CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Desenvolvimento de um *Framework* para aplicação de teste Funcional Orientado a Dados

THALES IAGO PEREIRA BALBO

Marília, 2014

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Desenvolvimento de um *Framework* para aplicação de teste Funcional Orientado a Dados

Trabalho de Conclusão de Curso apresentado ao Centro Universitário Eurípides de Marília – UNIVEM – na cidade de Marília – São Paulo para a obtenção do título de Bacharelado em Sistemas de Informação.
Orientador: Prof. Fábio Lúcio Meira

Marília, 2014

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Tháles Iago Pereira Balbo

Desenvolvimento de um Framework para aplicação de teste
Funcional Orientado a Dados

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Sistemas de Informação.

Nota: 8.0 (OITO)

Orientador: Fabio Lucio Meira

1°. Examinador: Paulo Augusto Nardi

2°. Examinador: Ricardo José Sabatine

Marília, 02 de dezembro de 2014.

Agradecimento

Agradeço primeiramente a meus pais Maria e Valdeci por todo amparo, estrutura dada durante toda minha vida e apoio nos momentos difíceis.

Quero agradecer aos irmãos Tiago e Thanilo por todo apoio e compreensão devido a minha ausência em momentos importantes de família

Quero agradecer ao amigo Edimilson por ter me ajudado no momento mais importante do curso com locomoção até a faculdade, e a amiga Jéssica Freire por todo incentivo e por sempre acreditar em mim.

Aos amigos de faculdade em especial Danniel Covo, Sidney Santo, Brayan Rastelli e Laio por sempre estarem presentes durante cada faze do curso.

Agradeço a cada funcionário da Univem pelos serviços prestados que mantem a faculdade funcionando para que possamos estudar.

Agradecer aos professores Adriano Bezerra, Elton Aquinori Yokomizo, Elvis Fusco, Emerson Alberto Marconato, Fabio Dacencio Pereira, Fabio Lucio Meira, Fabio Piola Navarro, Geraldo Pereira Junior, Giuliana Marega Marques, Leonardo Castro Botega, Mauricio Duarte, Paulo Augusto Nardi, Paulo Rogerio de Mello Cardoso, Renata Aparecida de Carvalho Paschoal, Ricardo José Sabatine, Rodolfo Barros Chiaramonte, Juliana e José Eduardo Santarem por contribuírem para meu crescimento pessoal, acadêmico e profissional.

Ao meu orientador Fabio Lucio Meira, por ter desempenhado seu papel em prol dessa pesquisa, sempre com seu apoio, compartilhando experiências e ensinamentos.

"As nuvens mudam sempre de posição, mas são sempre nuvens no céu. Assim devemos ser todo dia, mutantes, porém leais com o que pensamos e sonhamos; lembre-se, tudo se desmancha no ar, menos os pensamentos".

(Paulo Beleki)

BALBO, Thales Iago Pereira. **Desenvolvimento de um Framework para aplicação de teste Funcional Orientado a Dados.** 62 f. 2014. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) - Centro Universitário Eurípides de Marília, UNIVEM, Marília/SP, 2014.

Resumo

A presente monografia tem como finalidade o desenvolvimento de um *framework* para a elaboração de teste funcional em sistemas web orientado a dados. A área de teste de *software* ainda é pouco difundida entre as empresas de desenvolvimento por diversos motivos no qual podemos citar como um dos principais o tempo gasto na atividade de teste e o custo da atividade de teste no desenvolvimento de *software*. Para melhorar esse cenário são usadas técnicas de desenvolvimento de testes automatizados unitários, funcionais e entre outros, que traz benefícios como tempo gasto para identificação de falhas e defeitos no *software*, garante os testes de regressão, entre outras vantagens. Sendo assim, o trabalho procura responder: Como a criação de um *framework* pode auxiliar no desenvolvimento de casos de teste Funcional orientado a dados em aplicações Web?. Esta investigação se justifica na importância da conscientização de empresas de desenvolvimento sobre as positivas características dos testes de *software*.

Palavras-chave: Teste. Automação. TesteNG. Selenium. *Framework*. Funcional.

BALBO, Thales Iago Pereira. **Desenvolvimento de um Framework para aplicação de teste Funcional Orientado a Dados.** 62 f. 2014. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) - Centro Universitário Eurípides de Marília, UNIVEM, Marília/SP, 2014.

Abstract

The Software Testing search is to show the maximum number of failures with minimal effort. In this perspective, the objective of this work is on developing a framework that aids the development of functional test cases for data-driven Web applications, with the same easy to use for analysts with little programming knowledge. This objective seeks to answer the research problem, based about the following question: How to create a framework can assist in developing functional test cases for data-driven Web applications?. This research is justified on the importance of awareness of development companies on the positive characteristics of software testing.

Keywords Test. Automation. TestNG. Selenium. *Framework*. Functional.

Lista de Figuras

Lista De Tabelas

LISTA DE ABREVIATURAS E SIGLAS

HTML	HyperText Markup Language, em português, Linguagem de Marcação de Hipertexto
IDE	Integrated Development Environment, em português, Ambiente Integrado de Desenvolvimento;
POM	Project Object Model, em português, projeto modelo de objeto;

Sumário

INTRODUÇÃO

Apresenta-se neste momento introdutório o problema de pesquisa, os objetivos a serem alcançados, a motivação junto à justificativa, a metodologia da pesquisa e a organização do trabalho.

Problemática de Pesquisa

A problemática da pesquisa busca responder o seguinte questionamento:
Como a criação de um *framework* pode auxiliar no desenvolvimento de casos de teste Funcional orientado a dados, em aplicações *Web*?

Objetivo Geral

O objetivo geral desse trabalho é o desenvolvimento de um *framework* que auxilia o desenvolvimento de casos de teste Funcional orientado a dados, em aplicações *Web*, esperando que o mesmo seja de fácil utilização para analistas com pouco conhecimento em programação.

Objetivos Específicos

- Desenvolver uma estrutura lógica para o *framework* onde seja de fácil manutenção e geração de planos e casos de teste;
- Disponibilizar ferramentas utilizadas na automação já consolidadas no mercado de automação em um único local de forma simples;
- Facilitar a utilização dos métodos do *Selenium* IDE (*WebDriver*);
- Exemplificar a utilização do conceito de *PageObject* na geração de casos de teste funcionais;
- Diminuir significativamente a quantidade de código gerado na automação de testes funcionais;
- Realizar a integração e exemplificar a utilização de massa de dados para a criação de variações de caso de teste;

Justificativa e Motivação

Esta investigação se justifica na importância de apresentar as características positivas do teste de *software* e automação de teste, através do estudo teórico sobre a área e a apresentação do *framework* proposto.

A motivação para o desenvolvimento desse trabalho é a crescente necessidade de se aplicar testes automatizados de forma simples e ágil no processo de desenvolvimento de *software,* sendo a programação de *scripts* automatizados complexa tendo em vista sua dificuldade. Assim, oferecer com esse *framework* uma ferramenta simples em que qualquer analista de teste poderá utilizar sem grande esforço.

Metodologia

Esta pesquisa tem um perfil descritivo, pois faz análise de conteúdo bibliográfico. Segundo Gil (2002) a pesquisa descritiva trabalha com estudos que procuram determinar opiniões ou projeções futuras nas respostas obtidas, este tipo de pesquisa se baseia na certeza de que os problemas são passíveis de serem resolvidos e as práticas podem ser melhoradas tendo como foco a descrição e análise de observações objetivas e diretas.

Explicando a pesquisa qualitativa, também característica aqui presente, Biklen, S. e Bogdan, R. (1994) afirmam que normalmente é direcionada sem ter como ferramenta instrumentos estatísticos nas análises dos dados. A este tipo de pesquisa cabe adquirir dados descritivos visando um contato direto e com interação do pesquisador com o objeto de estudo.

Para dar sustentação conceitual a esta pesquisa, foi feita uma revisão bibliográfica buscando encontrar conceitos mais plausíveis ao tema abordado.

> A pesquisa bibliográfica é desenvolvida com base em material já elaborado, constituído principalmente de livros e artigos científicos. Embora em quase todos os estudos seja exigido algum tipo de trabalho dessa natureza, há pesquisas desenvolvidas exclusivamente a partir de fontes bibliográficas (GIL, 2002, p. 44).

Ao passo inicial da pesquisa fez-se então uma revisão de literatura, em que buscou identificar:

TESTES DE *SOFTWARE*

TESTE FUNCIONAL: ABORDAGEM CONCEITUAL E REFLEXÃO

AUTOMAÇÃO DE TESTES FUNCIONAIS

Para que se efetive o objetivo descrito no capítulo "*FRAMEWORK* PARA APLICAÇÃO DE TESTE FUNCIONAL ORIENTADO A DADOS: DESENVOLVIMENTO", traceja-se o aporte metodológico, apresentando o *Framework* e a Arquitetura e Funcionamento deste *Framework* Proposto.

Estrutura Organizacional do Trabalho

Esta pesquisa está organizada em introdução e seis capítulos:

Capítulo 1 – Teste de *Software*: Aborda a área geral da pesquisa e todo o seu embasamento no qual surgiu a motivação e objetivo do trabalho, traz os fundamentos da área de teste de *software*, definindo o escopo do trabalho.

Capítulo 2 – Automação de Teste Funcionais: Apresentação de conceitos e técnicas de automação de teste com foco em teste funcionais e suas características, definindo o escopo do trabalho neste contexto.

