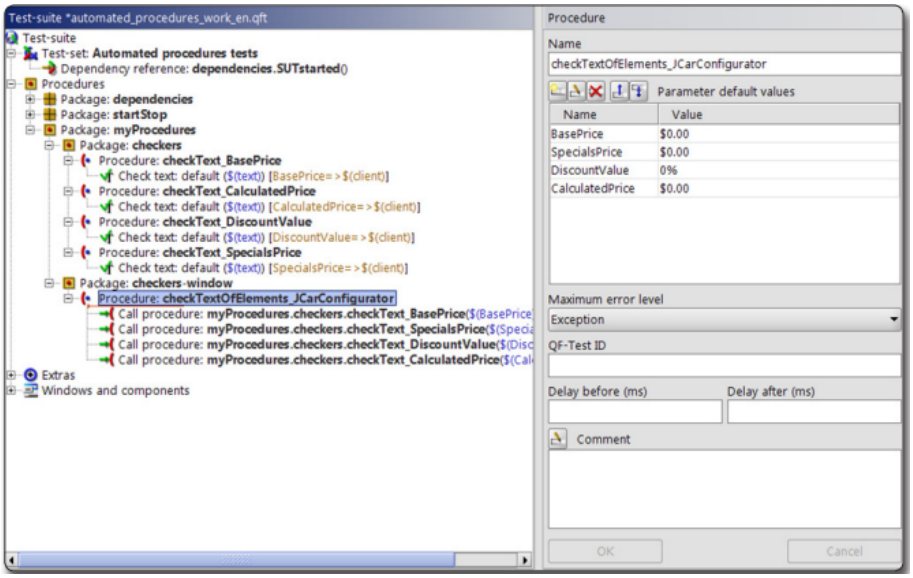
A chamada é feita com o nome da *package* separado por "." da *procedure*. O QF-Test possui também uma biblioteca com várias *packages* incluídas, que podem ser aplicadas para várias situações diferentes. Uma muito utilizada é *package Menu*, sua função é organizar a chamada de *menu-sub-menus* da aplicação (IZAC, 2015).

Para utilizar uma variável é necessário declará-la e então ela poderá receber outros valores, durante a execução dos testes. A variável pode ser declarada através do componente *Set Variable* (IZAC, 2015).

Segue abaixo as sintaxes utilizadas no QF-Test para declarar/atribuir valores a uma variável(IZAC, 2015):

- $(NomeVariável): Será passado nome da variável ao invés do conteúdo do campo;

Figura 38 – Exemplo de chamada de uma Procedure no QF-Test.

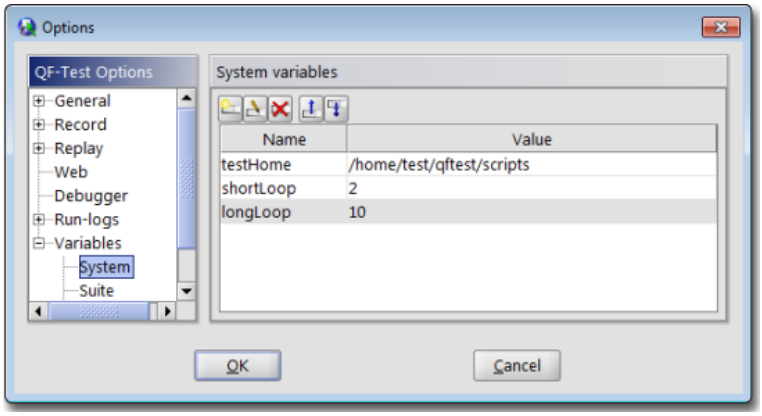


Fonte: (IZAC, 2015)

- $ResultadoGrupo:NomeCampo: Um valor é atribuído a uma variável depois de chamar um *ExecuteSelectStatement*. E então um nome é criado para ela nessa construção;
- $[expression]: Uma variável é atribuída, passando expressões numéricas.

As variáveis que recebem valores durante a execução do QF-Test podem ser visualizadas no menu: Edit/Options/Variables, um exemplo é mostrado na Figura 39 (IZAC, 2015).

Figura 39 – Exemplo de lista de variáveis que foram atribuídas na execução em QFTest.



Fonte: (IZAC, 2015)

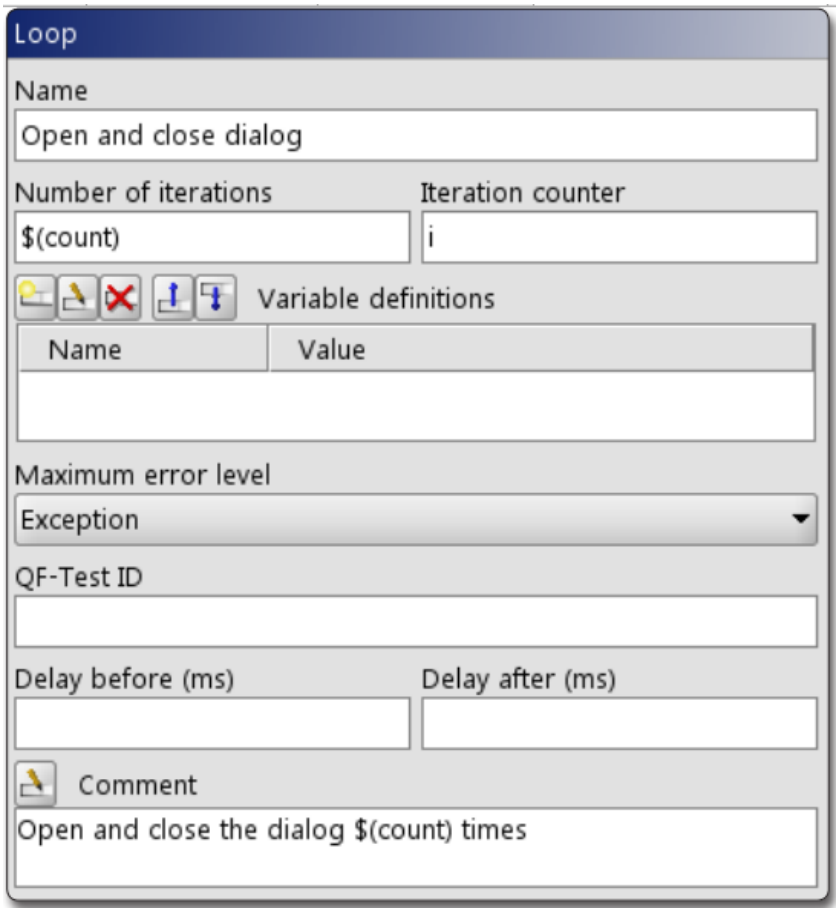
Um exemplo de uma variável sendo chamada em um componente é apresentado na Figura 40 (IZAC, 2015).

Figura 40 – Exemplo de variáveis em QF-Test.

Fonte: (IZAC, 2015)

Outros recursos disponibilizados pelo QF-Test, podem ser utilizados para uma melhor performance dos testes automáticos. Como por exemplo, o uso dos comandos condicionais *Loop*, *While*, *Break*, *If*, *Try*. Na figura 41 é mostrado um exemplo do uso da estrutura *Loop* para abrir e fechar um *dialog* executando o que estiver dentro. A variável $(count) é responsável por definir o número de iterações a serem executadas e o campo *"Iteration counter"* definine o contador inicial do *loop* (IZAC, 2015).

Figura 41 – Exemplo de estrutura de controle Loop no QF-Test.

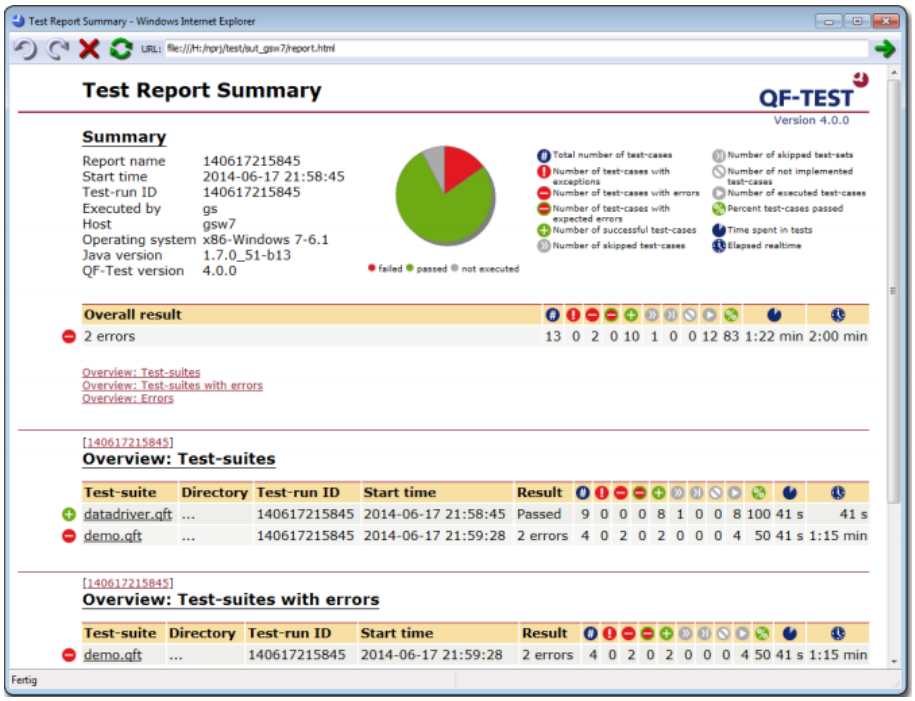


Fonte: (IZAC, 2015)

Essas foram algumas configurações disponíveis no QF-Test para melhorar a automação dos testes a serem executados. Segundo Izac (2015), são questões importantes a serem consideradas antes de iniciar a automação dos testes de uma aplicação, como a manutenibilidade, por exemplo. Pois a automação de testes funcionais não se resume apenas em realizar o *Record/Play*, é importante se ter um bom conhecimento sobre as práticas de teste e realizar um planejamento de teste bem definido para que se tenha sucesso.

Com o planejamento já definido e aplicado na ferramenta QF-Test, a próxima etapa é gravar os testes automáticos e então executá-los. Essa etapa é feita, como nas ferramentas anteriores, gravando-se as ações do usuário e reexecutando-as, posteriormente. O QF-Test possui alguns relatórios que podem ser gerados em *XML* ou *HTML* mostrando o resultado da execução dos testes, a quantidade total de testes existentes no *qft*, a duração do teste, a quantidade de testes que falharam, passaram ou não foram executados, entre outros, conforme a figura 42.

Figura 42 – Exemplo de Relatório gerado pelo QF-Test.



Fonte: (IZAC, 2015)

4.2.5 Selenium IDE

A ferramenta Selenium IDE é um ambiente de desenvolvimento integrado para testes com *selenium*. Ela é baseada na abordagem de *capture-replay* e inclui o Selenium Core, que permite a gravação e reprodução dos testes no ambiente onde eles são executados. (NETO, 2010) Funciona como um plugin para o navegador Mozilla Firefox e pos-

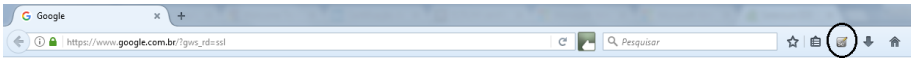
sui licença gratuita, podendo ser baixado em <https://addons.mozilla.org/pt-br/firefox/addon/selenium-ide/>.

Segue abaixo as principais funcionalidades do Selenium IDE, de acordo com (NETO, 2010):

1. Gravação e reprodução dos testes de forma simples;
2. Seleção de campo inteligente, que captura as informações sobre os campos a partir da página;
3. Funcionalidade autocompletar para todos os comandos de Selenium;
4. Possibilita navegar entre os testes;
5. Possibilita configurar depuração e pontos de paradas para verificação durante os testes;
6. Permite salvar os testes como *html*, *scripts* em Ruby ou outro formato.

Para gravar e executar os testes a partir do Selenium, primeiro deve-se abrir a página da Web que será testada. E logo depois deve-se abrir o Selenium IDE, que após baixado e instalado estará disponível ao lado da barra de navegação do Firefox, como na Figura 43 (NETO, 2010).

Figura 43 – Plugin Selenium IDE no Firefox.



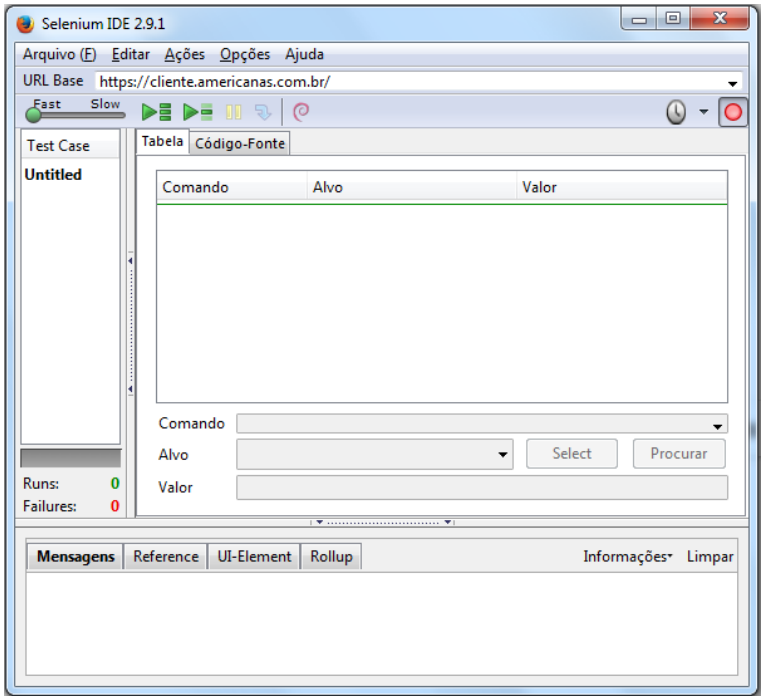
Fonte: Elaborado pelo autor.

Ao abrir a ferramenta, aparecerá a tela onde os testes são gravados e executados. A opção referente a gravação já estará ativada por *default*, a mesma pode ser visualizada na Figura 44, onde ícone de uma bola vermelha localizado no canto superior direito da tela já se encontra selecionado (NETO, 2010).

Com a ferramenta aberta e o botão de gravação ativado, deve-se voltar a página *Web* que será testada e executar manualmente as operações de entrada dos testes a serem feitos, que já deverão ter sido elaborados previamente. Os resultados dessas operações podem ser visualizados na Figura 45, referente ao teste realizado na página de cadastro de um novo cliente do site de compras da americanas <https://carrinho.americanas.com.br/portal/meuCadastro.portal> (NETO, 2010).