Capítulo 3 - Teste Funcional Abordagem Conceitual- Apresentação de conceitos aprofundados sobre teste funcionais e suas nomenclaturas e conceitos.

Capítulo 4 - *Framework* Proposto: Apresentação da estrutura do *framework*.

Capítulo 5 - Apresentação da Ferramenta: Apresentação do *framework* e casos práticos realizados.

Capítulo 6 - Conclusão: Conclusões obtidas no trabalho.

CAPÍTULO 1 TESTES DE *SOFTWARE*

O objetivo deste capítulo é mostrar, através de referências bibliográficas, conceitos e objetivo do teste de *software* e suas vertentes, a fim de propor um maior aprofundamento no assunto. Apresentar ideias acerca da temática "Testes de *Software*", contudo as definições inerentes ao Teste Funcional, ganharam um maior destaque e serão assunto do capítulo três, frente a sua importância na pesquisa aqui presente.

A área de tecnologia está em se tornando essencial no cotidiano das pessoas e empresas sendo cada vez maiores e complexos, dando ênfase a importância da qualidade, com esse aumento torna-se um momento muito bom para os desenvolvedores. Desenvolvedores estão sendo requisitados pelo mercado e metodologias, técnicas e ferramentas estão surgindo para auxiliar no desenvolvimento e manutenção dos sistemas. (BURNSTEIN, 2003).

Nos últimos anos, o estado da técnica em testes de *software* teve uma evolução significativa, e tais testes não são mais entendidos somente como um processo a ser executado após a implementação do *software* e sim como uma tarefa executada de forma integrada à implementação, não em etapas, ou seja, de maneira direta (processo continuado até seu término), chamado de *Big-Bang Testing*. (SILVA, 2008).

O fluxo padrão de etapas de um processo de desenvolvimento iterativo incremental prevê a etapa de testes durante todas as etapas do processo de desenvolvimento de *software*, sendo gradual e tendo sua maior interação após a construção do mesmo, como ilustrado na Figura 1.

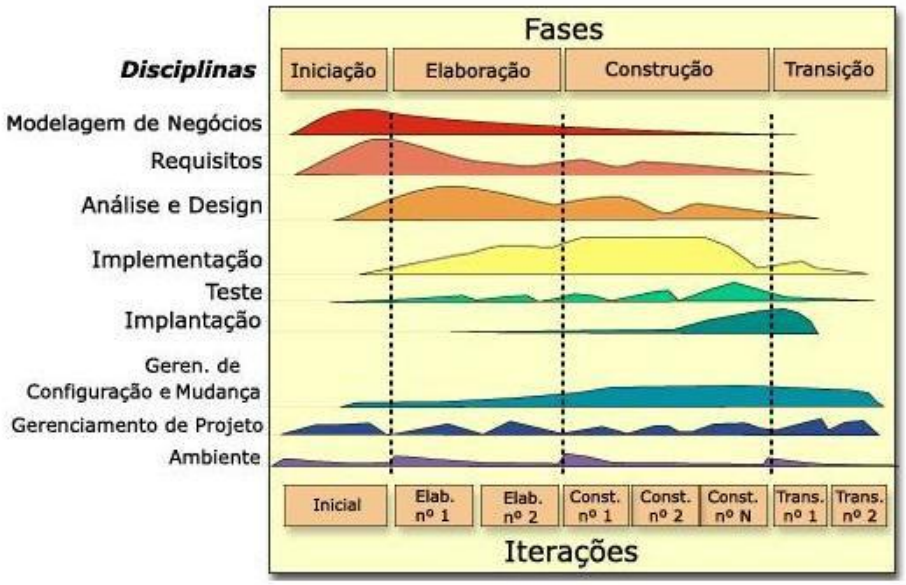

Figura 2 - Processo Unificado Fonte: (RUP, 2008).

A área de teste de *software* ainda é pouco difundida entre as empresas de desenvolvimento por diversos motivos, sendo que podemos citar como um dos principais, o tempo aplicado na atividade de teste e o custo da atividade de teste no desenvolvimento de *software*. Para melhorar esse cenário são usadas técnicas de desenvolvimento de testes automatizados unitários, funcionais, entre outros; que trazem benefícios como tempo gasto para identificação de falhas e defeitos no *software*, garantindo os testes de regressão, oferecendo garantia dos resultados esperados da qualidade do *software* e conformidade com os requisitos entre outros benefícios. Tais testes automatizados, ainda possuem um tempo de implementação muito alto, elevando o custo do projeto. Uma possível solução para esse problema é o desenvolvimento de *framework* que promove, através de métodos genéricos, reutilizáveis e dinâmicos, uma simples interpretação, utilizando os benefícios do uso da massa de dados tendo um ganho significativo no tempo de desenvolvimentos dos planos, casos e variações de testes, na medida em que é possível, com o mesmo script de teste, executar diversas varrições desse mesmo caso de teste. Além disso, segundo Myers (2004, p.50) "a utilização de ferramentas de teste automatizado pode minimizar parte das atividades maçantes e demorada".

Segundo Fayad et al (1999b) e Johnson & Foote (1988), um framework é um conjunto de classes que fazem parte de uma concepção abstrata para a solução de uma série de dificuldades e necessidade.

Segundo Mattsson (1996, 2000), um framework é uma arquitetura desenvolvida com a finalidade de alcançar a máxima reutilização de sua estrutura, representada como um conjunto de classes abstratas e concretas, sendo fácil de adaptação.

Buschmann et al. (1996), Pree (1995) e Pinto (2000) define que *framework* é como um software parcialmente completo projetado para ser instanciado. O framework determina uma arquitetura para uma série de subsistemas e oferece os construtores básicos para criá-los.

Trabalhando com a temática "teste de *software*", tem-se uma abordagem conceitual, em um primeiro momento, para tornar claro todos os caminhos apresentados neste universo de estudos. Assim, resgata-se a diferenciação entre os termos: Defeitos, Erros e Falhas. Tal definição advém da padronização terminológica regida pela Engenharia de *Software* do IEEE – Institute of Electrical and Electronics Engineers – (IEEE 610, 1990).

Tabela 1 – Conceitos Iniciais.

DEFEITO	É um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.
ERRO	É uma manifestação concreta de um defeito num artefato de *software*. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.
FALHA	Falha é um comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.

Fonte: (IEEE 610, 1990).

Com esta base, sabe-se que teste de *software* é um processo que procura executar um produto visando definir suas especificações e funcionamento correto no ambiente ao qual foi projetado (Myers, 2004 p 8).

O seu objetivo é evidenciar defeitos e erros em um produto, buscando as correções dessas falhas, através de sua identificação e corrigindo-as com o auxílio de

uma equipe de desenvolvimento, processo que deve ser antecedente à entrega final do *software* em desenvolvimento. Graças a estas vertentes das atividades de teste, pode-se afirmar que sua origem é “destrutiva”, e não “construtiva”, na medida em que busca o crescimento da confiança de um produto expondo seus problemas, antes de sua entrega ao consumidor final. O conceito de teste de *software* pode ser entendido sob uma perspectiva que considera a intuição ou ainda uma abordagem mais técnica. (DIAS NETO, 2008). Nos dias de hoje, inúmeras definições são entregues a este conceito, mas de maneira simples testar um *software* é observar criticamente, tendo por base uma execução controlada e se atentando ao comportamento do *software*, de modo a analisar se este está se comportando como previsto. O objetivo principal desta atividade é mostrar o número máximo de falhas com o mínimo de esforço, ou seja, tornar visível aos desenvolvedores se os resultados seguem conforme os padrões previstos. (DIAS NETO, 2008).

A atividade de teste é enxergada como um conjunto de ações formado por etapas passíveis de serem realizadas em inúmeros pontos no decorrer do processo de desenvolvimento convencional; ou seja, se define os testes, executando-os de modo simultâneo ao desenvolvimento do *software*” (Coelho et. al., 2006). Desse modo, cada etapa de desenvolvimento ou manutenção deve conter uma fase de testes Como se vê na Figura 2. Faz-se importante ressaltar que cada uma das etapas pode ser executada independentes. (SILVA, 2008).