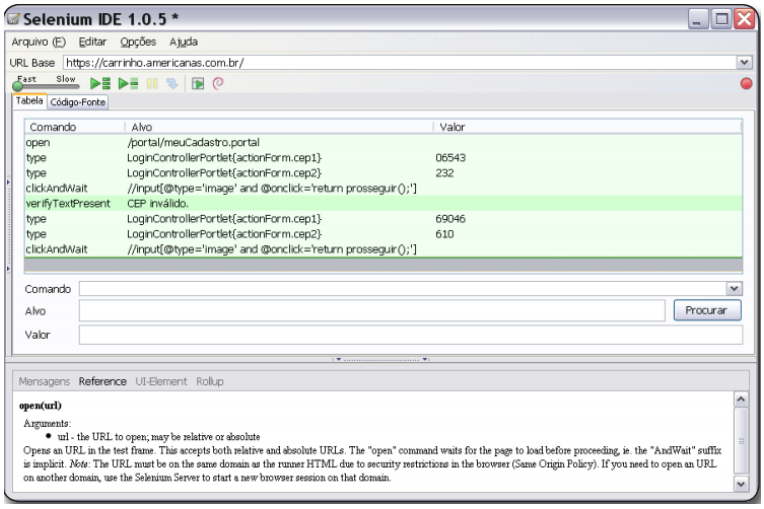
Após o término das operações de teste realizadas na página, a gravação pode ser interrompida pelo Selenium IDE ao clicar no botão vermelho redondo, que estava ativado. Assim, já se tem o primeiro roteiro de teste gravado, ele pode ser então salvo e reproduzido para se repetir os testes e tentar encontrar erros (NETO, 2010).

Figura 44 – Tela inicial do Selenium IDE, com opção de gravação ativada.



Fonte: Elaborado pelo autor.

Figura 45 – Roteiro de teste Selenium IDE 1.0.5.



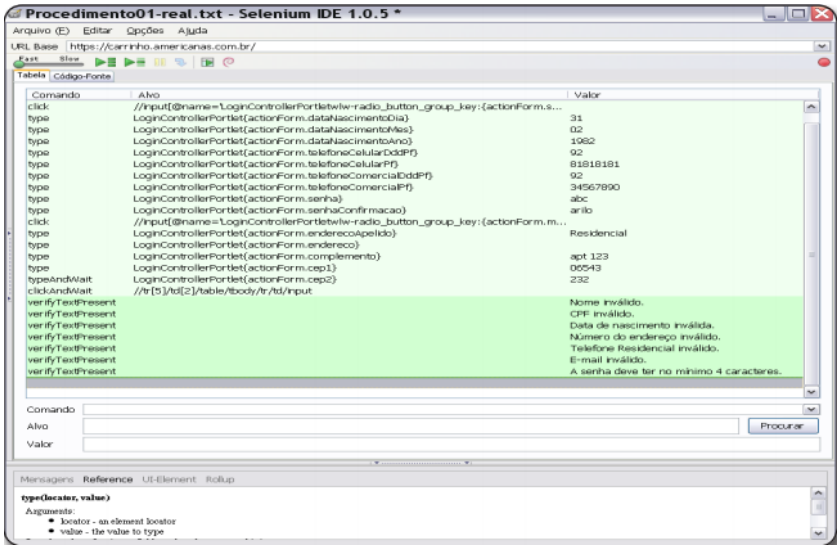
Fonte: (NETO, 2010).

No exemplo executado na Figura 45, foi adicionado o comando *verifyTextPresent*. Esse comando verifica se ao digitar um CEP inexistente, irá aparecer a mensagem "CEP inválido." na tela tela de retorno, como normalmente deveria acontecer para este caso de teste (NETO, 2010).

Outro exemplo de roteiro de teste, pode ser visualizado pelas figuras 46 e 47. Neste caso foi realizado o cadastro de um novo cliente, ao preencher todo o formulário com os

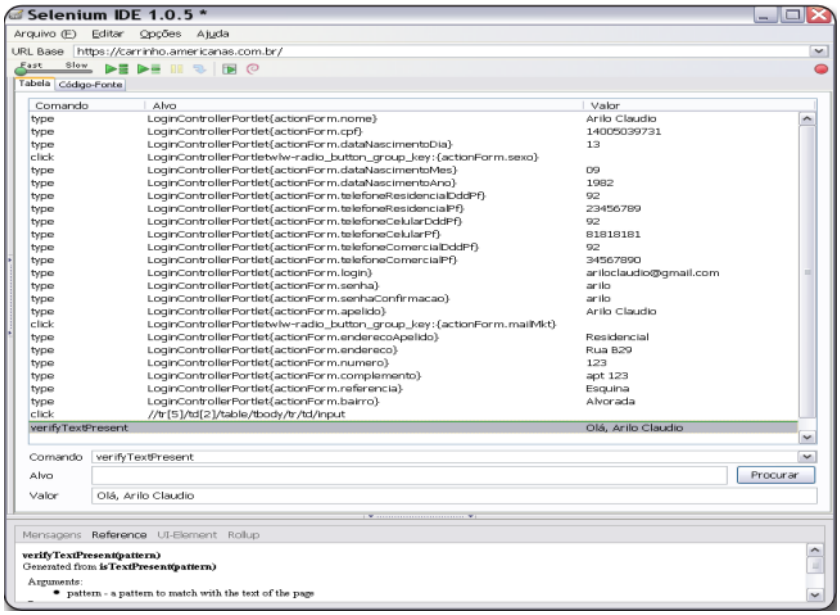
dados respectivos. Para fins de teste, no primeiro exemplo foram adicionados valores inválidos em alguns campos e no segundo exemplo apenas valores válidos (NETO, 2010).

Figura 46 – Roteiro de teste com valores inválidos.



Fonte: (NETO, 2010).

Figura 47 – Roteiro de teste com valores válidos.



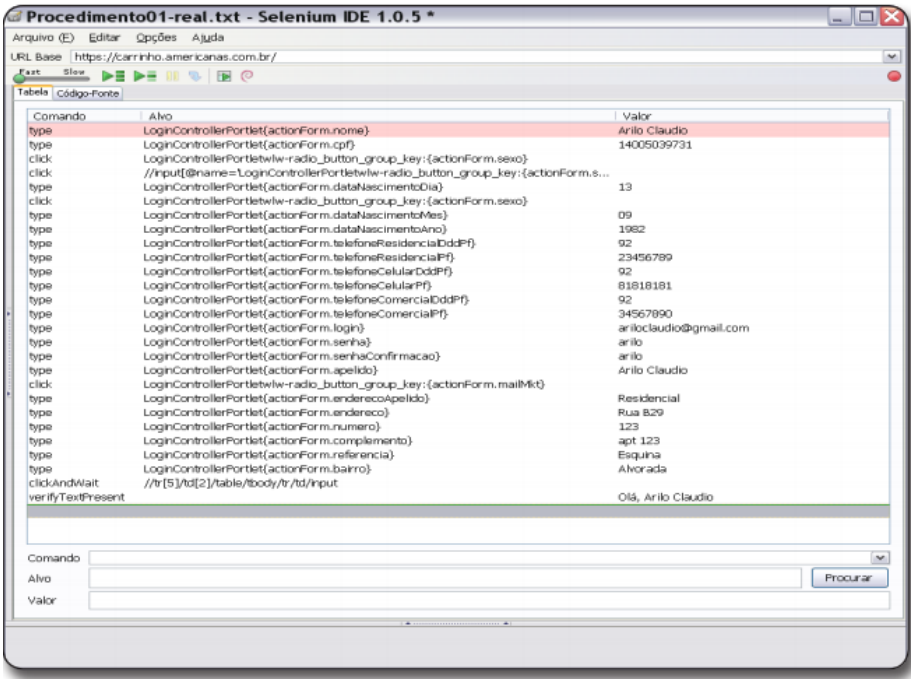
Fonte: (NETO, 2010).

O próximo passo é salvar os roteiros de teste gravados para que possam ser repetidos sempre que necessário.

Para reproduzir o teste novamente, basta abrir o arquivo com o roteiro de teste no local onde foi salvo e clicar no ícone verde com uma seta de *play* e três barras verdes à sua direita. A ferramenta Selenium irá então reproduzir todos os passos que foram realizados

durante a gravação dos testes, como o preenchimento dos campos de cadastro. Se um comando é executado com sucesso, a linha equivalente fica verde. Agora, caso haja uma falha a linha equivalente ficará vermelha, indicando um problema que deverá ser analisado (NETO, 2010). Uma falha de login ocorrida, é mostrada na Figura 48 ao tentar reproduzir o procedimento de teste executado na figura 46, onde foram adicionados valores inválidos.

Figura 48 – Reprodução de teste com falha.



Fonte: (NETO, 2010)

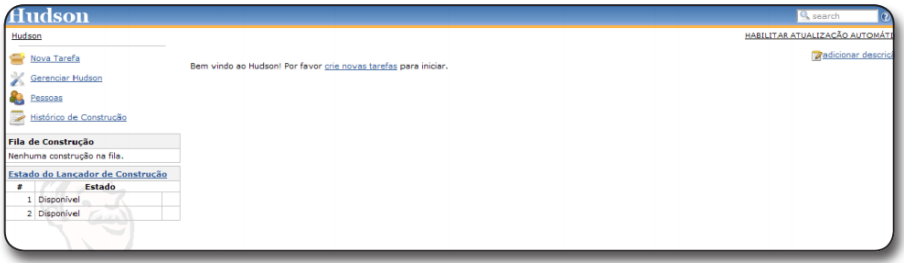
Obs.: Esses exemplos foram realizados na página de cadastro de novo cliente no site de compras americanas no ano de 2010. A página teve alterações, dessa forma o exemplo não é mais válido para a data atual.

O Selenium IDE também pode ser integrado a outras ferramentas de teste, de forma a tornar o sistema em teste mais confiável e livre de erros, pois possibilita a execução de testes periodicamente. Essa técnica é chamada de integração contínua, e é descrita em (ALMEIDA; RIBEIRO; ARAÚJO, 2010).

Almeida, Ribeiro e Araújo (2010) explicam como gerar testes funcionais a partir do Selenium IDE, logo após como importar esses testes para o Framework de testes JUnit e finalmente como exporta-los como um arquivo de teste do JUnit para serem executados nas ferramentas Hudson e Selenium RC.

O Selenium RC (Remote Control) é responsável por executar automaticamente os testes criados no Selenium IDE e o Hudson é um servidor de integração contínua, onde é possível programar a execução de testes, criando um ambiente de testes automatizados programaticamente (Figura 49) (ALMEIDA; RIBEIRO; ARAÚJO, 2010).

Figura 49 – Tela inicial ferramenta Hudson.



Fonte: (ALMEIDA; RIBEIRO; ARAÚJO, 2010)

O Selenium IDE é uma ótima ferramenta para a geração de testes funcionais em páginas *Web*. Possui uma interface bem intuitiva e fácil de usar, além de poder ser integrada a outras ferramentas e tornar a execução de testes mais eficiente, mas para que se tenha um processo de teste bem sucedido é importante que a equipe de teste tenha um bom conhecimento técnico para gerar os casos de teste ideais. (NETO, 2010).

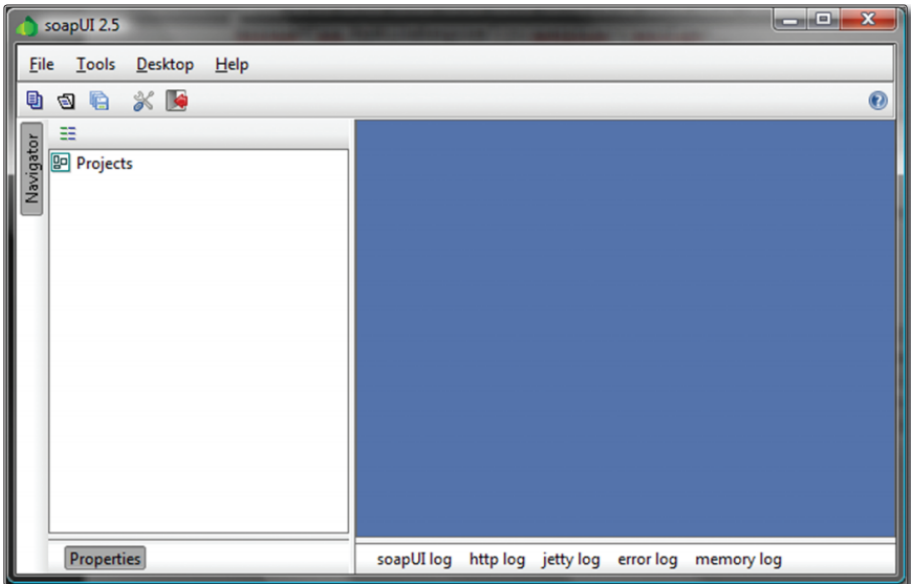
4.2.6 SoapUI

A SoapUI é uma ferramenta para teste de *WebServices*, desenvolvido na linguagem Java que roda nos principais sistemas operacionais, incluindo Windows e Linux. Possui licença nas versões *Open Source* e paga, que estão disponíveis pelo endereço eletrônico <https://www.soapui.org/> (DAIBERT; TOLEDO; ARAúJO, 2008). Dentre as principais funcionalidades citadas por Daibert, Toledo e Araújo (2008) estão a capacidade de gerenciar um número ilimitado de requisições para cada operação, o gerenciamento de múltiplos *endpoints* para cada *WebService*, a validação das requisições e respostas contra as suas definições no *WSDL*(Linguagem de Descrição para *WebServices*), os testes de carga e stress e os testes funcionais (DAIBERT; TOLEDO; ARAúJO, 2008).

Para realizar o teste funcional a partir do SoapUI, é necessário gerar a linguagem *WSDL* do serviço a ser testado, pois é a partir da *WSDL* que a ferramenta irá criar o projeto para teste e invocar a *WebService* referente. Essa linguagem é utilizada para descrever uma *WebService* e ela é baseada em *XML*. A tela inicial do SoapUI, pode ser visualizada na Figura 50 (DAIBERT; TOLEDO; ARAúJO, 2008).

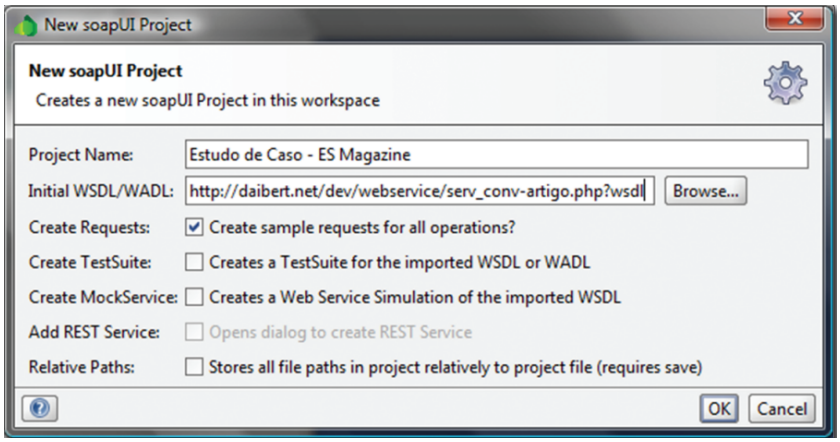
Para iniciar um novo projeto de teste, clica-se em *file* e seleciona-se a opção *New SoapUI Project*. Irá então aparecer uma janela, onde deverá ser preenchido o campo *Initial WDSL/WADL* com a *URL* da *WSDL* referente, assim que esse campo for preenchido o nome do projeto irá aparecer no campo *Project Name*, que pode ser alterado para o nome desejado. Após preencher os campos, as outras opções dessa janela devem ficar como *default*, apenas com a primeira opção do *Check Box* ativada, Figura 51 (DAIBERT; TOLEDO; ARAúJO, 2008).

Figura 50 – Interface principal da ferramenta SoapUI.



Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

Figura 51 – Criando um novo projeto no SoapUI.

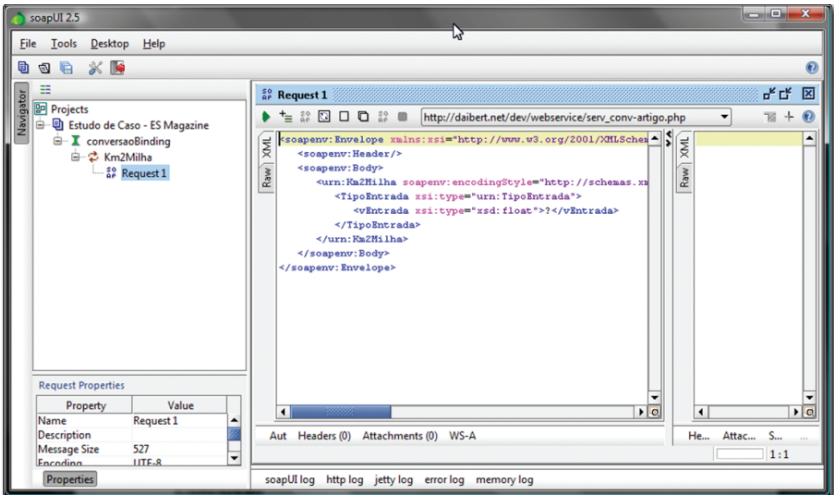


Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

Feito isso, o SoapUI irá carregar as informações lidas a partir da *WSDL* informada e já será possível testar manualmente o funcionamento da *WebService*. A Figura 52 apresenta a interface gerada, com o exemplo de um serviço criado (DAIBERT; TOLEDO; ARAúJO, 2008).

Nesse exemplo, o usuário deve informar um valor em quilômetros e será retornado esse valor convertido em milhas. A requisição do exemplo pode ser visualizada no SoapUI, por meio do *Request* 1, Figura 52. Essa opção permite submeter requisições ao WebService. Para isso, basta alterar o *XML* diretamente, inserindo um valor de entrada na tag *vEntrada*, no lugar de "?". Ao inserir o valor desejado basta clicar no botão que indica uma seta verde, localizado no menu superior da aba *Request* 1. Esse botão irá subme-

Figura 52 – Exemplo de requisição.



Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

ter o envelope ao *WebSevice* e irá gerar uma nova janela onde será exibido o resultado (DAIBERT; TOLEDO; ARAúJO, 2008).

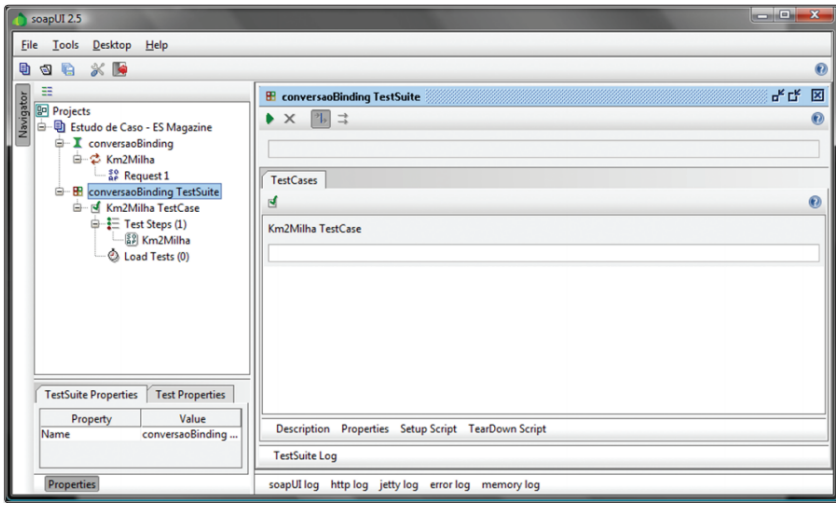
O próximo passo é a criação do ambiente de teste para que eles sejam executados de forma automática. Ele é chamado pelo SoapUI de *Test Suite*. Para criar esse ambiente de teste, deve-se acionar o menu de contexto clicando sobre a opção *conversaoBinding* e invocar *Generate Test Suit*. Uma interface para a criação de um *Test Suite* é então exibida, sendo possível selecionar as operações a serem testadas, para este caso a opção *Km2Milha*. Feito isso, seleciona-se a opção *"One TestCase for Each Operations"*, responsável por criar um caso de teste para cada operação, automaticamente. Após esta operação é criado o *TestSuite*, como o caso de teste para a operação *Km2Milha*. O *TestSuite* criado é mostrado na Figura 53 (DAIBERT; TOLEDO; ARAúJO, 2008).

O próximo passo é abrir a interface de execução e configuração do teste, onde é possível configurar o valor que será submetido ao *WebService* e saber qual será o valor esperado. Para abri-la, clica-se duas vezes em *Km2Milha* no item *TestSteps* da árvore de opções. Na figura 54, é possível visualizar as configurações feitas com o valor 100, adicionado na tag *vEntrada*, para ser submetido. O valor esperado também deve ser informado através da opção *"Adds na Assertion on this item"* e em seguida a opção *"Contains"* (DAIBERT; TOLEDO; ARAúJO, 2008).

Feito isso deve-se executar o teste, clicando no mesmo botão verde anterior. O resultado obtido, pode ser visualizado na figura 55. Como pode ser observado, um erro no teste foi detectado pelo SoapUI apresentando como resultado 310 milhas para a solicitação e não 62,14 como deveria ter sido informado. O erro é destacado apresentando os resultados em vermelho (DAIBERT; TOLEDO; ARAúJO, 2008).

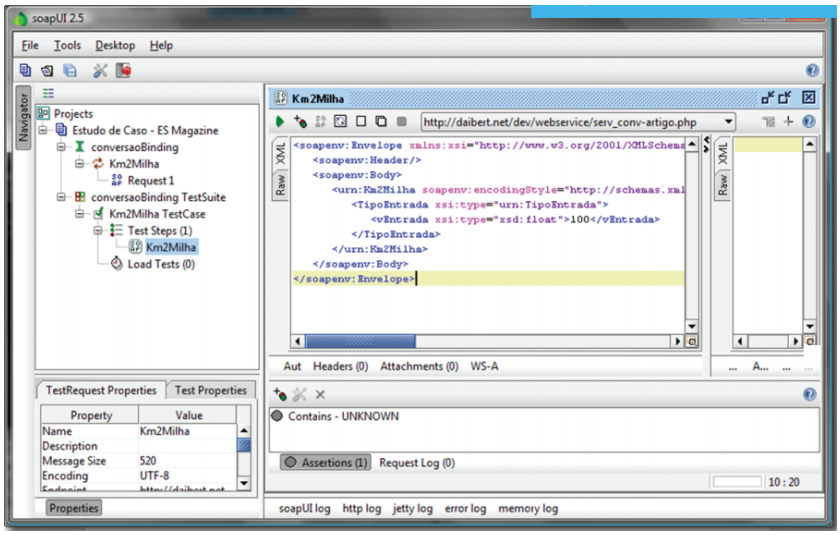
Uma tela com o caso de teste executado com sucesso, é mostrada na Figura 56,

Figura 53 – Criação do TestSuite.



Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

Figura 54 – Asserts e configurações do caso de teste.

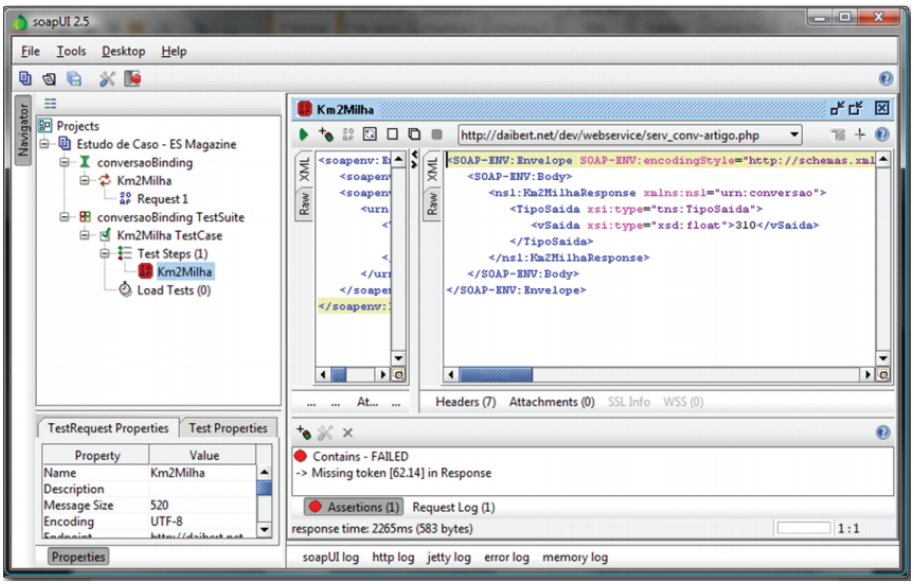


Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

após as correções feitas no código fonte do WebService.

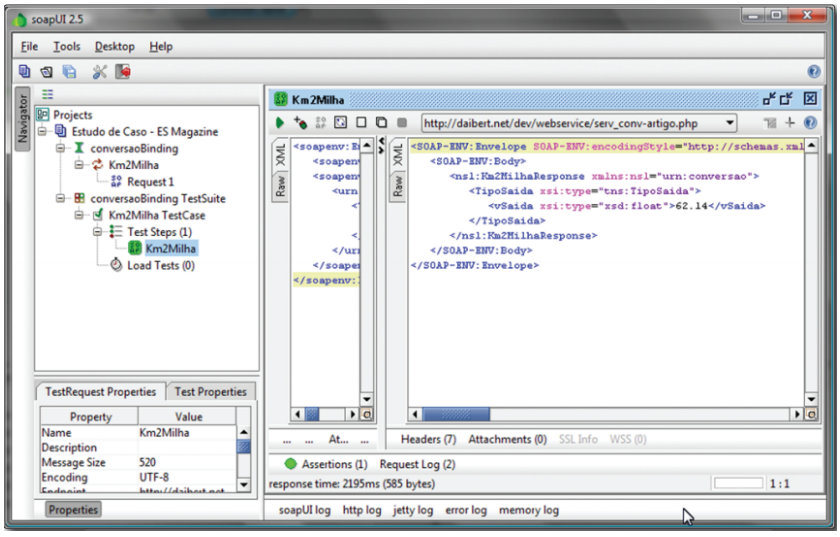
É possível criar a quantidade de casos de testes necessária, pois o SoapUI disponibiliza uma opção para executar vários casos de teste ao mesmo tempo, otimizando o processo de automatização de testes. Esse recurso pode ser obtido, acionando o menu de contexto em *TestSteps* e selecionado a opção *Show Test Case Editor* (DAIBERT; TOLEDO; ARAúJO, 2008).

Figura 55 – Erro após execução do teste.



Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

Figura 56 – Caso de teste executado com sucesso.



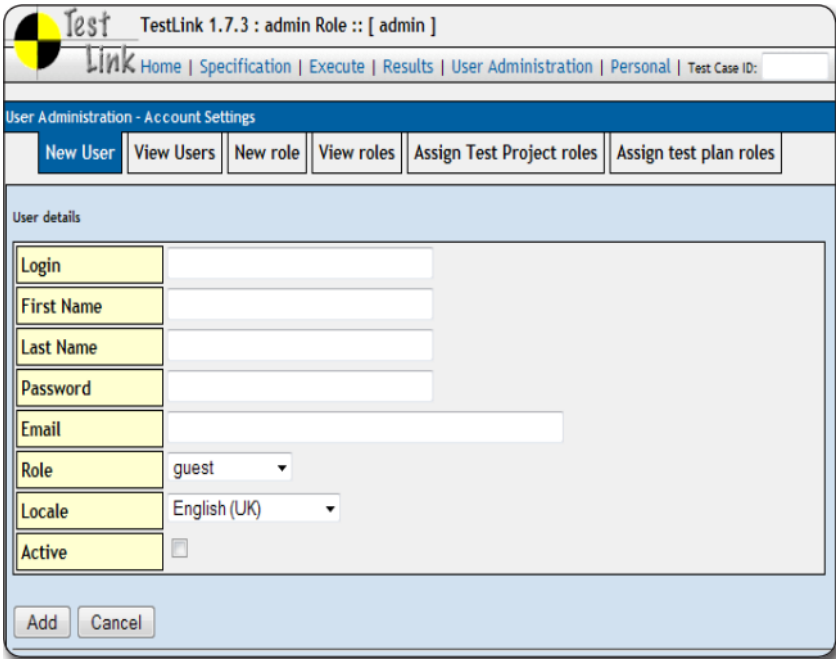
Fonte: (DAIBERT; TOLEDO; ARAúJO, 2008).

4.2.7 TestLink

A ferramenta TestLink é gratuita, desenvolvida na linguagem *PHP* e trata-se de uma aplicação *Web* que pode ser utilizada com qualquer servidor que o projeto em teste já utiliza. Entre suas funcionalidades estão o gerenciamento dos requisitos e planos de teste online, o controle de execução e relatórios dos resultados de testes (PATUCI, 2011).

A tela inicial do TestLink na versão 1.7.3 pode ser visualizada na Figura 57. Para iniciar o processo de teste, deve-se inicialmente cadastrar usuários e perfis necessários para a utilização do sistema.

Figura 57 – Tela de detalhes do usuário.



Fonte: (PATUCI, 2011)

Após a criação dos usuários e perfis necessários, pelo menos um projeto de testes precisa ser cadastrado pelo administrador do sistema, para que todo conteúdo de testes seja administrado dentro dele. Na figura 58 é possível visualizar como criar um novo projeto. Os campos *Name* e *RelatedNotes* devem ser preenchidos com o nome do Projeto de Testes e uma breve descrição de seu conteúdo, respectivamente. O campo *Enable Requirements Functionality*, servirá como um link entre o projeto e os requisitos que podem ser cadastrados futuramente na própria ferramenta (PATUCI, 2011).

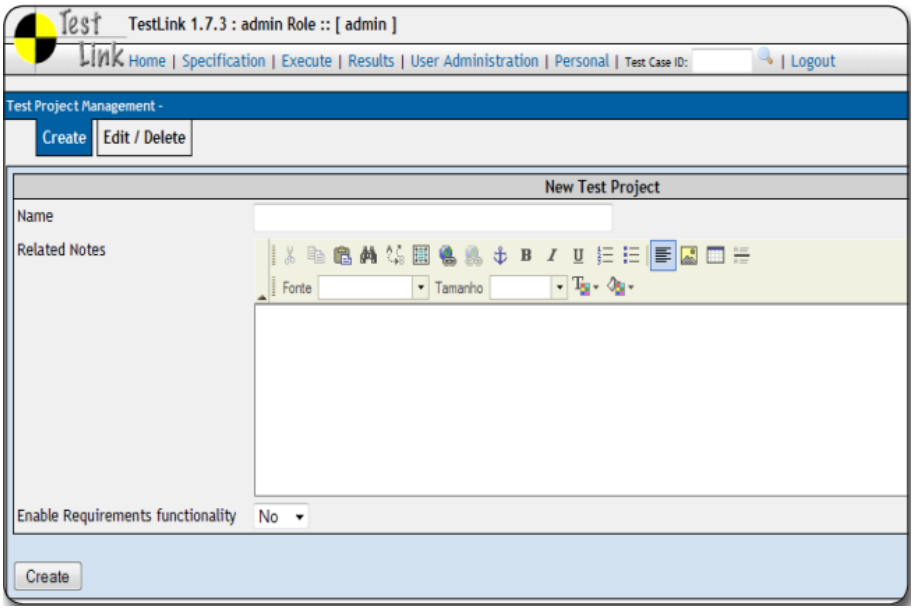
O administrador deve ainda criar uma *baseline*, que trata-se de um controlador de versões para todo o conteúdo do projeto, ver figura 58. A opção *Active*, é responsável por disponibilizar o *baseline* para a utilização do TestLink, caso ela fique desativada não é listada nas páginas de execução e relatórios. E a opção *Open*, é responsável por permitir que os resultados do teste possam ser modificados para a *baseline* (PATUCI, 2011).