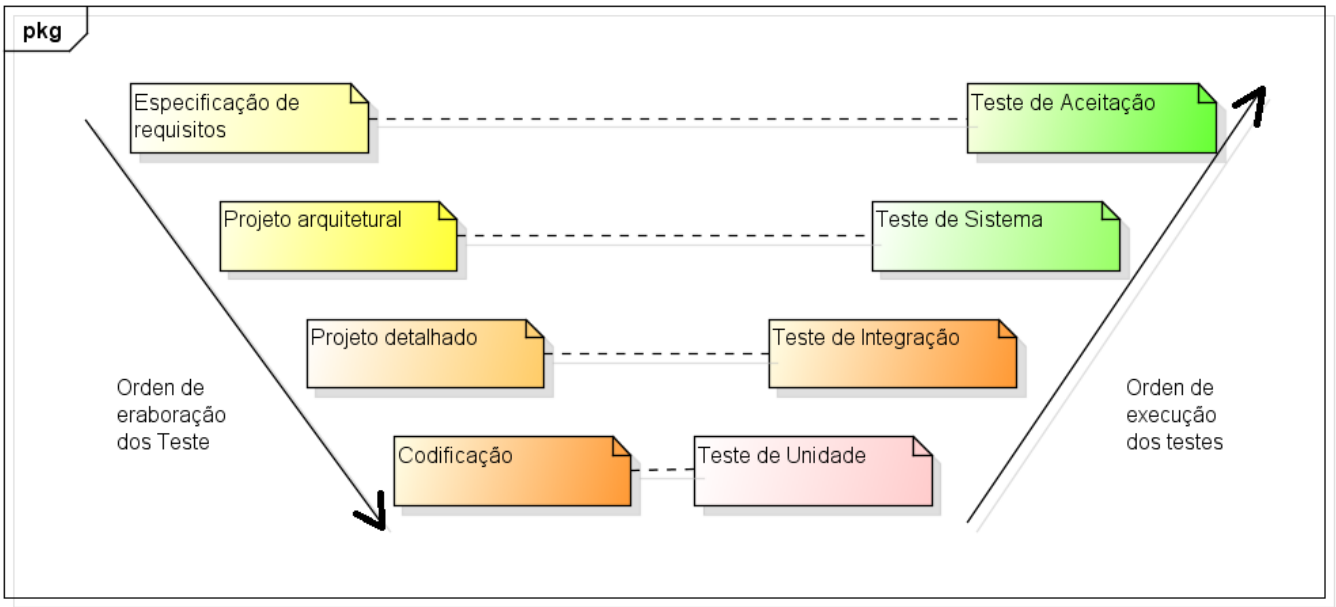

Figura 3 - Correspondência entre processo de desenvolvimento e de testes Fonte: (Myers, 2004).

Os testes elencados na Figura 2, sejam: a) Teste de Unidade; b) Teste de Integração; c) Teste de Sistema; d) Teste de Aceitação; e) Teste de Regressão; serão definidos no capítulo pertinente de forma não aprofundada, para não ferir o objetivo desta pesquisa.

1.1 Conceitos e Princípios

Teste de *software* é definido como um conjunto de execução de *software* de forma controlada pautado na pergunta: "O *software* se comporta conforme previsto?". Assim, apresenta-se que o intuito do teste de *software* foque somente do que está escrito na documentação, focando-se também no descobrimento de falhas que talvez venham a existir.

Um teste eficiente é aquele que verifica inúmeras vertentes do *software* induzindo-o a falhar. Neste seguimento, vê-se que o teste de *software* é uma ação destrutiva; afinal, com a mesma equivalência de importância está a averiguação de um *software* para resgatar o que ele realiza, como por exemplo, as tarefas para as quais foi projetado. Ainda se podem encontrar ações do *software* ocorrendo de modo não previsto.

Analisando a questão de custos do projeto, que alteração dos *software*s são cada vez mais caros, conforme passasse os dias, deste modo, se um problema for descoberto tardiamente, tão grande será o custo para que se corrija. Aconselha-se então que os testes sejam feitos no início do projeto. (ROTTA NETO; SANTOS, 2001). É importante entender que cada etapa do desenvolvimento de um *software* deve possuir uma fase de teste correspondente Figura 2, de modo que a Tabela 2, abaixo é apresentado cada uma destas correspondências:

Tabela 2 - Processo de desenvolvimento e de testes.

Teste de Unidade	O Teste de Unidade, também chamado de este Unitário, é a fase de um processo de teste na qual as menores unidades de um *software* em desenvolvimento (componentes, módulos) são testadas, normalmente

	pelos próprios desenvolvedores (diferentemente das demais fases de teste). Deste modo, o universo alvo desse tipo de teste normalmente são funções e métodos ou mesmo pequenos fragmentos de código. A complexidade dos testes de unidade e dos erros descobertos é limitada pelo escopo restrito estabelecido para este tipo de teste. O teste de unidade enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente
Teste de Integração	*Teste de integração* é uma fase sistemática na qual, enquanto a arquitetura do *software* está sendo construída, ao mesmo tempo, testes são conduzidos para descobrir erros associados às interfaces entre as unidades. A melhor forma de conduzir esses testes é de forma incremental, de forma que o programa seja construído e testado em pequenos incrementos, fazendo com que os erros sejam mais fáceis de serem isolados e corrigidos. Esta fase, assim como a de Testes de Unidade, é igualmente importante para que erros entre interfaces não sejam descobertos mais na frente, quando o foco já é nos testes do funcionamento do sistema como um todo.
Teste de Sistema	Teste de sistema consiste em uma série de diferentes testes cuja finalidade principal é exercitar por completo o sistema. Cada um destes testes tem um objetivo específico, porém a junção deles deve verificar se os elementos do sistema foram integrados de forma adequada e se executam as funcionalidades corretamente.
Teste de Aceitação	Nesta fase, os testes devem ser conduzidos pelos usuários finais do sistema. O objetivo é de demonstrar a conformidade com os requisitos definidos pelo usuário. Esta fase é bastante importante, pois seu resultado irá determinar se um sistema satisfaz ou não os critérios de aceitação, e permitir ao cliente julgar se aceita ou não o sistema.

Fonte: Davi Augusto Gadêlha Silva2008.

Pressman (2005), aborda que Teste de *software* é a atividade de executar um *software* procurando realizar uma avaliação quanto à qualidade, salientando seus defeitos e demais ruídos de execução; soma-se à isto o fato de que esta ação representa a última revisão de especificação, projeto e codificação.

O teste de *software*, como todo seguimento em uma área de conhecimento, possui conceitos que dão sustentação na ideia que envolve a compreensão do teste. Foi pensando assim que Rios (2012) abordou um conjunto de conceituações de teste de *software*, resgatando autores destacados nesta área e suas definições, como se pode ver na Tabela 3.

Tabela 3 - Conceitos

ISO/IEC	Testar software é a atividade de comparar o que um item de teste faz com o que é esperado que faça (ISO/IEC-29119)
Martin Pol e outros	Em linhas gerais, podemos dizer que o objetivo dos testes é encontrar defeitos: desta forma os testes são conduzidos para demonstrar a ausência de qualidade expressa pela presença de defeitos, para tal se faz necessário um processo (planejamento, especificação, execução, análise de resultados), considerando-se sempre os riscos do negócio e a qualidade do produto
Rios & Moreira	Verificar se o software é executado de forma controlada e está fazendo o que deveria fazer, de acordo com os seus requisitos, e não está fazendo o que não deveria fazer.
Glenford Myers	Teste de software é um processo, ou um grupo de processos, definidos para garantir que um código faz o que ele foi desenhado para fazer, e não faz nada que não foi especificado para fazer.

Fonte: Rios (2012)

O autor Mesquita (2011, p.25) define que o teste de *software* é "uma das atividades de garantia da qualidade que possui a finalidade de verificar se o produto em desenvolvimento está em conformidade com sua especificação". Já o IEEE (2004) conceitua teste de *software* como uma análise com dinamicidade, do comportamento do programa através de inúmeros testes, elaborados com a finalidade de resgatar surpresas no funcionamento do *software*.

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

Como o objetivo principal desde trabalho é o teste de *software*, outras atividades de V&V, como, por exemplo, revisões técnicas, não serão tratadas neste texto, mas salienta-se que são atividades que permitem a eliminação de outros tipos de defeitos, como em fases iniciais do processo de desenvolvimento, o que, em geral, representa uma economia significativa de recursos. (MESQUITA, 2011)

Silva (2011) colabora com esta pesquisa na medida em que traz acréscimos ao conceito de teste de *software*. O autor explica a garantia de confiabilidade do produto advém de técnicas para revisão, sendo formais e especificadas.

No universo do teste de *software*, é preciso ter um bom entendimento quanto aos três pilares que dão suporte intelectual ao processo, pois é a base para compreender a definição, bem como a adaptação prática e elaboração final do planejamento do teste em um projeto. Nesta ótica, cada etapa do processo de realização do *software* prevê a realização de cada teste. (NETS, 2012)

Mats (2001) *apud* Leal (2010) aponta cinco problemas que mais se destacam, no contexto de testes de *software*:

- Atrasos no cronograma do projeto, impossibilitando a equipe de completar os testes planejados devido à redução de recursos e tempo;
- Carência na rastreabilidade de casos e procedimentos de teste entre diferentes versões do *software*, o que dificulta a reutilização;
- Teste manual ou não-padronizado, resultando em um grande esforço a cada início de uma nova atividade de teste;
- Incerteza sobre o que está sendo testado, devido à falta de iniciação dos objetivos e escopo para as atividades de teste;
- Ausência de critérios para a seleção do conjunto de casos e procedimentos de testes. (LEAL, 2010).

1.2 Processo E Fases De Teste

Rotta Neto e Santos (2001) mostram em seu trabalho que existem duas principais técnicas de testes de *software*, que são: "Teste Estrutural ou *White-Box* ou ainda *Glass-Box* e o Teste Funcional ou *Black-Box*."

No Teste Estrutural os dados de teste exercitam a lógica interna dos componentes de *software* e comparam o comportamento do programa contra a intenção

aparente do seu código fonte. Na outra técnica, o Teste Funcional ou *Black-Box*, os dados de teste são utilizados para comparar o funcionamento do sistema contra seus requisitos não levando em consideração como os programas operam internamente. (ROTTA NETO; SANTOS, 2001).

Existem outros tipos de testes de *software* citados a baixo, contudo serão melhor abordados no capítulo seguinte.