Após a criação do projeto e da nova *baseline*, o sistema irá abrir, automaticamente, a tela inicial, onde o usuário poderá realizar o projeto de testes. Mas antes dessa etapa, é necessário a criação de um Plano de Testes, como é mostrado na figura 58. Os campos *Name* e *Description* devem ser preenchidos cuidadosamente, pois eles serão identificados no futuro (PATUCI, 2011).

O próximo passo é cadastrar os requisitos que serão utilizados no planejamento de testes, Figura 61. A criação dos requisitos não é obrigatória, mas pode ser muito útil para o gerenciamento de conteúdo e a criação dos casos de teste (PATUCI, 2011).

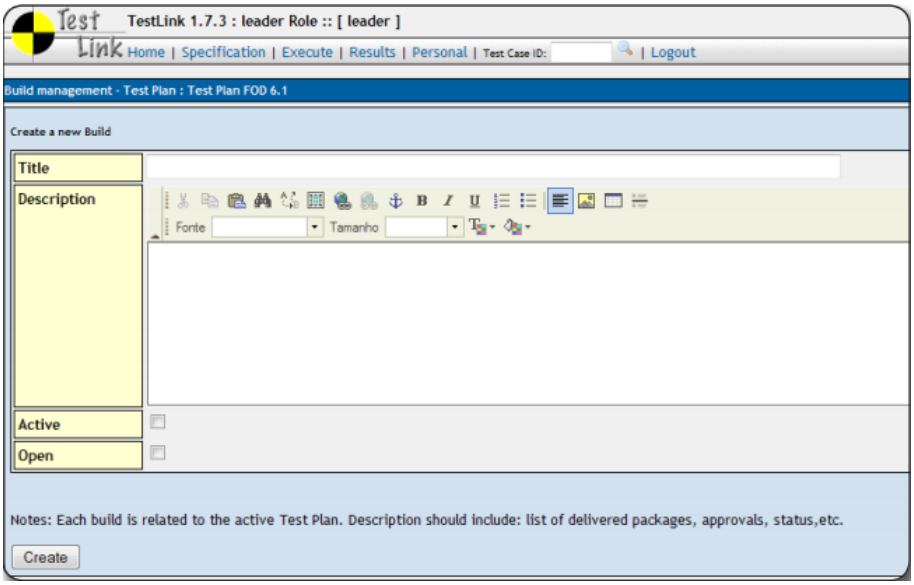
A próxima etapa consiste no desenvolvimento e cadastro dos casos de teste, como

Figura 58 – Tela de criação de novo projeto.



Fonte: (PATUCI, 2011)

Figura 59 – Tela de Nova Baseline.

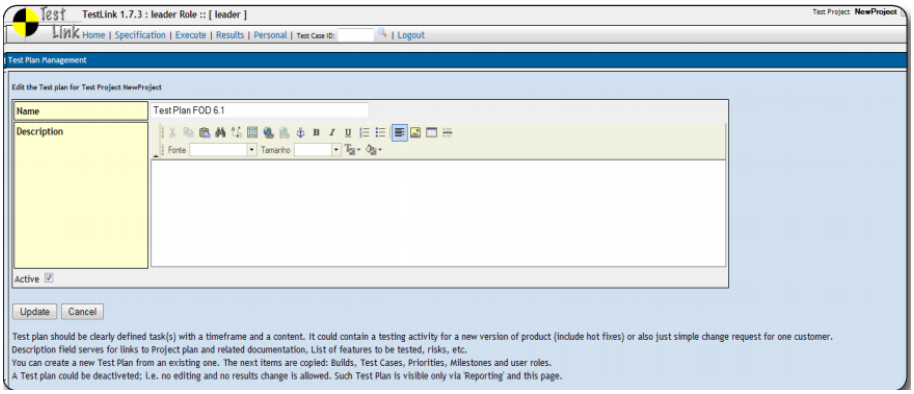


Fonte: (PATUCI, 2011)

pode ser visualizado na Figura 38. Nesta tela é possível cadastrar os casos de teste e o passo a passo a ser executado. Deve-se preencher os campos referentes ao título do caso de teste, a descrição e os Pré-Requisitos, corretamente. Após finalizar todas etapas acima, os testes poderão ser executados e os relatórios poderão se gerados (PATUCI, 2011).

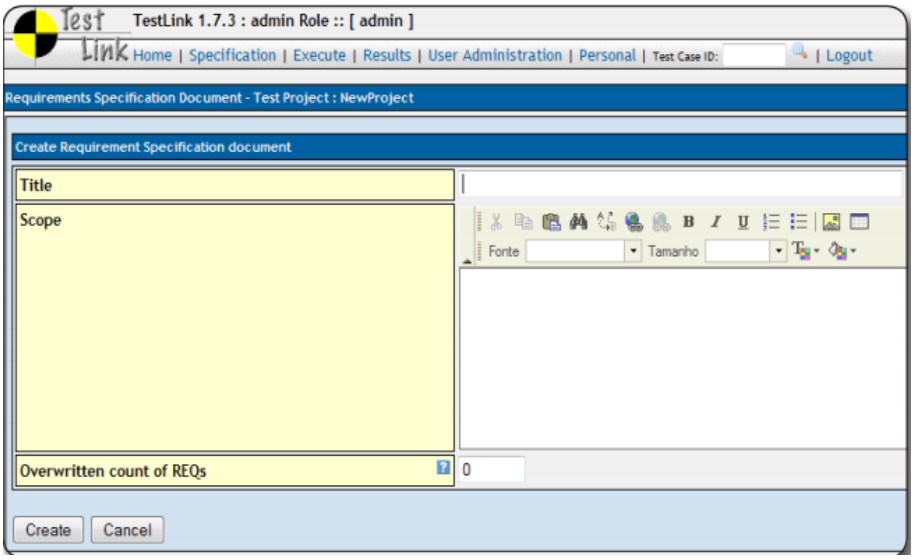
O TestLink também pode ser integrado a outras ferramentas de teste, com o intuito de tornar um processo de teste ágil e sem custos (COLLINS; LOBAO, 2011).

Figura 60 – Tela de novo plano de teste.



Fonte: (PATUCI, 2011)

Figura 61 – Tela de especificação de requisitos.



Fonte: (PATUCI, 2011)

Collins e Lobao (2011) buscam mostrar, por meio do uso de ferramentas *open-source* de teste automático, como agilizar as atividades de teste de regressão dentro de um projeto de software que utiliza a metodologia Scrum de desenvolvimento. Para isso são utilizadas, juntamente com o TestLink, as ferramentas Selenium, apresentada na seção 4.2.5, e Mantis, uma ferramenta *Open Source* para registro e controle de defeitos. Com a integração entre essas ferramentas pode-se ter a rastreabilidade entre caso de teste e defeito e caso de teste e *script* de teste automático.

Segundo Collins e Lobao (2011), apesar das dificuldades encontradas em ajustar atividades de configuração, desenvolvimento e programação de *scripts* de teste, os resultados positivos agregaram valor, pois as ferramentas *Open Source* não gastam recursos financeiros do projeto, apenas demandam tempo de aprendizagem que vale a pena investir.

5 Análise e discussão dos resultados

Este capítulo apresenta um resumo das técnicas e ferramentas encontradas na pesquisa e faz uma análise sobre os resultados obtidos, destacando as respostas encontradas para as questões de pesquisa levantadas na seção 1.1.1.

5.1 Resultados da pesquisa

A partir da busca realizada seguindo os métodos apresentados no capítulo 2, segue abaixo, no quadro 5.1, os resultados totais obtidos nas bases científicas, utilizando a *string* de busca que gerou melhores resultados. Optou-se pela escolha das *string* de busca que ao serem utilizadas em conjunto, retornaram resultados mais enxutos e relevantes ao tema. Isso porque ao se formar as *string* de busca com as palavras-chave de forma dividida e não sistematizada, foram obtidos resultados tão amplos que seriam inviáveis até para aplicar os filtros de seleção. A *string* de busca utilizada foi *"Black-box testing + Functional software testing"*. São discriminados no quadro o número de publicações por ano e a quantidade de artigos selecionados após cada filtro de seleção.

Quadro 5.1 – Resultado da busca "Black-box testing + Functional software testing".

Black-box testing + Functional software testing - 2000/2016																		
Bases	2016	2015	2014	2013	2012	2011	2010	2009	2008	2007	2006	2005	2004	2003	2002	2001	2000	Total
ACM	2	6	9	5	7	6	7	9	4	7	4	3	1	5	0	0	3	78
1º filtro	1	4	7	2	3	4	7	3	2	4	4	3	1	3	0	0	1	49
2º filtro	1	2	6	1	2	4	6	2	2	2	4	3	1	3	0	0	1	40
3º filtro	1	1	1	0	2	1	2	0	1	2	3	0	2	1	0	0	1	18
IEEE	3	21	24	25	21	24	35	20	15	13	7	5	8	3	7	5	3	239
1º filtro	1	15	9	14	10	10	24	14	13	8	6	3	2	1	5	4	0	139
2º filtro	1	11	7	10	7	8	15	10	10	6	6	2	2	0	4	3	0	102
3º filtro	1	6	4	5	7	5	7	4	7	3	4	2	1	0	2	0	1	59
SD	1	0	1	2	0	0	2	1	1	1	0	3	0	0	0	0	0	12
1º filtro	1	0	0	1	0	0	1	0	0	0	0	3	0	0	0	0	0	6
2º filtro	1	0	0	1	0	0	1	0	0	0	0	2	0	0	0	0	0	5
3º filtro	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
Total	6	27	34	32	28	30	44	30	20	21	11	11	9	8	7	5	6	329
Total - 3º Filtro	3	7	5	6	9	6	9	4	8	5	7	2	3	1	2	0	2	79

Fonte: Elaborada pelo autor.

Foi encontrado um total de 329 artigos a partir da *string* de busca utilizada e após a aplicação dos três filtros de seleção, restaram um total de 79 artigos para leitura. Os 68 artigos que foram descartados após o 2º filtro de seleção, podem ser encontrados no Apêndice A.

Após a leitura dos artigos, foi aplicado o critério de notas apresentado na seção 2.1.3, o resultado obtido pode ser visualizado no quadro 5.2. Devido à grande quantidade de artigos obtidos com notas superiores a três, deu-se preferência aos artigos de nota 5, por abordarem assuntos mais específicos sobre técnicas e ferramentas relacionadas a geração de casos de teste ou a execução dos testes funcionais. Dessa forma, foi obtido um total de 12 artigos. Destes 12 artigos 2 foram eliminados por abordar o mesmo tema e 1 foi eliminado por abordar sobre uma ferramenta já citada em outro artigo. Assim, 9 artigos foram definitivamente utilizados para compor o presente trabalho, cada um abordando uma técnica ou ferramenta diferenciada. Os demais artigos que foram descartados após a aplicação do critério de notas, estão disponibilizados no Apêndice B.

A base *Sciente Direct* pouco contribuiu para a construção desse trabalho. Como pode ser observado, foram encontrados poucos resultados sobre o tema e nenhum deles foi escolhido definitivamente, após a aplicação do critério de notas.

Quadro 5.2 – Quantidade de artigos por critério de notas nas bases científicas

Base/Notas	**1**	**2**	**3**	**4**	**5**
ACM	9	1	1	2	5
IEEE	3	5	8	36	7
Sience Direct	0	0	2	0	0
Total	**12**	**6**	**11**	**38**	**12**

Fonte: Elaborada pelo autor.

A maioria dos artigos encontrados nas bases de pesquisa científicas se referem mais à geração de casos de testes ou automação de casos de teste para facilitar o teste funcional, do que a execução do próprio teste funcional em si.

Foram obtidos resultados mais focados à ferramentas de automação de teste funcional, nos artigos publicados na revista de Engenharia de Software Magazine, onde são apresentadas a utilização de algumas ferramentas de teste funcional, como o Selenium, Abbot Framework e TestLink. As palavras-chave utilizadas podem ser visualizadas na seção 2.1.1. No quadro 5.3, são apresentados os artigos encontrados nas revistas publicadas pela Engenharia de Software Magazine e suas respectivas edições.

Todos os 27 artigos das revistas passaram nos filtros de seleção e foram selecionados para leitura. Como o objetivo principal deste trabalho é fazer um levantamento sobre as ferramentas de automação de testes funcionais e a maioria dos artigos publicados nas bases se referiam a geração de casos de teste, deu-se preferência aos artigos das revistas que descreviam o funcionamento dessas ferramentas ou incluíam algum conteúdo relevante para composição do texto. No total 13 artigos das revistas foram selecionados para compor este trabalho.

Quadro 5.3 – Busca realizada na revista Engenharia de Software Magazine.

Resultado da busca: revista Engenharia de Software Magazine		
Edição	Quantidade	Titulo do artigo
1	1	Teste de software
3	1	A importância dos testes automatizados
4	1	Introdução a automação de teste
5	1	Melhores Práticas na Automação de Testes
6	1	Planejamento de Testes a partir de Casos de Uso
8	1	Testes Automatizados de Software em Web Services
11	1	Introdução a teste de software
15	1	Manutenção: Desafios para os Testes numa Fábrica de Software
16	1	Teste funcional utilizando o Abbot Framework
22	1	Experiência em Automação de Testes
24	3	Automatizando Testes Funcionais em Aplicações Web
		Testes de segurança em aplicações Web
		Testes funcionais automatizados com Hudson e Selenium RC
29	2	Automação dos Testes: um lobo na pele de cordeiro?
		Processo e automação de testes de Software
37	1	Ferramentas de teste de software
43	1	Teste de regressão ágil com integração de ferramentas open source de teste
45	1	Testes funcionais de software
58	1	Desafios e benefícios da automação de testes
63	1	Automação de testes funcionais para aplicações da plataforma Android
79	1	Automação de teste de software com QF-Test
82	3	Levantamento de requisitos de software com Canvas
		Automação de testes funcionais
		Geração de relatórios de teste com o Selenium Evidence
83	2	Testes funcionais com a ferramenta QF-Test
		Automação de testes funcionais com Selenium WebDriver utilizando Java
84	1	Trabalhando com o TestLink
Total de artigos	**27**	

Fonte: Elaborada pelo autor.

5.2 Análise das pesquisas

A partir dos resultados foi possível detectar algumas ferramentas para execução de testes funcionais, tanto *Open Source* quanto comercial. O mercado voltado para o desenvolvimento de técnicas de apoio aos testes está em constante crescimento, concomitantemente ao aumento de complexidade dos softwares atuais. Contudo, uma dificuldade muito vivenciada pelos profissionais de teste ao utilizar essas ferramentas é a falta de recursos para auxiliar na geração dos casos de teste que serão executados por elas, devido ao grande número de funcionalidades existentes. Por essa razão é notável o grande acervo de pesquisas de cunho acadêmico, voltado para automatização da geração desses casos de teste. Além das ferramentas AutoBlackTest, LBTest, MoMuT::UML e Taxi voltadas para a geração automática de casos de teste, são apresentadas técnicas para gerá-los de forma mais prática, reduzí-los ou até mesmo priorizar os casos de maior importância.

Como a revista Engenharia de Software Magazine aborda temas mais técnicos, voltados para o âmbito profissional, levando em conta que os autores possuem algum vínculo com empresas da área de software e convivem com o ambiente de testes no dia-a-dia da empresa, foram encontrados resultados mais direcionados ao uso de ferramentas de automação de testes funcionais. Já as bases cientifícas trouxeram um questionamento muito amplo sobre as dificuldades encontradas ao utilizar essas ferramentas, apresentando possíveis soluções para sucumbí-las. Uma solução apresentada e que está disponibilizada para uso é a ferramenta MoMuT::UML, que tem como objetivo a geração de casos de teste automatizadas baseadas em modelos. Ela já foi utilizada em diferentes aplicações industriais a partir de sistemas embarcados, gerando resultados satisfatórios.

Através das informações obtidas com o levantamento bibliográfico, percebeu-se que o teste funcional é muito importante para ser ter uma garantia maior a respeito da validação do software final e este esteja de acordo com os padrões de qualidade que o cliente espera. Já que o mesmo testa a parte que o usuário realmente irá interagir. O teste funcional pode cobrir falhas como a detecção de funcionalidades incorretas, um comportamento inesperado ao executar determinada ação, informações incorretas ou ausentes na tela ou algum erro na interface do sistema. Essa tarefa possibilita ao testador, encontrar bugs do software que não estão de acordo com os requisitos do sistema, diminuindo bugs de interface e aumentando a confiança. Quando bem planejados e executados, os testes funcionais constituem de um passo muito importante para a obtenção de qualidade do software. Com isso a QP1, que visava saber se os testes funcinais podem contribuir para a obtenção de qualidade, pôde ser respondida.

Contudo, é importante destacar que esse tipo de teste pode ser muito custoso dependendo da extensão do software, quando realizado de forma manual. Essa é a vantagem de se utilizar as ferramentas de automação de testes funcionais, que podem auxiliar o desenvolvedor de diversas maneiras, proporcionando um menor custo e esforço, maior ra-

pidez na execução dos testes, além de uma maior cobertura de testes. Com TestLink, por exemplo, é possível gerenciar o plano de teste, escrever casos de teste, cadastrar resultados, além de gerar relatórios de execução e uma documentação do plano de teste. As ferramentas Abbot Framework, Marathon, Selenium IDE e QF-Test, são capazes de realizar testes funcionais automatizados a partir da técnica de *Capture-and-Replay*, onde elas capturam as ações realizadas pelo usuário na interface do sistema em teste e criam *Scripts* de testes que podem ser gravados e reexecutados. Com o MobileTest é possível testar *Websites* de dispositivos móveis inteligentes, a partir de eventos sensíveis ao contexto. A ferramenta SoapUI além de realizar testes funcionais, também é capaz de realizar teste de carga e teste de stress. Além disso, essas ferramentas podem ser integradas, de forma a se obter uma maior eficiência e produtividade. O TestLink, por exemplo, ao ser integrado com as ferramentas Selenium IDE e Mantis, ferramenta para registro e controle de defeitos, agiliza o processo de teste de regressão dentro de um projeto de software, quando é utilizada a metodologia Scrum, que possui uma documentação pouco detalhada. A partir da descrição dessas ferramentas a QP3, que procurava saber sobre a existência de ferramentas para automatizar os testes funcionais e a QP4, que visava saber sobre as vantagens de utilizá-las, puderam ser respondidas.

As ferramentas de testes funcionais são de grande apoio para alcançar um resultado esperado, em busca de um sistema livre de erros e com qualidade, mas isso não é o suficiente. Elas podem detectar falhas, em casos de testes repetitivos e que possuem uma interface mais estável. Mudanças de interface muito dinâmicas, acabam influenciando nos resultados dos *Scripts* de teste gerados, e os mesmos acabam tendo que ser alterados. Além disso, apesar de se ter em mãos essas tecnologias como auxílio, os testes manuais não devem ser abolidos completamente. Existem casos em que os resultados apenas podem ser avaliados por intervenção humana. Detalhes de erros que precisam de uma avaliação minuciosa antes de ser aprovado. Por isso, é importante que o profissional responsável por executar os testes saiba detectar quais funcionalidades poderão ser automatizadas e quais não poderão e deve se ter em mente um plano de teste bem definido, para não haver erros e inconsistências. As ferramentas de automação de testes funcionais não se tratam apenas da técnica de *Capture and Replay* (IZAC, 2015), os testes devem ser trabalhados anteriormente e a ferramenta deve ser escolhida de acordo com o que se deseja testar. A partir dessas observações, é possível perceber que o teste funcional pode ser realizado de forma automatizada, mas existem casos que devem ser feitos de forma manual, respondendo assim a QP2, que desejava saber como o teste funcional pode ser realizado.

Tendo em vista os pontos discutidos acima, conclui-se que as questões levantadas na seção 1.1.1 puderam ser respondidas. Vale ressaltar que nem todas as ferramentas citadas foram testadas para se ter uma resposta mais precisa sobre a eficiência das mesmas.

5.3 Descrição das técnicas e ferramentas abordadas

Para uma análise mais simplificada das técnicas e ferramentas descritas nas seções 4.1 e 4.2, elas foram discriminadas nos quadros 5.4 e 5.5 de forma breve e resumida, destacando-se as principais características de cada uma.

Quadro 5.4 – Técnicas e ferramentas para geração de casos de teste.

Ferramenta / metodologia	**Tipo**	**Descrição**
AutoBlackTest	Ferramenta para a geração de casos de teste	Ferramenta para a geração automática de casos de teste em aplicações interativas. Usa o aprendizado por reforço para aprender como interagir com o aplicativo em teste e estimular suas funcionalidades.
EEOCP	Abordagem para a geração de casos de teste	Abordagem para reduzir o número de casos de teste utilizando o particionamento de classe de equivalência.
Framework para geração de dados de teste a partir da especificação de negócios	Framework para a geração de dados de teste	Estrutura para a geração de dados de teste para teste de caixa-preta, a partir de uma especificação de negócios.
LBTest	Ferramenta para a geração de casos de testes	Ferramenta baseada em algoritmos de aprendizagem para testes de sistemas reativos.
MoMuT::UML	Ferramenta para a geração de casos de teste	Ferramenta de automatização para a geração de casos de teste, baseada na mutação de um modelo do sistema em teste - MBMT (teste de mutação baseado em modelo). Disponível para download em <https://momut.org/>
NNBBBT	Sistema para priorizar casos de teste	Sistema baseado em Redes Neurais que prioriza os casos de teste de maior importância.
TAXI	Ferramenta para a geração de casos de teste	Ferramenta de automatização para a geração de casos de teste. Utiliza a abordagem de teste de partição baseada em XML, para gerar instâncias XML automáticas.
Teste baseado em modelo para aplicação WEB	Abordagem para a geração de casos de teste a partir de um modelo	Abordagem para geração de casos de teste com base no modelo de uma aplicação Web, criado com um alto grau de automatização durante o processo.

Fonte: Elaborada pelo autor.

Quadro 5.5 – Ferramentas de Automação de Testes de Software.

Ferramenta	**Licença**	**Descrição**	**Url**
Abbot Framework	*Open Source*	Ferramenta para teste de aplicativos Java GUI. Facilita a geração de ações do usuário e a verificação do estado do componente.	<https://sourceforge.net/projects/abbot/>
Marathon	*Open Source*	Ferramenta de automatização de testes funcionais. Testa aplicativos desenvolvidos na plataforma Java Swing	<https://marathontesting.com/>
MobileTest	Comercial	Ferramenta automática de teste de caixa preta para dispositivos móveis inteligentes, baseada em eventos sensíveis ao contexto.	<http://mobiletest.me/>
QF-Test	Comercial	Ferramenta de automação para testes de software em aplicações Java e Web com Interfaces Gráfica.	<https://www.qfs.de/en.html>
Selenium IDE	*Open Source*	Ambiente de desenvolvimento integrado para os testes com Selenium. Foi desenvolvido como uma extensão do que grava e reproduz as interações do usuário com o navegador.	<http://www.seleniumhq.org/download/>
SoapUI	*Open Source* e Comercial	Ferramenta para teste de Web-Services, desenvolvido na linguagem Java. Pode realizar testes de carga, testes de stress e testes funcionais	<https://www.soapui.org/downloads/soapui.html>
TestLink	*Open Source*	Ferramenta de gerenciamento de teste baseada na web. O aplicativo fornece especificação de teste, planos de teste e execução, relatórios, especificação de requisitos e colabora com rastreadores de bug bem conhecidos.	<https://sourceforge.net/projects/testlink/>

Fonte: Elaborada pelo autor.

6 Considerações finais e trabalhos futuros

A execução dos testes funcionais é uma grande aliada para auxiliar na obtenção de um software qualificado às condições estabelecidas pelo cliente. Quanto maior e mais eficiente forem os dados de entrada avaliados pelos testes, melhor será o resultado pela busca de erros. Mas devido a grande extensão dos softwares atuais, essa é uma tarefa bastande árdua e demorada se realizada de forma manual. É essa a razão de se procurar por meios que facilitem essa tarefa, através de ferramentas que automatizem a execução de testes e até mesmo a geração dos dados de entrada a serem testados. No entanto, são poucos os trabalhos técnicos-científicos que abordam os testes de software, deixando os futuros profissionais pouco informados sobre assunto. Dessa forma, este estudo apresenta um levantamento bibliográfico para verificar a existência de ferramentas de automação de testes funcionais, em três bases de pesquisa e em uma revista técnica de grande referência na área tecnológica. Contribuindo assim para a área de desenvolvimento de teste de software e servindo como parte de um acervo de pesquisa para futuras fontes de consultas acadêmicas, dentre os profissionais da área.

A partir das buscas realizadas, percebeu-se a existência de diversas ferramentas comerciais disponibilizadas para uso e suas principais características. Apesar da maioria delas estarem voltadas para a geração de *Scripts* de teste, através da técnica de *Capture-and-Replay*, onde as ações do usuário são gravadas e reexecutadas, cada uma têm a sua qualidade e suas limitações. A escolha da ferramenta a ser utilizada é um ponto essencial e não deve ser descartada. Elas apresentam um layout bem intuitivo e fácil de usar, mas não basta apenas rodar a ferramenta e começar a executar os testes de um sistema em teste. Para se obter um resultado produtivo e eficiente, deve-se ter em mente quais funcionalidades serão necessárias testar e quais devem ser priorizadas e elaborar um bom plano de teste com detalhes referentes a essas funcionalidades, como os casos de teste de entradas possíveis e as saídas esperadas. Por isso é importante se ter uma documentação com a especificação dos requisitos.

Uma grande dificuldade é criar todos os casos de teste possíveis de entrada, devido as inúmeras funções exercidas por um software. Essa tarefa costuma ser executada de forma manual, por existir pouca solução automatizada. Existem algumas técnicas para minimizar os casos de teste, que são às vezes seguidas até intuitivamente, como a o particionamento de classes de equivalência, a análise do valor limite e a tabela de decisão. Mesmo essas técnicas sendo de grande utilidade, não são o suficiente quando se trata de sistemas muito complexos, com inúmeras entradas possíveis. Pesquisas para solicionar estes desafios estão em constante crecimento. O MoMuT::UML é uma solução já disponibilizada para uso, mas que não pode ser usado para todos os casos possíveis.

Não é possível saber qual técnica ou ferramenta apresentada é a melhor ou ideal para ser utilizada. Na prática, cada empresa tem uma visão diferenciada e introduz o seu próprio método e ferramenta mais apropriados. Por isso é importante se ter um bom conhecimento sobre as técnicas de testes de software, além de boas estratégias de teste.

Existem outras ferramentas para automação de testes funcionais, como o Quick Test Professional, WinRunner, Rational Functional Tester, Robot, Canoo Webtest e Mantis que não foram abordadas, seja porque foram apenas citadas nos artigos lidos ou por não apresentarem detalhes suficientes para serem descritos. Seria interessante a realização de um estudo sobre o funcionamento das mesmas e também de outras ferramentas com o mesmo fim, como as ferramentas de interface gráfica citadas tabela 3.2 que não foram mencionadas neste trabalho.

Este estudo fez um levantamento bibliográfico para verificar a existência de ferramentas de automação de testes funcionais, mas não realizou uma comparação para se ter uma visão mais ampla e objetiva sobre a qualidade e eficiência de cada ferramenta. Por isso, como sugestão para trabalhos futuros, recomenda-se fazer uma seleção e análise comparativa das ferramentas encontradas. E para se ter uma resposta mais direcionada ao âmbito comercial, seria interessante realizar uma pesquisa de mercado sobre o uso das ferramentas desse tipo pelas empresas de software da região. Seria também um trabalho futuro completar o levantamento bibliográfico aqui presente como os artigos de nota 4 descartados.

Referências

AICHERNIG, B. et al. Momut::uml model-based mutation testing for uml. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION, 8., 2015, Graz. [S.l.]: IEEE, 2015. p. 1–8. Citado nas páginas 43 e 44.

ALMEIDA, F. N. de; RIBEIRO, V. V.; ARAÚJO, M. A. P. Testes funcionais automatizados com hudson e selenium rc. In: *DevMedia*. 24. ed. [S.l.]: Engenharia de Software Magazine, 2010. p. 57–62. Citado nas páginas 71 e 72.

BENDER. [S.l.: s.n.], 1996. Citado na página 14.

BERLATTO, L. *Teste de software.* 2012. Disponível em: <http://testenext.blogspot.com.br/2012/10/teste-de-software.html>. Acesso em: 12 set. 2016. Citado na página 24.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. In: *DevMedia*. 3. ed. [S.l.]: Engenharia de Software Magazine, 2008. p. 54–57. Citado na página 15.

BERTOLINO, A. et al. Taxi - a tool for xml-based testing. In: ICSE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 29., 2007. *ACM SIGSOFT Software Engineering Notes*. Washington, DC, USA: ACM, 2007. p. 53–54. Citado nas páginas 45, 46, 47 e 48.

BHASIN, H.; KHANNA, E. Neural network based black box testing. In: ACM SIGSOFT SOFTWARE ENGINEERING NOTES, 1., 2014. *ACM SIGSOFT Software Engineering Notes*. New York, NY, USA: ACM, 2014. p. 1–6. Citado nas páginas 44 e 45.

BO, J.; XIANG, L.; XIAOPENG, G. Mobiletest: A tool supporting automatic black box test for software on smart mobile devices. In: INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST, 2., 2007, Minneapolis, MN. [S.l.]: IEEE, 2007. p. 8. Citado nas páginas 61, 62 e 63.