1.2.1 Funcional

O teste funcional é utilizado quando o objetivo é verificar se as especificações foram atendidas, ou seja, o foco do teste funcional são os requisitos funcionais do sistema. Portanto, o teste funcional olha para as funcionalidades do sistema e também é chamado de teste baseado em especificação, ou teste caixa preto. (NETS, 2012).

1.2.2 Não funcional

Embora de modo a não se limitar, os testes não funcionais incluem: teste de performance; teste de carga; teste de estresse; teste de usabilidade; teste de interoperabilidade; teste de manutenibilidade; teste de confiabilidade e teste de portabilidade. É o teste de "como" o sistema trabalha. Testes não funcionais podem ser realizados em todos os níveis de teste. O termo teste não funcional descreve que o teste é executado para medir as características que podem ser quantificadas em uma escala variável, como o tempo de resposta em um teste de performance. O teste não funcional busca analisar os aspectos comportamentais do sistema. Como exemplos de teste não funcional, podemos citar o teste de usabilidade. (NETS, 2012).

1.2.3 Estrutural

A técnica de teste estrutural é mais conhecida como teste caixa branca. O teste estrutural tem como base o conhecimento da estrutura interna da codificação. Portanto, o teste estrutural é baseado no código escrito para implementar um componente ou sistema. O teste caixa branca é usado para garantir que elementos de uma estrutura foram corretamente definidos. Por exemplo, as técnicas de teste estrutural podem ser

usadas para garantir que declarações do código de um determinado sistema são executadas uma única vez. Nesse caso, os testes estruturais podem acontecer em qualquer nível de teste. (NETS, 2012).

Quanto aos testes estruturais, os autores Rotta Neto e Santos (2001) ainda acrescentam que estes analisam internamente as estruturas do *software*, sendo passíveis de derivar testes que:

- Garantam que todos os caminhos independentes dentro de um módulo sejam exercitados pelo menos uma vez;
- Exercitem todas as decisões lógicas para valores *falsos* ou *verdadeiros*;
- Executem todos os laços em suas fronteiras e dentro de seus limites operacionais;
- Exercitem as estruturas de dados internas para garantir a sua validade. (ROTTA NETO E SANTOS, 2001, p. 52).

CAPÍTULO 2 AUTOMAÇÃO DE TESTES FUNCIONAIS

Neste capítulo são abordados os princípios básicos do conceito de teste automação de teste com ênfase em teste funcional, trazendo um comparativo entre diversas técnicas de teste automatizado e suas vantagens e desvantagens.

A complexidade dos testes vem se intensificando no universo do desenvolvimento de aplicações de *software*. Tendo consciência que o tempo é um fator cada vez mais escasso, é necessário que os profissionais da área de teste pensem em ferramentas que viabilizem os testes de modo mais eficiente e rápido. (CALDAS, 2009).

Para que teste uma característica determinada, faz-se viável, executar os testes uma quantidade plural de vezes; verificando os erros encontrados nas tentativas de testes anteriores e buscando efetivas soluções, atentando para o aparecimento de novos erros na aplicação. Nesta ótica, testes de regressão[1] é uma opção pertinente. (FEWSTER; GRAHAM, 1999)

Um projeto de pequena proporção, com a necessidade de ser submetido a um número variado de testes, pode sofrer com a impossibilidade de repetições do mesmo teste, ignorando ainda o fato de que executar um teste por várias vezes é algo enfadonho e pode leva um tempo considerável. Este problema pode ser resolvido com a presença de ferramentas de testes de automação, que subtrai a uma execução manual e de muito tempo. Lembre-se que a automação reduz o tempo de execução do teste, contudo não possui uma ligação direta com a qualidade do teste, ou seja, execução manual ou com o auxílio de ferramentas, pode-se gerar o resultado igual. O pilar que sustenta a maior vantagem em automatizar um teste, está sob a premissa de que erros serão encontrados com maior facilidade e velocidade, embora o custo em sustentar este tipo de teste seja maior. (PETTICHORD, 2001).

Os tipos de automação, habitualmente, reuni tendo por base a interação dos testes automatizados com a aplicação, um exemplo deste agrupamento é visto na Tabela 4 - Agrupamento de tipos de automação:

[1] Teste de Regressão é um seguimento do Teste de software que busca aplicar a última versão do software, se atentando a novos defeitos. Quando comparado o componente analisado e alterado, com os demais componentes do sistema, observará se há defeitos nos componentes que não sofreram alterações, uma vez havendo, tem-se que o sistema regrediu.

Tabela 4 - Agrupamento de tipos de automação.

Paradigma	Tipos de teste automatizado
Baseado na interface	Baseado na interface gráfica (Capture/Playback)
	Dirigido a dados (Data-Driven)
	Dirigido a palavra-chave (Keyword-Driven)
Baseado na lógica de negócio	Baseado na linha de comando (Command Line Interface)
	Baseado em API (Application Programming Interface)
	Test Harness

Fonte: Adaptado pelo autor.

A relação entre os testes automatizados e a aplicação resulta no agrupamento dos tipos de automação, estes possuem divisão em dois paradigmas, baseados em Interface gráfica e baseado na lógica de negócio. A Tabela 5 conceitua estes dois importantes paradigmas trazendo vantagens e desvantagens, contudo, esta pesquisa entende que os testes de automação não se limitam nestes dois modelos.

Tabela 5 - Comparação tipos de automação

Baseados na interface gráfica: Nesta abordagem os testes automatizados interagem diretamente com a interface gráfica da aplicação simulando um usuário. Normalmente as ações dos usuários são gravadas (*Capture*) por meio da ferramenta de testes automatizados. A ferramenta transforma as ações dos usuários em um *script* que pode ser reproduzido (*Playback*) posteriormente.	**Vantagens:** Não requer modificações na aplicação para criar os testes automatizados. Também não é necessário tornar a aplicação mais fácil de testar (testabilidade) porque os testes baseiam-se na mesma interface utilizada pelos usuários.
	Desvantagens: Existe uma forte dependência da

<table>
<tr><td></td><td>estabilidade da interface gráfica. Se a interface gráfica mudar, os testes falham. Baixo desempenho para testes automatizados que exigem centenas de milhares de repetições, testes de funcionalidades que realizam cálculos complexos, integração entre sistemas diferentes e assim por diante.</td></tr>
<tr><td rowspan="2">Baseados na lógica de negócio:

Nesta abordagem os testes automatizados exercitam as funcionalidades da aplicação sem interagir com a interface gráfica. Normalmente é necessário realizar modificações na aplicação para torná-la mais fácil de testar (testabilidade). Essas modificações resultam em mecanismos para expor ao mundo exterior as funcionalidades da aplicação (APIs, Interfaces de Linha de Comando, Hooks, etc.), como veremos mais adiante. A interface gráfica é apenas uma casca (camada) que tem o objetivo de fornecer um meio para a entrada dos dados e apresentação dos resultados. A camada que abriga a funcionalidade e o comportamento da aplicação é a camada de lógica de negócio. Esta abordagem de testes é baseada no entendimento que 80% das falhas estão associados a erros na lógica de negócio.</td><td>Vantagens:

Foco na camada onde existe maior probabilidade de existir erros. Independência das mudanças da interface gráfica. Alto desempenho para testes automatizados que exigem centenas de milhares de repetições, testes de funcionalidades que realizam cálculos complexos, integração entre sistemas diferentes e assim por diante.</td></tr>
<tr><td>Desvantagens:

Requer grandes modificações na aplicação para expor as funcionalidades ao mundo exterior. Exige profissionais especializados em programação para criar os testes automatizados. Existem poucas ferramentas/frameworks que suportam essa abordagem (normalmente é necessário criar soluções caseiras).</td></tr>
</table>

Fonte: Adaptado pelo autor.

A automação subdivide-se em alguns tipos, são eles: Gravação e Reprodução; Dirigido a dados; Dirigido à palavra-chave; Baseado na linha de comando; Baseado na linha de comando; Baseado em API e Test Harness. Nesta ótica, estende-se:

Tabela 6 Tipos de Automação

Tipos de teste automatizados	Vantagens	Desvantagens
Gravação e reprodução	Rapidez e facilidade para criar scripts de teste;	Sem planejamento adequado os scripts ficam difíceis de dar manutenção
	Não é necessário modificar a aplicação ou torna-la mais fácil de testar;	Forte dependência a estabilidade da interface gráfica;
Dirigido a dados	Alta reutilização dos scripts de teste;	Não é possível criar scripts de teste complexos
	Não é necessário modificar a aplicação ou torna-la mais fácil de testar;	Forte dependência a estabilidade da interface gráfica;
Dirigido a palavra Chave	O usuário final pode criar os scripts de teste;	Não é possível criar scripts de teste complexos;
	Independente da Interface Gráfica;	Forte dependência da Interface gráfica;
Baseado na linha de Comando	Baixa exigência de modificação da aplicação	Pouca flexibilidade;
	Independente da interface gráfica;	Não é possível criar scripts de teste complexos;
Baseado em API	Possibilidade em scripts de teste robustos e complexos;	Alta exigência de modificação na aplicação;
	Independente da interface gráfica;	
Test-Harness	Uso racional e inteligente da automação;	Alta exigência de modificação da aplicação e criação do Test-Harness
	Possibilidade em scripts de teste robustos e complexos;	Existe poucas ferramentas/frameworks que suportam essa abordagem

	Melhor performance para testes que exigem muitas repetições e cálculos complexos;	(Normalmente torna-se necessário criar próprias soluções);

Fonte: (CAETANO, 2013).

Sabido que é complexa a tarefa de automatizar os testes ao *software*, deve-se considerar que a automação possui um alto preço, na medida em que a normalização deste teste requer um tempo maior para criar, verificar e documentar, com maior atenção aos detalhes. Realizar o processo de forma manual, pode otimizar o trabalho de automação. Lembra-se que automatizar o teste pode possibilitar a presença de novos custos, afinal, o valor monetário dos erros no *software* cresce mediante o ciclo de vida do *software*. Deste modo, um grande tempo voltado a criar os scripts, condicionará que os testes sejam realizados em fases mais avançadas do desenvolvimento. (CALDAS, 2009).

Quando consideramos as vantagens inerentes às ferramentas de *software*, há de considerar as seguintes características:

Velocidade – quando comparado a execução de testes de forma manual, a automação traz esta característica de forma significativa e pertinente. (CALDAS, 2009).

Eficiência – O formato manual de agir, requer que outras atividades sejam colocadas em formato de espera, de modo que, como a automação e a redução do tempo de execução do teste, haverá oportunidades de trabalhar com maior tempo em momentos como planeamento e desenho de novos casos de teste. (FEUP CIQS,2008).

Exatidão e precisão – Entende-se que algumas horas dedicadas a um trabalho monótono e repetitivo, pode provocar uma desatenção por parte do testador e isso vem a gerar possíveis erros. Esta lógica é tão designada ao trabalho manual, que é aceitável que repita-se o teste algumas vezes. Com a automação, conta-se com a subtração deste quadro. (CALDAS, 2009).

Redução de Recursos – Em muitas oportunidades, executar um teste requer materiais e mão de obra que são ausentes em data ocasião. Pelo custo inerente a estes, a execução do teste pode ser inviabilizada. Assim, uma ferramenta pode vir a simular a realidade e

reduzir de modo expressivo os recursos físicos necessários para a execução do teste. (FEWSTER; GRAHAM, 1999)

Simulação e Emulação – Ferramentas de teste podem vir a simular *hardware* ou *software* que, em muitas vezes, procura uma interação com o produto. Isso proporciona que crie situações ao *software* que, no universo manual, seria algo de extrema complexidade. (PETTICHORD, 2001).

É importante lembrar que a automação do teste, para que seja bem sucedida, requer algumas estratégias e as ferramentas necessárias para tanto, não são um substituto do profissional testador. É sim, um auxiliador para facilitar o trabalho com uma maior agilidade. Soma-se ainda que, em dadas ocasiões, substituir o teste manual é inviável.

CAPÍTULO 3 TESTE FUNCIONAL: ABORDAGEM CONCEITUAL

Neste capítulo será tratado, com uma abordagem mais profunda, a temática "teste funcional", afim de atingir uma conceituação buscando sua aplicabilidade pertinente no desenvolvimento do framework.

Define-se teste funcional, inicialmente, abordando que este também pode ser conhecido como caixa-preta ou comportamental que inclina atenção principalmente ao conjunto de requisito funcional do *software*, porem demais teste também se baseia nos requisitos. Beizer (1990), explica que a nome "Caixa Preta" advém da realidade em que lida-se com o *software* como se este fosse uma caixa, em que vê-se uma interface disponível, assim sendo, seu lado externo. Myers (1976) explora mais a ideia dizendo que o Teste Funcional leva esse nome porque enxerga o componente de *software* a ser testado como se tratasse de uma caixa-preta, ou melhor, desconsiderando seu comportamento interno. O "lado externo" do programa pode ser visto na Figura 3. O profissional que executará o teste usa a especificação do *software* para adquirir os requisitos do teste ou os dados de teste, seguindo de forma despreocupada com a implementação.

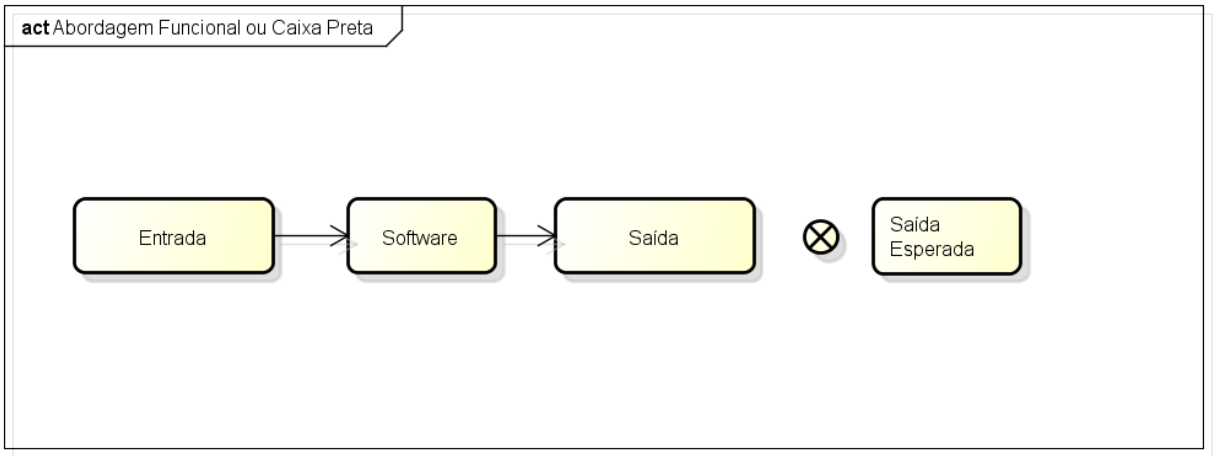

Figura 4 - Abordagem Funcional ou Caixa Preta. Fonte: Adaptado pelo autor (Coelho, 2005).

O teste funcional tem por base a identificação de requisitos funcionais, de forma especificada, fazendo uso de entrada e saída, sem focar na estrutura do código.

(Mayrhauser, 1990). Nesta ótica, "uma especificação de alta qualidade que cubra os requisitos do cliente é fundamental para a aplicação do teste funcional." (SILVA, 2008).

O Teste Funcional proporciona ao profissional da área de *software* derivar basicamente um conjunto de condições de entrada que vão exercitar todos os requisitos funcionais de um programa.

A análise de um produto realizado pelo teste funcional, dá-se de forma macroscópica, pois a implementação ou conteúdo, é ignorada pelo testador, de modo que este, faz uso da especificação, para derivar os requisitos de teste.

Pressman (2005), mostra que entre as atuações do Teste Funcional, está a denúncia da satisfatoriedade: - dos requisitos funcionais do *software*; -da aceitação da entrada; - da produção da saída prevista; - da integridade das informações externas. Nesta ótica, Modesto (2006) nos ajuda a concluir que esta sequência de ações que validam o teste funcional é proporcional à falta de cuidados com a estrutura lógica interna do sistema.

A aplicação da técnica de Teste Funcional implica no uso de métodos que tiveram sua gênese buscando atender os critérios dessa abordagem. Alguns dos métodos usados são: Particionamento de Equivalência (Equivalence Partition) e Análise de Valores de Fronteira (Boundary Value Analyses). (Pressman apud SILVA, 2008).

> O método de Particionamento de Equivalência divide o domínio de entrada de um programa em classes, para que casos de teste sejam derivados destas classes. Dessa maneira, um caso de teste definido por esse método é eficaz se descobre classes de erros, reduzindo assim o número total de casos de teste desenvolvidos. O método de Análise de Valores de Fronteira parte do princípio de que muitos erros ocorrem nas fronteiras do domínio de entrada, e não no "centro". Esse método é um complemento do método de particionamento de equivalência, e procura selecionar casos de teste que exercitam nessas fronteiras. (SILVA, 2008, p. 35).

O Teste Funcional exercita o sistema à ótica do usuário, de modo que não leva em conta a estrutura interna ou a maneira ao qual o sistema foi implementado. Quando não se tem conhecimento aprofundados em programação, o teste funcional passa a ser uma escolha muito viável. Sendo o mais popular modo de testar, o objetivo é observar se o sistema executa da forma prevista, suas funcionalidades. (PRESSMAN, 2005).

Técnicas de Caixa Branca, existem para detectar determinados tipos de erros no sistema e, ressalta-se que o teste funcional não é um segmento ou uma das alternativas

da Caixa Branca. Nota-se o oposto na medida em que o teste funcional denuncia um conjunto diferente de erros, perpassando pelos requisitos funcionais do *software*. Através do teste funcional, se possibilita identificar erros em entradas e saídas de cada unidade do *software*. O teste funcional preocupa-se com "o que", e não com o "como" uma determinada unidade do *software* está sendo executada (BEIZER *apud MODESTO*, 1990). Nota-se que

> Apesar de suas vantagens, a aplicação apenas da abordagem Caixa Preta não é suficiente, pois não é possível garantir que determinadas partes essenciais da implementação do *software* foram exercitadas. Além deste problema, as especificações do sistema, que são a única fonte de informação para os testes funcionais, podem estar incompletas ou escritas de forma ambígua, tornando os testes também insatisfatórios. A próxima seção trata de uma outra importante abordagem de teste, que complementa a abordagem Caixa Preta. (SILVA, 2008, p. 48).