BOEHM. [S.l.: s.n.], 1979. Citado na página 22.

CAETANO, C. Introdução à automação de teste. *Engenharia de software magazine*, DevMedia, v. 4, p. 60–66, 2008. Citado nas páginas 14, 15, 30 e 32.

COLLINS, E.; LOBAO, L. Experiência em automação de testes. In: *DevMedia*. 22. ed. [S.l.]: Engenharia de Software Magazine, 2010. p. 60–66. Citado nas páginas 31, 32 e 34.

COLLINS, E.; LOBAO, L. Processo e automação de testes de software: Trabalhando em ambiente de programação exploratória. In: *DevMedia*. 29. ed. [S.l.]: Engenharia de Software Magazine, 2010. p. 26–33. Citado nas páginas 55, 56, 57, 58, 59 e 60.

COLLINS, E.; LOBAO, L. Teste de regressão ágil com integração de ferramentas open source de teste. In: *DevMedia*. 43. ed. [S.l.]: Engenharia de Software Magazine, 2011. p. 19–25. Citado nas páginas 78 e 79.

COPELAND. [S.l.: s.n.], 2004. Citado nas páginas 28 e 29.

COTA, T. T. *Projeto de jogos móveis para idosos: um estudo com foco na motivação para jogar.* 2014. 190 p. Dissertação (Mestrado em Informática) — Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, 2014. Citado na página 18.

DAIBERT, M. S.; TOLEDO, J. V.; ARAúJO, M. A. P. Testes automatizados de software em web services. In: *DevMedia*. 8. ed. [S.l.]: Engenharia de Software Magazine, 2008. p. 52–59. Citado nas páginas 72, 73, 74, 75 e 76.

Forrester Research Inc. [S.l.: s.n.], 2006. Citado na página 30.

GOMES, F. A importância dos testes para a qualidade do software. *DevMedia*, Canal Engenharia de Software, 2013. Disponível em: <http://www.devmedia.com.br/a-importancia-dos-testes-para-a-qualidade-do-software/28439>. Acesso em: 30 jun. 2016. Citado nas páginas 23 e 24.

IZAC, A. F. eira. Automação de teste de software com qf-test: Planejando testes para utilização a longo prazo. In: *DevMedia*. 79. ed. [S.l.]: Engenharia de Software Magazine, 2015. p. 54–60. Citado nas páginas 63, 64, 65, 66, 67 e 84.

JÚNIOR, F. C. de L. *Algoritmo Q-learning como estratégia de exploração e/ou explotação para as metaheurísticas GRASP e algoritmo genético*. 2009. 140 p. Tese (Doutorado em Ciências) — Universidade Federal do Rio Grande do Norte, 2009. Disponível em: <https://repositorio.ufrn.br/jspui/bitstream/123456789/15129/1/FranciscoCLJ.pdf>. Acesso em: 18 mar. 2017. Citado nas páginas 35 e 36.

MALLLMANN, C. *Tabela de decisão*. 2015. Disponível em: <http://quatest.com.br/Portal/tabela-de-decisao/>. Acesso em: 12 set. 2016. Citado na página 30.

MARIANI, L. et al. Autoblacktest: A tool for automatic black-box testing. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33., 2011. *Proceedings...* New York, NY, USA: ACM, 2011. p. 1013–1015. Citado nas páginas 35 e 36.

MATIELI, L. V.; ARAÚJO, F. O. de. *Estimativa de teste de software: levantamento Bibliográfico em veículos brasileiros e internacionais*. 2015. Disponível em: <http://www.rpep.uff.br/index.php/RPEP/article/view/11/8>. Acesso em: 17 mar. 2017. Citado na página 15.

MEINKE, K.; SINDHU, M. A. Lbtest: A learning-based testing tool for reactive systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION, 6., 2013, Luembourg. [S.l.]: IEEE, 2013. p. 447–454. Citado nas páginas 41, 42 e 43.

MOLINARI. [S.l.: s.n.], 2008. Citado na página 24.

NAPSOL. *Automatização de Teste de Software: com ênfase em Ferramentas Open Source*. 2016. Disponível em: <http://napsol.icmc.usp.br/ats/>. Acesso em: 12 set. 2016. Citado nas páginas 26, 27, 28 e 29.

NAUTIYAL, L.; GUPTA, N.; DIMRI, S. C. A novel approach to component-based software testing. In: ACM SIGSOFT SOFTWARE ENGINEERING NOTES, 1., 2016. *ACM SIGSOFT Software Engineering Notes*. New York, NY, USA: ACM, 2016. p. 1–5. Citado nas páginas 25, 34, 36, 37 e 38.

NEGRINI, C. Desafios e benefícios da automação de testes. In: *DevMedia*. 58. ed. [S.l.]: Engenharia de Software Magazine, 2013. p. 38–41. Citado na página 31.

NETO, A. C. D. Planejamento de testes a partir de casos de uso. In: *DevMedia*. 6. ed. [S.l.]: Engenharia de Software Magazine, 2008. p. 36–41. Citado na página 15.

NETO, A. C. D. Automatizando testes funcionais em aplicações web. In: *DevMedia*. 24. ed. [S.l.]: Engenharia de Software Magazine, 2010. p. 49–56. Citado nas páginas 14, 25, 67, 68, 69, 70, 71 e 72.

PATUCI, G. de O. Ferramentas de teste de software: Como elas podem auxiliar nas atividades do dia a dia. In: *DevMedia*. 37. ed. [S.l.]: Engenharia de Software Magazine, 2011. p. 19–26. Citado nas páginas 76, 77, 78 e 79.

PINTO, T. D. *Uma ferramenta para geração e execução automática de testes funcionais baseados na descrição textual de casos de uso*. 2013. 190 p. Tese (Mestrado em Informática) — Pontifícia Universidade Católica Do Rio De Janeiro, Rio de Janeiro, 2013. Disponível em: <http://www.maxwell.vrac.puc-rio.br/Busca_etds.php?strSecao=resultado&nrSeq=24924@1>. Acesso em: 12 set. 2016. Citado nas páginas 14 e 15.

PRESSMAN, R. S. *Engenharia de software: Uma abordagem profissional*. 7. ed. Porto Alegre, Brasil: AMGH, 2011. Citado nas páginas 14, 22, 23, 25, 26 e 27.

SOMMERVILLE, I. *Engenharia de software*. 9. ed. São Paulo, Brasil: Pearson, 2011. Citado nas páginas 14, 22 e 31.

SOUZA, K. P. de; GASPAROTTO, A. M. S. A importância da atividade de teste no desenvolvimento de software. *ENEGEP*, XXXIII Encontro nacional de engenharia de producao, 2013. Disponível em: <http://www.abepro.org.br/biblioteca/enegep2013_TN_STO_177_007_23030.pdf>. Acesso em: 12 set. 2016. Citado nas páginas 22 e 24.

SOUZA, V. R. de; VALE, R. C.; ARAÚJO, M. A. P. Teste funcional utilizando o abbot framework: Automatizando testes em aplicações java desktop. In: *DevMedia*. 16. ed. [S.l.]: Engenharia de Software Magazine, 2008. p. 51–56. Citado nas páginas 49, 50, 51, 52, 53 e 54.

TORSEL, A.-M. Automated test case generation for web applications from a domain specific model. In: ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE WORKSHOPS, 35., 2011, Munich. [S.l.]: IEEE, 2011. p. 137–142. Citado nas páginas 34, 47 e 49.

WANG, F.; YANG, X.; ZHU, X. An adaptive framework for test data generation from business specification. In: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY AND COMPUTER SCIENCE, 2009. *IEEE Computer Society*. Hangzhou, China: IEEE, 2009. p. 581–584. Citado nas páginas 35, 38, 39, 40 e 41.

WILLY ROBSON, V. R. E. R. S. Testes de software - parte 01. *DevMedia*, Canal Engenharia de Software, 2008. Disponível em: <http://www.abepro.org.br/biblioteca/enegep2013_TN_STO_177_007_23030.pdf>. Acesso em: 30 jun. 2016. Citado nas páginas 24 e 25.

APÊNDICE A – ARTIGOS DESCARTADOS APÓS FILTROS DE SELEÇÃO

Este apêndice apresenta os 68 artigos das bases científicas que foram descartados a parir do 2º filtro de seleção, descrito na seção 2.1.3. As referências estão no formato gerado pelas bases referentes.

Base ACM

1. A. C. Rajeev, Prahladavaradan Sampath, K. C. Shashidhar, and S. Ramesh. 2010. CoGenTe: a tool for code generator testing. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). ACM, New York, NY, USA, 349-350. DOI=10.1145/1858996.1859070 <http://doi.acm.org/10.1145/1858996.1859070>

2. Ajitha Rajan. 2006. Automated requirements-based test case generation. SIGSOFT Softw. Eng. Notes 31, 6 (November 2006), 1-2. DOI=<http://dx.doi.org/10.1145/1218776.1218799>

3. Alessandro Oliveira Arantes, Nandamudi Lankalapalli Vijaykumar, Valdivino Alexandre de Santiago Junior, and Danielle Guimarães. 2008. WEB-PerformCharts: a collaborative web-based tool for test case generation from statecharts. In Proceedings of the 10th International Conference on Information Integration and Web-based Applications and Services (iiWAS '08), Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil (Eds.). ACM, New York, NY, USA, 374-381. DOI=<http://dx.doi.org/10.1145/1497308.1497375>

4. Antonia Bertolino. 2009. Approaches to testing service-oriented software systems. In Proceedings of the 1st international workshop on Quality of service-oriented software systems (QUASOSS '09). ACM, New York, NY, USA, 1-2. DOI=<http://dx.doi.org/10.1145/1596473.1596475>

5. Antonia Bertolino and Stefania Gnesi. 2003. Use case-based testing of product lines. In Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11). ACM, New York, NY, USA, 355-358. DOI=<http://dx.doi.org/10.1145/940071.940120>

6. Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. 2009. Whitening SOA testing. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE '09). ACM, New York, NY, USA, 161-170. DOI=<http://dx.doi.org/10.1145/1595696.1595721>

7. Chao-Chun Yeh, Shih-Kun Huang, and Sung-Yen Chang. 2013. A black-box based android GUI testing system. In Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys '13). ACM, New York, NY, USA, 529-530. DOI=<http://dx.doi.org/10.1145/2462456.>

8. Christian Schwarzl and Bernhard Peischl. 2010. Generation of executable test cases based on behavioral UML system models. In Proceedings of the 5th Workshop on Automation of Software Test (AST '10). ACM, New York, NY, USA, 31-34. DOI=<http://dx.doi.org/10.1145/1808266.1808271>

9. Francis Akowuah, Jerrisa Lake, Xiaohong Yuan, Emmanuel Nuakoh, and Huiming Yu. 2015. Testing the security vulnerabilities of OpenEMR 4.1.1: a case study. J. Comput. Sci. Coll. 30, 3 (January 2015), 26-35.

10. Harsh Bhasin. 2014. Artificial life and cellular automata based automated test case generator. SIGSOFT Softw. Eng. Notes 39, 1 (February 2014), 1-5. DOI=<http://dx.doi.org/10.1145/2557833.2557843>

11. Jiangyuan Yao, Zhiliang Wang, Xia Yin, Xingang Shi, Jianping Wu, and Yahui Li. 2014. Model Based Black-Box Testing of SDN Applications. In Proceedings of the 2014 CoNEXT on Student Workshop (CoNEXT Student Workshop '14). ACM, New York, NY, USA, 37-39. DOI=<http://dx.doi.org/10.1145/2680821.2680828>

12. Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. 2005. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In Proceedings of the 5th ACM international conference on Embedded software (EMSOFT '05). ACM, New York, NY, USA, 299-306. DOI=<http://dx.doi.org/10.1145/1086228.1086283>

13. Markus Clermont and David Parnas. 2005. Using information about functions in selecting test cases. In Proceedings of the 1st international workshop on Advances in model-based testing (A-MOST '05). ACM, New York, NY, USA, 1-7. DOI=<http://dx.doi.org/10.1145/1082983.1083276>

14. Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. 2010. Environment modeling with UML/MARTE to support black-box system testing for real-time embedded systems: methodology and industrial case studies. In Proceedings of the

13th international conference on Model driven engineering languages and systems: Part I (MODELS'10), Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen (Eds.). Springer-Verlag, Berlin, Heidelberg, 286-300.

15. Peter Steinbacher, Florian Fankhauser, Christian Schanes, and Thomas Grechenig. 2010. Black-box approach for testing quality of service in case of security incidents on the example of a SIP-based VoIP service: work in progress. In Principles, Systems and Applications of IP Telecommunications (IPTComm '10), Georg Carle, Helmut Reiser, Gonzalo Camarillo, and Vijay K. Gurbani (Eds.). ACM, New York, NY, USA, 101-110. DOI=<http://dx.doi.org/10.1145/1941530.1941545>

16. Prasad Bokil, Padmanabhan Krishnan, and R. Venkatesh. 2015. Achieving Effective Test Suites for Reactive Systems using Specification Mining and Test Suite Reduction Techniques. SIGSOFT Softw. Eng. Notes 40, 1 (February 2015), 1-8. DOI=<http://dx.doi.org/10.1145/2693208.2693226>

17. Rubing Huang, Jinfu Chen, Zhicheng Li, Rongcun Wang, and Yansheng Lu. 2014. Adaptive random prioritization for interaction test suites. In Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14). ACM, New York, NY, USA, 1058-1063. DOI: <http://dx.doi.org/10.1145/2554850.2554854>

18. Steffen Mazanek and Christian Rutetzki. 2011. On the importance of model comparison tools for the automatic evaluation of the correctness of model transformations. In Proceedings of the 2nd International Workshop on Model Comparison in Practice (IWMCP '11), Davide Di Ruscio and Dimitris S. Kolovos (Eds.). ACM, New York, NY, USA, 12-15. DOI=<http://dx.doi.org/10.1145/2000410.2000413>

19. Tristan Allwood, Cristian Cadar, and Susan Eisenbach. 2011. High coverage testing of Haskell programs. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, New York, NY, USA, 375-385. DOI=<http://dx.doi.org/10.1145/2001420.2001465>

20. Voin Legourski, Christian Trödhandl, and Bettina Weiss. 2005. A system for automatic testing of embedded software in undergraduate study exercises. SIGBED Rev. 2, 4 (October 2005), 48-55. DOI=<http://dx.doi.org/10.1145/1121812.1121822>

21. Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. 2003. Web application security assessment by fault injection and behavior monitoring. In Proceedings of the 12th international conference on World Wide Web (WWW '03). ACM, New York, NY, USA, 148-159. DOI=<http://dx.doi.org/10.1145/775152.775174>

22. Zhiqiang Zhang and Jian Zhang. 2011. Characterizing failure-causing parameter interactions by adaptive testing. In Proceedings of the 2011 International Symposium

on Software Testing and Analysis (ISSTA '11). ACM, New York, NY, USA, 331-341. DOI=<http://dx.doi.org/10.1145/2001420.2001460>

Base IEEE

1. A. Barrett and D. Dvorak, "A Combinatorial Test Suite Generator for Gray-Box Testing,"2009 Third IEEE International Conference on Space Mission Challenges for Information Technology, Pasadena, CA, 2009, pp. 387-393. doi: 10.1109/SMC-IT.2009.53