O teste funcional possui, em suma, o objetivo de identificar (PRESSMAN, 2005):

- Funções incorretas ou ausentes;
- Erros de interface;
- Erros nas estruturas de dados ou no acesso a bancos de dados externos;
- Erros de desempenho;
- Erros de inicialização e término.

Visando aplicar o teste funcional, é necessário que se considere o estado dos objetos presentes no contexto do teste. Aplica-se o teste de maneira que este atinja seu ciclo de vida por completo, desde modo, as operações dos objetos não devem ser submetidas ao teste de maneira unitária ou individual. Soma-se que o teste funcional ignora a estrutura de controle, inclinando preocupação no domínio da informação. Para tanto, o teste funcional possui alguns critérios, indiferente de qual seja o teste, que se elencam: Particionamento de Equivalência, Análise de Valor Limite, Grafo Causa-Efeito.

O Particionamento de Equivalência divide o domínio de entrada de um programa em classes de equivalência válidas e inválidas, tendo como início as condições de entrada de dados identificadas na especificação. Posteriormente, seleciona casos de teste acreditando que um elemento de dada classe representaria a classe toda, não ignorando

que, para classes inválidas, devem ser gerados casos de teste divergentes. A utilização desse critério proporciona a restrição do número de casos de teste necessários. (PRESSMAN, 2002).

A Análise de Valor Limite é um critério que complementa o critério de particionamento de equivalência. Substitui a realização da seleção de qualquer elemento de uma classe de equivalência, de modo que a análise de valor limite leva à seleção de casos de testes nas extremidades da classe, afinal, esses pontos podem conter uma grande concentração de erros. Ao se concentrar somente nas condições de entrada, a análise de valor limite deriva os casos de teste também do domínio de saída. (PRESSMAN, 2002).

O Grafo Causa-Efeito estabelece requisitos de teste tendo por base as possíveis combinações das condições de entrada que os critérios de Particionamento de Equivalência e Análise de Valor Limite não investigam de modo minucioso. O primeiro passo consiste em levantar as possíveis condições de entrada (causa) e as possíveis ações(efeitos) do programa; posteriormente, é desenvolvido um grafo relacionando causas e efeitos, que é convertido em tabela de decisão a partir da qual são derivados os casos de teste. (PRESSMAN, 2002).

Tais critérios inerentes ao teste funcional são aplicados enquanto ocorre a geração ou seleção de casos de teste que acontece, tendo início na análise da especificação de requisitos. As características mais salientes do teste funcional, podem ser elencadas (ARANTES, 2012, p. 83):

> - Pode ser baseado na especificação e, portanto independente da implementação, o conjunto de testes adequado às funcionalidades permanece inalterado, mesmo que haja alteração da implementação;
> - Os critérios de teste funcional derivam subconjuntos representativos de todo o domínio da entrada das variáveis em teste;
> - A derivação dos casos de teste pode acontecer paralelamente à implementação, reduzindo o tempo do projeto;
> - Alguns dos critérios componentes do teste funcional (Grafo de Causa e Efeito, Tabela de Decisão, Teste Baseado em Transição de Estados e Teste Baseados em casos de Uso) são ótimas ferramentas para auxiliar na especificação dos requisitos do *software*.

Os testes funcionais desejavam visualizar erros nas seguintes vertentes: Funções incorretas ou ausentes;

- Erros de interface;

- Erros nas estruturas de dados ou no acesso a banco de dados externos;
- Erros de desempenho;
- Erros de inicialização e término.
- Os testes funcionais, normalmente aplicados durante as últimas etapas da atividade de teste, são
- Projetados para responder às seguintes perguntas:
- Quais entradas constituirão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como as fronteiras dos dados de entrada são isoladas?
- Quais volumes de dados o sistema pode tolerar? (CALDAS, 2009, p. 42)

Relacionado com este tipo de testes estão as seguintes atividades, mostradas por Caldas(2009):

- Construir o modelo: o modelo formal ou semiformal é construído tendo por base os requisitos da aplicação;
- Gerar inputs: os inputs do teste são criados partindo do modelo. Estes inputs são passos que servirão para interagir com a AUT.
- Gerar outputs esperados: os outputs esperados do teste são gerados partindo do modelo formal e servirão de referência para comportamento que se espera do sistema.
- Executar testes: a aplicação é executada com os inputs gerados, gerando outputs;
- Comparar outputs reais com os esperados: os outputs da aplicação que está a ser testada são comparados com outputs esperados gerados a partir do modelo.
- Decidir ações futuras: Após o resultado final, analisar a necessidade de gerar mais testes ou para-los, fazer uma estimativa sobre a fiabilidade (qualidade) do *software* ou até mesmo ponderar uma mudança no modelo. (CALDAS, 2009, p. 38)

As atividades acima podem ser exemplificadas com a Figura 5.

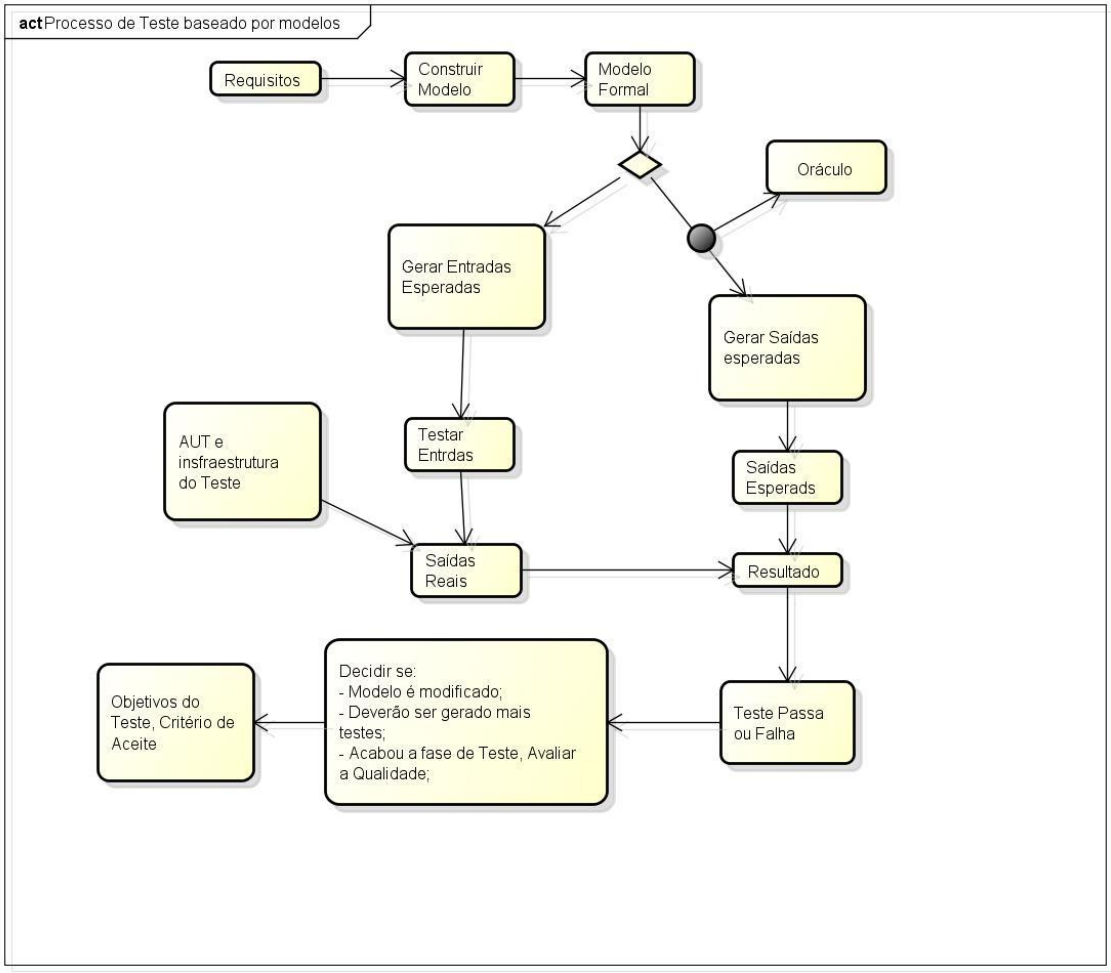

Figura 6 - Passos do processo de teste baseado em modelos. Fonte: Adaptado pelo Autor: (EL-FAR; WHITTAKER, 2001).

Uma característica importante do teste funcional é que este pode ser aplicado nas diferentes fases de teste (de unidade, integração, sistema e aceitação), praticamente inalterado na forma de aplicação dos testes, embora, sua maior aplicabilidade se dá nas duas últimas fases do processo de testes. Soma-se aos critérios de testes funcionais, que estes têm por base as especificações, se tornando independentes de linguagem ou plataforma. Tal característica proporciona a eles a possibilidade de serem utilizados em testar *software* procedimental, "orientado a objetos, e/ou orientado a aspectos, além de componentes de *software*" (BINDER *apud* SILVA, 2008) (OFFUTT; IRVINE *apud* SILVA, 2008)

CAPÍTULO 4 *FRAMEWORK* PROPOSTO

O presente capítulo apresentará a estrutura do *framework* proposto detalhando a importância de cada componente desenvolvido e as ferramentas utilizadas no desenvolvimento e conceitos básicos sobre *framework*.