2. A. Bertolino, J. Gao, E. Marchetti and A. Polini, "Automatic Test Data Generation for XML Schema-based Partition Testing,"Automation of Software Test , 2007. AST '07. Second International Workshop on, Minneapolis, MN, 2007, pp. 4-4. doi: 10.1109/AST.2007.6

3. A. D. Junior and D. J. Cecilio da Silva, "Code-coverage Based Test Vector Generation for SystemC Designs,"IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07), Porto Alegre, 2007, pp. 198-206. doi: 10.1109/ISVLSI.2007.31

4. A. Patil, M. Proeller, A. Kshirasagar and A. Nahar, "A Framework for Automated Testing of RTL Designs,"2015 International Conference on Computing Communication Control and Automation, Pune, 2015, pp. 989-991. doi: 10.1109/ICCUBEA.2015.195

5. B. Cao, Z. Chen, H. Liu, G. Ma, P. Zhang and G. Peng, "Black Box Testing for Cloud-Based Client Security Software in Network Behaviors,"2013 Fourth International Conference on Networking and Distributed Computing, Los Angeles, CA, 2013, pp. 75-79. doi: 10.1109/ICNDC.2013.9

6. B. Kovacevic, M. Kovacevic, D. Stefanovic and V. Pekovic, "Scenario-based set-top box testing,"2014 22nd Telecommunications Forum Telfor (TELFOR), Belgrade, 2014, pp. 1087-1090. doi:10.1109/TELFOR.2014.7034596

7. C. Lee, B. Lee and C. Wu, "Providing the Guideline of Determining Quality Checklists Priorities Based on Evaluation Records of Software Products,"2008 15th Asia-Pacific Software Engineering Conference, Beijing, 2008, pp. 169-176. doi: 10.1109/APSEC.2008.75

8. C. Ma, C. Du, T. Zhang, F. Hu and X. Cai, "WSDL-Based Automated Test Data Generation for Web Service,"2008 International Conference on Computer Science and Software Engineering, Wuhan, Hubei, 2008, pp. 731-737. doi: 10.1109/CSSE.2008.790

9. D. Marijan, N. Teslic, T. Tekcan and V. Pekovic, "Multimedia system verification through a usage model and a black test box,"The 2010 International Conference

on Computer Engineering Systems, Cairo, 2010, pp. 178-182. doi: 10.1109/ICCES.2010.5674849

10. 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 (TAICPART-MUTATION 2007), Windsor, 2007, pp. 137-137. doi: 10.1109/TAIC.PART.2007.36

11. E. Farchi, Y. Krasny and Y. Nir, "Automatic simulation of network problems in UDP-based Java programs,"18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., 2004, pp. 267-. doi: 10.1109/IPDPS.2004.1303342

12. F. Kantz, T. Ruschival, P. Nenninger and D. Streitferdt, "Testing with Large Parameter Sets for the Development of Embedded Systems in the Automation Domain,"2009 33rd Annual IEEE International Computer Software and Applications Conference, Seattle, WA, 2009, pp. 504-509.doi: 10.1109/COMPSAC.2009.183

13. F. Naseer, S. ur Rehman and K. Hussain, "Using meta-data technique for component based black box testing,"2010 6th International Conference on Emerging Technologies (ICET), Islamabad, 2010, pp. 276-281. doi: 10.1109/ICET.2010.5638474

14. H. M. Sneed and C. Verhoef, "Measuring test coverage of SoA services,"2015 IEEE 9th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA), Bremen, 2015, pp. 59-66. doi: 10.1109/MESOCA.2015.7328128

15. I. Kastelan, V. Pekovic, N. Teslic, T. Tekcan and D. Marijan, "Automatic Black Box Testing of television systems on the final production line,"2011 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, 2011, pp. 869-870. doi: 10.1109/ICCE.2011.5722910

16. J. Chen, G. v. Jourdan, W. Ma and H. Ural, "Improving Coverage in Functional Testing,"2006 Sixth International Conference on Quality Software (QSIC'06), Beijing, 2006, pp. 99-106. doi: 10.1109/QSIC.2006.34

17. Jessica Chen, Hasan Ural, Guy-V. Jourdan, Wenxin Ma, "Improving Coverage in Functional Testing", Quality Software, International Conference on, vol. 00, no. , pp. 99-106, 2006, doi:10.1109/QSIC.2006.34

18. J. G. Hwang, J. H. Baek, H. J. Jo and K. M. Lee, "Applicability test on black-box testing tool of railway signaling system in consideration of the convenience of use,"2014 14th International Conference on Control, Automation and Systems (ICCAS 2014), Seoul, 2014, pp. 1489-1495. doi: 10.1109/ICCAS.2014.6987797

19. J. G. Hwang, J. H. Baek, H. J. Jo and K. M. Lee, "Software black-box testing tool for railway signaling system by real interface,"2013 13th International Conference on Control, Automation and Systems (ICCAS 2013), Gwangju, 2013, pp. 508-511. doi:10.1109/ICCAS.2013.6703987

20. Jin-Hui Shan, Ji Wang and Zhi-Chang Qi, "On path-wise automatic generation of test data for both white-box and black-box testing,"Proceedings Eighth Asia-Pacific Software Engineering Conference, 2001, pp. 237-240.

21. K. K. Mohan, A. K. Verma and A. Srividya, "Software reliability estimation through black box and white box testing at prototype level,"2010 2nd International Conference on Reliability, Safety and Hazard - Risk-Based Technologies and Physics-of-Failure Methods (ICRESH), Mumbai, 2010, pp. 517-522. doi: <10.1109/ICRESH.2010.5779604>

22. Lilan Wu, Bo Liu, Yi Jin and Xiaoyao Xie, "Using back-propagation neural networks for functional software testing,"2008 2nd International Conference on Anti-counterfeiting, Security and Identification, Guiyang, 2008, pp. 272-275.

23. M. Á. B. Júnior, F. B. de Lima Neto and J. C. S. Fort, "Improving black box testing by using neuro-fuzzy classifiers and multi-agent systems,"2010 10th International Conference on Hybrid Intelligent Systems, Atlanta, GA, 2010, pp. 25-30. doi:10.1109/HIS.2010.5600020

24. M. Fischer and R. Tönjes, "Generating test data for black-box testing using genetic algorithms,"Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012), Krakow, 2012, pp. 1-6. doi: 10.1109/ETFA.2012.6489789

25. M. Iglewski, "Automatic Testing of SCR Specifications,"2006 Canadian Conference on Electrical and Computer Engineering, Ottawa, Ont., 2006, pp. 2455-2459. doi: 10.1109/CCECE.2006.277546

26. N. Li, Z. Li and L. Zhang, "Mining Frequent Patterns from Software Defect Repositories for Black-Box Testing,"2010 2nd International Workshop on Intelligent Systems and Applications, Wuhan, 2010, pp. 1-4. doi: 10.1109/IWISA.2010.5473578

27. N. Li, Z. Li and X. Sun, "Classification of Software Defect Detected by Black-Box Testing: An Empirical Study,"2010 Second World Congress on Software Engineering, Wuhan, 2010, pp. 234-240. doi: 10.1109/WCSE.2010.28

28. N. Sasikaladevi and L. Arockiam, "Correctness evaluation model for composite web service,"3rd International Conference on Trendz in Information Sciences Computing (TISC2011), Chennai, 2011, pp. 188-193. doi: 10.1109/TISC.2011.6169112

29. P. J. Schroeder, P. Faherty and B. Korel, "Generating expected results for automated black-box testing,"Proceedings 17th IEEE International Conference on Automated Software Engineering„ 2002, pp. 139-148. doi: 10.1109/ASE.2002.1115005

30. R. P. Tan, P. Nagpal and S. Miller, "Automated Black Box Testing Tool for a Parallel Programming Library,"2009 International Conference on Software Testing Verification and Validation, Denver, CO, 2009, pp. 307-316. doi: 10.1109/ICST.2009.32

31. R. Setiadi and M. F. Lau, "Identifying Data Inconsistencies Using After-State Database Testing (ASDT) Framework,"2014 14th International Conference on Quality Software, Dallas, TX, 2014, pp. 105-110. doi: 10.1109/QSIC.2014.39

32. S. Beydeda and V. Gruhn, "Integrating white- and black-box techniques for class-level regression testing,"25th Annual International Computer Software and Applications Conference. COMPSAC 2001, Chicago, IL, 2001, pp. 357-362. doi: 10.1109/CMPSAC.2001.960639

33. S. Liu and S. Nakajima, "A "Vibration"Method for Automatically Generating Test Cases Based on Formal Specifications,"2011 18th Asia-Pacific Software Engineering Conference, Ho Chi Minh, 2011, pp. 73-80. doi: 10.1109/APSEC.2011.16

34. S. Liu, "Automatic Specification-Based Testing: Challenges and Possibilities,"2011 Fifth International Conference on Theoretical Aspects of Software Engineering, Xi'an, Shaanxi, 2011, pp. 5-8. doi: 10.1109/TASE.2011.36

35. S. Kukolj, V. Marinkovic, M. Popovic and S. Bognár, "Selection and Prioritization of Test Cases by Combining White-Box and Black-Box Testing Methods,"2013 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems, Budapest, 2013, pp. 153-156. doi: 10.1109/ECBS-EERC.2013.28

36. S. Wieczorek and A. Stefanescu, "Improving Testing of Enterprise Systems by Model-Based Testing on Graphical User Interfaces,"2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, Oxford, 2010, pp. 352-357. doi: 10.1109/ECBS.2010.59

37. S. Wu, Y. Wu and S. Xu, "Acceleration of Random Testing for Software,"2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing, Vancouver, BC, 2013, pp. 51-59. doi: 10.1109/PRDC.2013.15

38. T. Farah, D. Alam, M. A. Kabir and T. Bhuiyan, "SQLi penetration testing of financial Web applications: Investigation of Bangladesh region,"2015 World Congress on Internet Security (WorldCIS), Dublin, 2015, pp. 146-151.

39. T. Murnane and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique,"Proceedings 2001 Australian Software Engineering Conference, Canberra, ACT, 2001, pp. 12-20. doi: 10.1109/ASWEC.2001.948492

40. T. Noguchi, H. Washizaki, Y. Fukazawa, A. Sato and K. Ota, "History-Based Test Case Prioritization for Black Box Testing Using Ant Colony Optimization,"2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, 2015, pp. 1-2. doi: 10.1109/ICST.2015.7102622

41. "Using functional view of software to increase performance in defect testing,"Students Conference, 2002. ISCON '02. Proceedings. IEEE, Lahore, Pakistan, 2002, pp. 11-11. doi:10.1109/ISCON.2002.1214599

42. W. Tian, J. Xu, K. M. Lian, Y. Zhang and J. f. Yang, "Research on mock attack testing for SQL injection vulnerability in multi-defense level web applications,"The 2nd International Conference on Information Science and Engineering, Hangzhou, China, 2010, pp. 1-5. doi: 10.1109/ICISE.2010.5689924

43. Z. Franjcic and M. Fjeld, "Automated Component Testing of Evolutionary Software: An Industry Example,"2015 41st Euromicro Conference on Software Engineering and Advanced Applications, Funchal, 2015, pp. 463-463. doi: 10.1109/SEAA.2015.71

Base Science Direct

1. Antti Kervinen, Pablo Virolainen, Heuristics for Faster Error Detection With Automated Black Box Testing, Electronic Notes in Theoretical Computer Science, Volume 111, 2005, Pages 53-71, ISSN 1571-0661, <http://dx.doi.org/10.1016/j.entcs.2004.12.007>.

2. Bernhard Schätz, Christian Pfaller, Integrating Component Tests to System Tests, Electronic Notes in Theoretical Computer Science, Volume 260, 2010, Pages 225-241, ISSN 1571-0661, <http://dx.doi.org/10.1016/j.entcs.2009.12.040>.

3. Giuseppe Scollo, Silvia Zecchini, Architectural Unit Testing, Electronic Notes in Theoretical Computer Science, Volume 111, 2005, Pages 27-52, ISSN 1571-0661, <http://dx.doi.org/10.1016/j.entcs.2004.12.006>.

APÊNDICE B – ARTIGOS DESCARTADOS APÓS CRITÉRIO DE NOTAS

Este apêndice apresenta os 70 artigos das bases científicas que passaram pelos três filtros de seleção, descritos na seção 2.1.3, mas foram descartados após aplicar os critérios de notas. As referências estão no formato gerado pelas bases referentes.

Base ACM

1. Avik Sinha and Carol Smidts. 2006. HOTTest: A model-based test design technique for enhanced testing of domain-specific applications. ACM Trans. Softw. Eng. Methodol. 15, 3 (July 2006), 242-278. DOI=http://dx.doi.org/10.1145/1151695.1151697

2. Harsh Bhasin and Harleen Kaur. 2014. Toward a secured automated test-data generator using S-Box. SIGSOFT Softw. Eng. Notes 39, 5 (September 2014), 1-5. DOI=http://dx.doi.org/10.1145/2659118.2659127

3. Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy oracle: a system for finding application leaks with black box differential testing. In Proceedings of the 15th ACM conference on Computer and communications security (CCS '08). ACM, New York, NY, USA, 279-288. DOI=http://dx.doi.org/10.1145/1455770.1455806

4. Jeong Seok Kang and Hong Seong Park. 2012. Web-based automated black-box testing framework for component based robot software. In Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12). ACM, New York, NY, USA, 852-859. DOI=http://dx.doi.org/10.1145/2370216.2370410

5. Karl Meinke. 2004. Automated black-box testing of functional correctness using function approximation. In Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '04). ACM, New York, NY, USA, 143-153. DOI=http://dx.doi.org/10.1145/1007512.1007532

6. Koen Claessen, John Hughes, Michał Pałka, Nick Smallbone, and Hans Svensson. 2010. Ranking programs using black box testing. In Proceedings of the 5th Workshop on Automation of Software Test (AST '10). ACM, New York, NY, USA, 103-110.

7. Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. 2006. Coverage metrics for requirements-based testing. In Proceedings of the 2006

international symposium on Software testing and analysis (ISSTA '06). ACM, New York, NY, USA, 25-36. DOI=http://dx.doi.org/10.1145/1146238.1146242

8. Miguel A. Francisco and Laura M. Castro. 2012. Automatic generation of test models and properties from UML models with OCL constraints. In Proceedings of the 12th Workshop on OCL and Textual Modelling (OCL '12). ACM, New York, NY, USA, 49-54. DOI: https://doi.org/10.1145/2428516.2428525 DOI=http://dx.doi.org/10.1145/180826

9. Patrick J. Schroeder and Bogdan Korel. 2000. Black-box test reduction using input-output analysis. In Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00), Mary Jean Harold (Ed.). ACM, New York, NY, USA, 173-177. DOI=http://dx.doi.org/10.1145/347324.349042

10. Peter M. Kruse, Joachim Wegener, and Stefan Wappler. 2010. A cross-platform test system for evolutionary black-box testing of embedded systems. SIGEVOlution 5, 1 (May 2010), 3-9. DOI=http://dx.doi.org/10.1145/1811155.1811156

11. Petros Papadopoulos and Neil Walkinshaw. 2015. Black-box test generation from inferred models. In Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE '15). IEEE Press, Piscataway, NJ, USA, 19-24.