4.1 Estratégias Para Automação De Software

Molde a expectativa da gerência em relação ao tempo dos benefícios da automação.

a. Há um grande benefício em automatizar uma suíte de testes: executar estes testes cinquenta ou cem vezes durante o desenvolvimento de um *release*. Mesmo se levar dez vezes mais tempo para desenvolver cada teste do que executar cada teste à mão, e mais dez vezes o tempo para a manutenção, haverá ainda um ganho de tempo entre trinta e oitenta execuções manuais.

b. Provavelmente, o benefício da automação não será percebido no primeiro *release* em que é iniciada a automação. A diminuição do esforço se dará paulatinamente nos próximos *releases (KANER apud* NÓBREGA, 2006).

***Desenvolvimento de uma automação de testes é um desenvolvimento de* software.**

a. Não se pode desenvolver grandes suítes de teste que sobreviverão e serão úteis em diversos releases e que terão baixo custo de manutenção sem um planejamento claro e realístico.

b. A automação utiliza uma linguagem de programação, mesmo que simplificada, como todo desenvolvimento de software.

c. Cada caso de teste pode ser encarado como uma funcionalidade.

d. Os automatizadores, da mesma forma que os desenvolvedores de software fazem, devem: entender os requisitos, entender de programação, adotar uma arquitetura robusta que permita desenvolver, integrar e manter as funcionalidades, entre outras atribuições de qualquer desenvolvedor de software. (KANER apud NÓBREGA, 2006).

Use uma arquitetura de testes orientada a dados.

a. A técnica de testes orientados a dados armazena as entradas dos testes em arquivos separados dos scripts de teste. Quando os scripts de teste são executados, esses

dados são lidos destes arquivos ao invés de estarem dentro do código do script. Esta separação clara aumenta consideravelmente a manutenibilidade dos scripts de teste. (KANER apud NÓBREGA, 2006).

Use uma arquitetura baseada em um *framework*.

a. Um *framework* provê uma abordagem diferente à automação de testes e geralmente é utilizada com um ou mais estratégias de testes orientados a dados.

b. Um *framework* isola a aplicação sob teste dos scripts de teste provendo um conjunto de funções em bibliotecas de funções. Os scripts de teste utilizam essas funções como se fossem comandos básicos da linguagem da ferramenta de automação. Então é possível programar os scripts de caso de teste independentemente da interface do usuário. (KANER apud NÓBREGA, 2006).

Lembre-se da Realidade da sua equipe.

a. Automação de Regressão através da GUI pode trazer uma falsa ideia de cobertura que não existe. E isto pode causar uma sobrecarga na equipe deixando os profissionais mais experientes criando e mantendo scripts ao invés de encontrando bugs.

b. Ferramentas de automação de testes pela interface com usuário são bastante úteis, mas requerem um investimento significativo, um planejamento detalhado, uma equipe bem treinada e bastante cuidado. (KANER apud NÓBREGA, 2006).

4.2 Tecnologias Envolvidas

Inicialmente, foi definida a utilização da linguagem de programação Java por ser robusta, livre, orientada a objeto e possuir uma comunidade de desenvolvedores bem ativa no mercado.

O *framework* possui tecnologias, ferramentas e conceitos que visam facilitar o desenvolvimento de casos de teste e poderá potencializar sua utilização aumentando suas funcionalidades e possíveis aplicações. Nesta perspectiva, tem-se:

Eclipse: - ferramenta *open-source* muito utilizada para desenvolvimento, sendo ela uma IDE que suporta a linguagem de Programação Java e integração das demais ferramentas e aplicação dos conceitos que serão utilizados nesse trabalho (Eclipse Documentação).

TESTNG: - *framework open-source* que visa facilitar a criação, gerenciamento e execução de planos de teste automatizados, tanto funcional quanto unitários, de forma simples (TESTNG, Documentação).

Maven: - ferramenta *open-source* usada para gerenciar projetos, *builds* e gerencia as dependências por meio de um conceito de (POM) Objeto modelo de Projeto (Maven Documentação).

Selenium 2: - *framework open-source* que auxilia na automação de teste em aplicações web, sendo flexível facilita a localização de elementos na interface simulando ação real do usuário através do *WebDriver* e permitindo a execução em múltiplas plataformas de navegadores (Selenium, Documentação Introdução ao Selenium).

PageObject: - trata-se de um conceito abstrato passível de mapear os elementos de uma interface de usuário e transformá-los em métodos nos casos de teste assim reduzindo quantidade de códigos duplicados, caso seja preciso realizar alguma mudança na interface a mesma será alterada em um único ponto (PageObejcts, Documentação).

4.3 Estrutura do *Framework*

Para atender os objetivos proposto no trabalho a estrutura do *framework* foi constituída por 4 pacotes no qual cada um tem suas distintas funções, sendo eles (*Framework*, CasoTeste, *PageObejcts* e Parametros) como mostra o modelo a Figura 5.

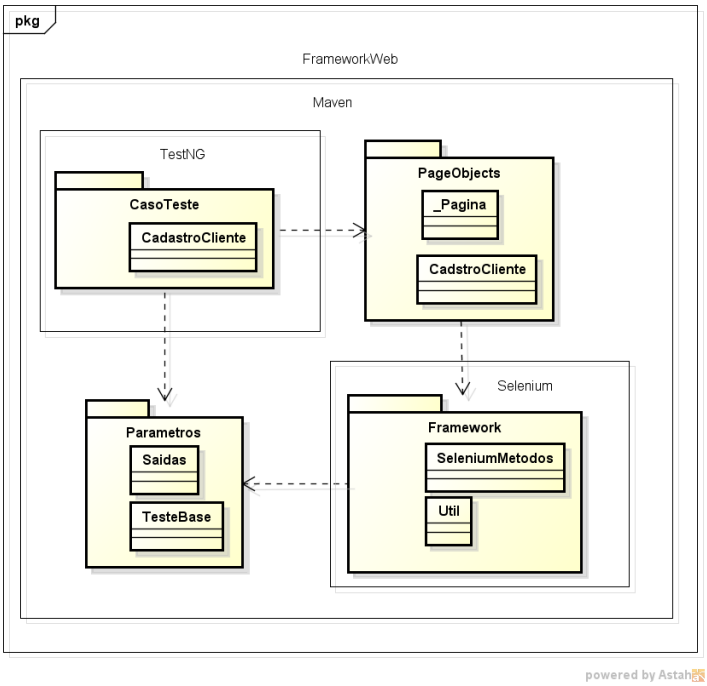

Figura 5 - Estrutura do *Framework*. Fonte: Autoria própria.

Intencionar a descrição de cada pacote presente no código, pode dizer que o Pacote Parâmetros contém todas as informações comuns para as demais classes do *framework.*

Pacote *PageObject*s possui a uma classe _pagina na qual contém métodos para identificação e tratamento dos elementos da página *web* e contém as "funções, elementos" das páginas traduzidos em métodos independentes que acessam os métodos do pacote *Framewor*k.

Pacote *Framewor*k possui, por sua vez, os métodos do *WebDrive*r que são simplificados para facilitar no desenvolvimento dos *PageObejcts*.

Pacote CasoTeste é o local onde é gerado os casos de teste utilizando os *PageObjects* e integrando com as massas de dados, cada pacote serão abordados detalhadamente a seguir.

4.3.1 Pacote Parametros

Para centralizar configurações, foram separadas duas Classes (TesteBase e Saidas), que serão comuns para que os analistas de teste possam utilizar para todo o projeto de teste.

O TesteBase possui atributos, como visto na Figura 6, que serão acessados pelas demais classes do *framework* como, por exemplo navegador a ser utilizado, diretório de massa de dados e informações de acesso.

```
31      //driver que executa os comandos no navegador
32      public static SeleniumMetodos driverSelenium;
33      //Navegador utilizado (firefox, chrome, ie)
34      public static String navegador = "firefox";
35      //Diretório raiz para armazenar as screenshots
36      public static String diretoriaRaziEvidencias = "C:/Prints/";
37      //Armazena diretório final das screenshots
38      public static String diretorioEvidencias;
39      //Armazena arquivo de Massa de Dados
40      public static Workbook massaDados;
41      //Armazena planilha de um arquivo xls
42      public static Sheet planilhaExcel;
43      //Diretório que armazena as Massas de Dados
44      public static String direotorioRaizMassa = System.getProperty("user.dir")+"/massas de dados/";
45      //Diretório que armazena as imagens para Massas de Dados
46      public static String diretorioImagens = direotorioRaizMassa+"imagens/";
47      //PageObjects
48      public static PaginaInicial paginaInicial;
```

Figura 6 - Atributos Classe TesteBase. Fonte: Autoria própria.

Para facilitar o uso de atributos comuns para todo o *framework*, foram definidos atributos genéricos comuns em que *SeleniumMetodos* instancia um driver do *WebDriver* no qual será utilizado durante toda a execução de um caso de teste, o atributo *navegador* recebe como padrão um navegador no qual será executado o caso de teste podendo variar entre opções como (FIREFOX, CHROME, INTERNET EXPLORE, SAFARI). *diretorioRaizEvidencias* e *diretoriEvidencias* determinam um diretório para armazenamento das evidências de cada execução de teste assim podendo ser local ou um diretório em rede compartilhado; os atributos *massaDados* e *planilhaExcel* determinam a massa de dados que será consumido a cada execução dos casos de teste, são usados para importar o arquivo Excel, que é preciso estar no formato .xls. Por último, o atributo *dirotorioRaizMassa* determina o diretório onde ficará as massas de dados. Estas massas contém um segundo atributo *dirotorioImagens* que determina um diretório exclusivo para o armazenamento de imagens.