12. Sezer Gören and F. Joel Ferguson. 2006. Test sequence generation for controller verification and test with high coverage. ACM Trans. Des. Autom. Electron. Syst. 11, 4 (October 2006), 916-938. DOI=http://dx.doi.org/10.1145/1179461.1179467

13. Y. T. Yu, Pak-Lok Poon, S. P. Ng, and T. Y. Chen. 2003. On the use of the classification-tree method by beginning software testers. In Proceedings of the 2003 ACM symposium on Applied computing (SAC '03). ACM, New York, NY, USA, 1123-1127. DOI: https://doi.org/10.1145/952532.952751

Base IEEE

1. A. M. Torsel, "Automated Test Case Generation for Web Applications from a Domain Specific Model,"2011 IEEE 35th Annual Computer Software and Applications Conference Workshops, Munich, 2011, pp. 137-142. doi: 10.1109/COMPSACW.2011.32

2. A. Rajan, "Coverage Metrics to Measure Adequacy of Black-Box Test Suites,"21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, 2006, pp. 335-338. doi: 10.1109/ASE.2006.31

3. A. Shahbazi and J. Miller, "Black-Box String Test Case Generation through a Multi-Objective Optimization,"in IEEE Transactions on Software Engineering, vol. 42, no. 4, pp. 361-378, April 1 2016. doi: 10.1109/TSE.2015.2487958

4. B. Qu, C. Nie, B. Xu and X. Zhang, "Test Case Prioritization for Black Box Testing,"31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Beijing, 2007, pp. 465-474. doi: 10.1109/COMPSAC.2007.209

5. C. C. Yeh and S. K. Huang, "CovDroid: A Black-Box Testing Coverage System for Android,"2015 IEEE 39th Annual Computer Software and Applications Conference, Taichung, 2015, pp. 447-452. doi: 10.1109/COMPSAC.2015.125

6. C. Fu, M. Grechanik and Q. Xie, "Inferring Types of References to GUI Objects in Test Scripts,"2009 International Conference on Software Testing Verification and Validation, Denver, CO, 2009, pp. 1-10. doi: 10.1109/ICST.2009.12

7. D. Agarwal, D. E. Tamir, M. Last and A. Kandel, "A Comparative Study of Artificial Neural Networks and Info-Fuzzy Networks as Automated Oracles in Software Testing,"in IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, vol. 42, no. 5, pp. 1183-1193, Sept. 2012. doi: 10.1109/TSMCA.2012.2183590

8. D. Sinnig, F. Khendek and P. Chalin, "A Formal Model for Generating Integrated Functional and User Interface Test Cases,"2010 Third International Conference on Software Testing, Verification and Validation, Paris, 2010, pp. 255-264. doi: 10.1109/ICST.2010.56

9. E. Fong, R. Gaucher, V. Okun, P. E. Black and E. Dalci, "Building a Test Suite for Web Application Scanners,"Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008), Waikoloa, HI, 2008, pp. 478-478. doi: 10.1109/HICSS.2008.79

10. F. Belli and C. J. Budnik, "Towards self-testing of component-based software,"29th Annual International Computer Software and Applications Conference (COMPSAC'05), 2005, pp. 205-210 Vol. 1. doi: 10.1109/COMPSAC.2005.158

11. H. Freeman, "Software testing,"in IEEE Instrumentation Measurement Magazine, vol. 5, no. 3, pp. 48-50, Sep 2002. doi: 10.1109/MIM.2002.1028373

12. H. Jin, Y. Wang, N. W. Chen, S. Wang and L. M. Zeng, "Predication of Program Behaviours for Functionality Testing,"2009 First International Conference on Information Science and Engineering, Nanjing, 2009, pp. 4993-4996.

13. H. Wu, "An effective equivalence partitioning method to design the test case of the WEB application,"2012 International Conference on Systems and Informatics (ICSAI2012), Yantai, 2012, pp. 2524-2527. doi: 10.1109/ICSAI.2012.6223567

14. J. Jainae and T. Suwannasart, "A Tool for Test Case Impact Analysis of Database Schema Changes Using Use Cases,"2014 International Conference on Information Science Applications (ICISA), Seoul, 2014, pp. 1-4. doi: 10.1109/ICISA.2014.6847346

15. Kabsu Han, Insick Son and Jeonghun Cho, "A study on test automation of IVN of intelligent vehicle using model-based testing,"2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), Da Nang, 2013, pp. 123-128. doi: 10.1109/ICUFN.2013.6614794

16. Kwang Ik Seo and Eun Man Choi, "Comparison of Five Black-box Testing Methods for Object-Oriented Software,"Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06), Seattle, WA, 2006, pp. 213-220. doi: 10.1109/SERA.2006.22

17. L. C. Briand, Y. Labiche and Z. Bawar, "Using Machine Learning to Refine Black-Box Test Specifications and Test Suites,"2008 The Eighth International Conference on Quality Software, Oxford, 2008, pp. 135-144. doi: 10.1109/QSIC.2008.5

18. L. Christophe, R. Stevens, C. D. Roover and W. D. Meuter, "Prevalence and Maintenance of Automated Functional Tests for Web Applications,"2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 141-150.

19. L. Mariani, M. Pezzè, O. Riganelli and M. Santoro, "AutoBlackTest: a tool for automatic black-box testing,"2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, 2011, pp. 1013-1015. doi: 10.1145/1985793.1985979

20. L. Mariani, M. Pezze, O. Riganelli and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications,"2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, 2012, pp. 81-90. doi: 10.1109/ICST.2012.88

21. M. A. Khan and M. Sadiq, "Analysis of black box software testing techniques: A case study,"The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11), Dubai, 2011, pp. 1-5. doi: 10.1109/CTIT.2011.6107931

22. M. Alawairdhi and E. Aleisa, "A Scenario-Based Approach for Requirements Elicitation for Software Systems Complying with the Utilization of Ubiquitous Computing Technologies,"2011 IEEE 35th Annual Computer Software and Applications Conference Workshops, Munich, 2011, pp. 341-344. doi: 10.1109/COMPSACW.2011.63

23. M. Grechanik, Q. Xie and C. Fu, "Creating GUI Testing Tools Using Accessibility Technologies,"2009 International Conference on Software Testing, Verification, and Validation Workshops, Denver, CO, 2009, pp. 243-250. doi: 10.1109/ICSTW.2009.31

24. M. Hong, Q. Zeng, Z. Wang, Y. Zhang and H. Wang, "An Approach of Automated Test Cases Generation in Database Stored Procedure Testing,"2010 Second International Workshop on Education Technology and Computer Science, Wuhan, 2010, pp. 533-537. doi: 10.1109/ETCS.2010.554

25. M. Sharma and S. C. B., "Automatic Generation of Test Suites from Decision Table - Theory and Implementation,"2010 Fifth International Conference on Software Engineering Advances, Nice, 2010, pp. 459-464. doi: 10.1109/ICSEA.2010.78

26. N. Sasikaladevi and L. Arockiam, "Correctness evaluation model for composite web service,"3rd International Conference on Trendz in Information Sciences Computing (TISC2011), Chennai, 2011, pp. 188-193. doi: 10.1109/TISC.2011.6169112

27. P. Accioly, P. Borba and R. Bonifácio, "Comparing Two Black-Box Testing Strategies for Software Product Lines,"2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse, Natal, 2012, pp. 1-10. doi: 10.1109/SBCARS.2012.17

28. P. Fenkam, H. Gall and M. Jazayeri, "Constructing CORBA-supported oracles for testing: a case study in automated software testing,"Proceedings 17th IEEE International Conference on Automated Software Engineering„ 2002, pp. 129-138.

29. Qing Xie, Mark Grechanik and Chen Fu, "REST: A tool for reducing effort in script-based testing,"2008 IEEE International Conference on Software Maintenance, Beijing, 2008, pp. 468-469. doi: 10.1109/ICSM.2008.4658108

30. R. Chaipraserth, A. Leelasantitham and S. Kiattisin, "A test automation framework in POCT system using TDD techniques,"2013 13th International Symposium on Communications and Information Technologies (ISCIT), Surat Thani, 2013, pp. 600-604. doi: 10.1109/ISCIT.2013.6645931

31. R. Chopra and S. Madan, "Reusing black box test paths for white box testing of websites,"2013 3rd IEEE International Advance Computing Conference (IACC), Ghaziabad, 2013, pp. 1345-1350. doi: 10.1109/IAdCC.2013.6514424

32. R. Dudila and I. A. Letia, "Towards combining functional requirements tests and unit tests as a preventive practice against software defects,"2013 IEEE 9th International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, 2013, pp. 279-282. doi: 10.1109/ICCP.2013.6646121

33. R. Wang, Y. Xu and Y. Xiang, "Research and Realization of WEB Security Auto-Testing Tool Based on AHP,"2010 International Conference on Computational Intelligence and Software Engineering, Wuhan, 2010, pp. 1-4. doi: 10.1109/CISE.2010.5676792

34. R. S. S. Filho, C. J. Budnik, W. M. Hasling, M. McKenna and R. Subramanyan, "Supporting Concern-Based Regression Testing and Prioritization in a Model-Driven Environment,"2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, Seoul, 2010, pp. 323-328. doi: 10.1109/COMPSACW.2010.63

35. R. Tommy, N. Prasannakumaran, K. V. Sumithra and S. Kesav, "Test scenario modeling: Modeling test scenarios diagrammatically using specification based testing techniques,"Proceedings of the 2015 Third International Conference on Computer, Communication, Control and Information Technology (C3IT), Hooghly, 2015, pp. 1-7. doi: 10.1109/C3IT.2015.7060198

36. S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee and A. M. Memon, "Lightweight Static Analysis for GUI Testing,"2012 IEEE 23rd International Symposium on Software Reliability Engineering, Dallas, TX, 2012, pp. 301-310. doi: 10.1109/ISSRE.2012.25

37. S. Benli, A. Habash, A. Herrmann, T. Loftis and D. Simmonds, "A Comparative Evaluation of Unit Testing Techniques on a Mobile Platform,"2012 Ninth International Conference on Information Technology - New Generations, Las Vegas, NV, 2012, pp. 263-268. doi: 10.1109/ITNG.2012.45

38. S. Beydeda, V. Gruh and M. Stachorski, "A graphical class representation for integrated black- and white-box testing,"Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, Florence, 2001, pp. 706-715.

39. S. Iftikhar, M. Z. Iqbal, M. U. Khan and W. Mahmood, "An automated model based testing approach for platform games,"2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), Ottawa, ON, 2015, pp. 426-435. doi: 10.1109/MODELS.2015.7338274

40. S. Kimoto, T. Tsuchiya and T. Kikuno, "Pairwise Testing in the Presence of Configuration Change Cost,"2008 Second International Conference on Secure System Integration and Reliability Improvement, Yokohama, 2008, pp. 32-38.

41. S. Noikajana and T. Suwannasart, "Web Service Test Case Generation Based on Decision Table (Short Paper),"2008 The Eighth International Conference on Quality Software, Oxford, 2008, pp. 321-326. doi: 10.1109/QSIC.2008.7

42. S. Pei, B. Wu, K. Zhu and Q. Yu, "Novel software automated testing system based on J2EE,"in Tsinghua Science and Technology, vol. 12, no. S1, pp. 51-56, July 2007. doi: 10.1016/S1007-0214(07)70083-4

43. S. P. G. and H. Mohanty, "Automated Scenario Generation Based on UML Activity Diagrams,"2008 International Conference on Information Technology, Bhubaneswar, 2008, pp. 209-214. doi: 10.1109/ICIT.2008.52

44. S. Rosiello, A. Choudhary, A. Roy and R. Ganesan, "Combining Black Box Testing with White Box Code Analysis: A Heterogeneous Approach for Testing Enterprise SaaS Applications,"2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, 2014, pp. 359-364. doi: 10.1109/ISSREW.2014.113

45. T. Murnane, K. Reed and R. Hall, "Tailoring of black-box testing methods,"Australian Software Engineering Conference (ASWEC'06), Sydney, NSW, 2006, pp. 8 pp.-299. doi: 10.1109/ASWEC.2006.49

46. T. Murnane, R. Hall and K. Reed, "Towards describing black-box testing methods as atomic rules,"29th Annual International Computer Software and Applications Conference (COMPSAC'05), 2005, pp. 437-442 Vol. 2. doi: 10.1109/COMPSAC.2005.157

47. Tsuyoshi Yumoto, Tohru Matsuodani, Kazuhiko Tsuda, "A study on an approach for analysing test basis using I/O test data patterns", 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), vol. 00, no. , pp. 1-8, 2015, doi:10.1109/ICSTW.2015.7107429

48. T. Y. Chen and Pak-Lok Poon, "Experience with teaching black-box testing in a computer science/software engineering curriculum,"in IEEE Transactions on Education, vol. 47, no. 1, pp. 42-50, Feb. 2004. doi: 10.1109/TE.2003.817617

49. W. H. Allen, C. Dou and G. A. Marin, "A Model-based Approach to the Security Testing of Network Protocol Implementations,"Proceedings. 2006 31st IEEE Conference on Local Computer Networks, Tampa, FL, 2006, pp. 1008-1015. doi: 10.1109/LCN.2006.322216

50. W. Liu, X. Wu, W. Zhang and Y. Xu, "The research of the test case prioritization algorithm for black box testing,"2014 IEEE 5th International Conference on Software Engineering and Service Science, Beijing, 2014, pp. 37-40. doi: 10.1109/ICSESS.2014.6933509

51. X. Peng and L. Lu, "A new approach for session-based test case generation by GA,"2011 IEEE 3rd International Conference on Communication Software and Networks, Xi'an, 2011, pp. 91-96.

52. Y. Chen, S. Liu and W. E. Wong, "A Method Combining Review and Testing for Verifying Software Systems,"2008 International Conference on BioMedical Engineering and Informatics, Sanya, 2008, pp. 827-831. doi: 10.1109/BMEI.2008.332

53. Y. T. Hu and N. W. Lin, "Automatic black-box method-level test case generation based on constraint logic programming,"2010 International Computer Symposium (ICS2010), Tainan, 2010, pp. 977-982. doi: 10.1109/COMPSYM.2010.5685369

54. Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo and F. Massacci, "Towards Black Box Testing of Android Apps,"2015 10th International Conference on Availability, Reliability and Security, Toulouse, 2015, pp. 501-510. doi: 10.1109/ARES.2015.70

55. Z. Djuric, "A black-box testing tool for detecting SQL injection vulnerabilities,"2013 Second International Conference on Informatics Applications (ICIA), Lodz, 2013, pp. 216-221. doi: 10.1109/ICoIA.2013.6650259

Base Sience Direct

1. Swapnili P. Karmore, Anjali R. Mahajan, New Approach for Testing and providing Security Mechanism for Embedded Systems, Procedia Computer Science, Volume 78, 2016, Pages 851-858, ISSN 1877-0509, <http://dx.doi.org/10.1016/j.procs.2016.02.073>.

2. Tsuyoshi Yumoto, Toru Matsuodani, Kazuhiko Tsuda, A Test Analysis Method for Black Box Testing Using AUT and Fault Knowledge, Procedia Computer Science, Volume 22, 2013, Pages 551-560, ISSN 1877-0509, <http://dx.doi.org/10.1016/j.procs.2013.09.135>.