Ainda na TesteBase possui os métodos, como pode ser visto na Figura, que serão acessados pelas demais classes do *framework*, assim tornando-as reutilizáveis.

```
52      /** Inicializa o driver e o configura para execução dos testes **/
53      @BeforeSuite (alwaysRun=true)
54      public void antesSuite() {
55          diretoriaRaziEvidencias += "Test@"+formataData("file")+"/";
56          driverSelenium = new SeleniumMetodos(navegador);
57          driverSelenium.manage().window().maximize();
58          paginaInicial = new PaginaInicial(driverSelenium).abrirPagina();
59
60      }
61      /** Fecha todas as janelas do navegador abertas pelo driver e encerra execução **/
62      @AfterSuite (alwaysRun=true)
63      public void depoisSuite() {
64          driverSelenium.quit();
65      }
66      /** Valida uma condição lógica e gera uma falha caso seja falsa **/
67      public static void verificaVerdadeiro(boolean condition) {
68          try {
69              assertTrue(condition);
70          } catch(Throwable e) {
71              adicionaFalhas(e);
72          }
73      }
74      /** Valida uma condição lógica e gera uma falha caso seja verdadeira **/
75      public static void verificaFalso(boolean condition) {
76          try {
77              assertFalse(condition);
78          } catch(Throwable e) {
79              adicionaFalhas(e);
80          }
81      }
82      /** Valida a igualdade de dois objetos e gera uma falha caso sejam diferentes **/
83      public static void verificaIgualdade(Object actual, Object expected) {
84          try {
85              assertEquals(actual, expected);
86          } catch(Throwable e) {
87              adicionaFalhas(e);
88          }
89      }
```

Figura 7 - Métodos Classe TesteBase. Fonte: Autoria Própria.

Para facilitar a utilização de funções e rotinas na qual deve ser reutilizadas para todo caso de teste a ser criado, foi desenvolvido diversos métodos, sendo eles: a) *Map*, método usado para armazenar temporariamente as falhas ocorridas na execução de um caso de teste; b) método *antesSuite*, que executa um rotina a cada execução que por padrão inicia os atributos *dirotorioImagens* e um *driver*; c) *depoisSuite* fecha todas as janelas abertas pelo driver durante a execução; d) *verificaVerdadeiro* recebe um condição lógica e verifica caso seu resultado seja verdadeiro e armazena no *Map;* e) *verificaFalso* recebe um condição lógica, verificando a falha no resultado e armazenando no *Map;* f) *verificaIgualdade* verifica se dois objetos são iguais e armazena seu resultado no *Map*; g) *List* verifica todas as exceções geradas na execução do teste e as exibem numa lista para análise.

A classe Saídas possui métodos que são invocados pelo *TestNG* durante certos pontos da execução do teste, como visto na Figura 8, informações do processo da execução do teste automatizado em forma de texto. Tem-se como exemplo a ser utilizado: [INICIO], [FALHA] e [SUCESSO], seus nomes de métodos estão em inglês pois são nativos do TestNG.

```
15      /** Executa ao início da execução de um teste **/
16      public void onTestStart(ITestResult result) {
17          System.out.println("\n[INICIO] Execução do teste \""+result.getMethod().getDescription()+"\" iniciada.\n");
18      }
19
20      /** Executa ao término da execução de um teste com sucesso **/
21      public void onTestSuccess(ITestResult result) {
22          System.out.println("\n[SUCESSO] Teste \""+result.getMethod().getDescription()+"\" executado com Sucesso!");
23      }
24
25      /** Executa ao término da execução de um teste com falha **/
26      public void onTestFailure(ITestResult result) {
27          System.out.println("\n[FALHA] Teste \""+result.getMethod().getDescription()+"\" executado com Falha!\n");
28          result.getThrowable().printStackTrace();
29      }
30
31      /** Executa quando um teste não é executado **/
32      public void onTestSkipped(ITestResult result) {
33          System.out.println("\n[PENDENTE] Teste \""+result.getMethod().getDescription()+"\" não executado\n");
34          result.getThrowable().printStackTrace();
35
36      }
```

Figura 8 - Métodos Classe Saídas. Fonte: Autoria Própria.

4.3.2 Pacote *PageObejcts*

Para uma abstração simplificada do sistema toda página *web* é dividia em pedaços denominados objetos da página e organizados no pacote *PageObjects* onde são desenvolvidos as classes referentes a cada página da aplicação a ser testada, para a

criação do *PageObject* a partir de uma análise mapeando todas as funcionalidades da página onde cada interação constitui um método do *PageObject*, e cada elemento se tornam um @FindBy que facilita a interpretação e alteração do código caso o sistema altere esse valor não sendo necessário a fatoração de todos os casos de teste que utiliza esse elemento, onde se pode analisar na Figura 9, toda essa organização e separação.

```
package PageObejcts;
import jxl.Sheet;
/** PageObject referente a Página de cadastro **/
public class CadstroCliente extends _Pagina{
    @FindBy(id = "pf_nome_cliente") private WebElement nome;
    @FindBy(id = "pf_data_nascimento") private WebElement dataNascimento;
    @FindBy(id = "btn_editar") private WebElement BotaoEditar;
    @FindBy(id = "btn_cadastrar") private WebElement BotaoCadastrar;
    public CadstroCliente(SeleniumMetodos driver) {
        super(driver);
    }
    public CadstroCliente abrirPagina(){
        driver.get("UrlPaginaCadastro");
        return this;
    }
    public void PrencherCampos(Sheet planilha, int coluna){
        driver.type(nome, planilha.getCell(coluna, 2).getContents());
        driver.type(dataNascimento, planilha.getCell(coluna, 3).getContents());
    }
    public void Editar(){
        BotaoEditar.click();
    }
    public void Cadastrar(){
        BotaoCadastrar.click();
    }
}
```

Figura 9 – Exemplo da classe *PageObject* cadastro de cliente. Fonte: Autor.

No pacote *PageObject* foi desenvolvido uma classe padrão chamada _pagina na qual todos as demais classes devem estender seus atributos, nela contém métodos de identificação e tratamento dos elementos HTML das páginas da aplicação web, também nessa classe foi desenvolvido métodos para auxiliar na manipulação dos elementos sendo eles validarCampo, que verifica qual o tipo do elemento HTML, podendo ser text, combo, checkbox, radio e list. Assim, pode-se fazer de forma genérica as devidas validações nos elementos de acordo com o seu tipo, facilitando a forma de se fazer comparações nos casos de teste, deixando-as mais intuitivas, como exemplifica a Figura - 10.

```
protected boolean validarCampo(WebElement elemento, String valorEsperado, String tipoElemento){
    boolean validacao = true;
    String valorEncontrado = elemento.getAttribute("value");
    String elementoNome = "[" + elemento.toString().split(" -> ")[1];

    if(!valorEsperado.isEmpty()){
        valorEsperado = valorEsperado.equals("<vazio>") ? "" : valorEsperado;
        if(tipoElemento.equals("TEXT")){
            valorEncontrado = elemento.getAttribute("value");
            validacao = valorEncontrado.equals(valorEsperado);
        }
        else if(tipoElemento.equals("COMBO")){
            valorEncontrado = new Select(elemento).getFirstSelectedOption().getText();
            validacao = valorEncontrado.equals(valorEsperado);
        }
        else if(tipoElemento.equals("CHECKBOX") || tipoElemento.equals("RADIO")){
            valorEncontrado = elemento.isSelected() ? "true" : "false";
            validacao = valorEncontrado.equals(valorEsperado);
            //valorEncontrado.replaceAll("\\s+", " ").equals(valorEsperado);
        }
        else if(tipoElemento.equals("LIST")){
            Select select = new Select(elemento);
            for(int index=0; index<valorEsperado.split("\\|").length; index++){
                validacao = validacao
                        && select.getOptions().size() > index
                        && select.getOptions().get(index).getText().equals(
                        valorEsperado.split("\\|")[index]);
            }
            for( WebElement opcao : select.getOptions()){
                valorEncontrado += opcao.getText()+"|";
            }
            if(!valorEncontrado.isEmpty())
                valorEncontrado = valorEncontrado.substring(0, valorEncontrado.lastIndexOf("|"));
        }
    }
        if(validacao)
            System.out.println("[TRUE]  Valor(es) encontrado(s) corresponde(m) com " +
                    "valor(es) esperado(s): '"+valorEsperado+"'! " + elementoNome);
        else
            System.out.println("[FALSE] Valor(es) encontrado(s): '"+valorEncontrado+"'" +
                    " não corresponde(m) com valor(es) esperado(s): '"+valorEsperado+"'! " + elementoNome);
    return validacao;
}
```

Figura 10 - Método validaCampo. Fonte: Autoria própria.

Para a validação de texto foi desenvolvido na classe _pagina, dois métodos: validarTexto e validarAlerta, de modo que cada qual recebe parâmetros diferentes, facilita a maneira que se realiza as comparações de texto independentemente do tipo do elemento HTML, que é passado como localizador no caso de teste, sendo exemplificado na Figura - 11.

```
/** Realiza a validação do texto interno de um elemento HTML
 * @param elemento : Elemento HTML cujo texto deve ser validado
 * @param textoEsperado : Texto que deve ser validado no elemento HTML **/
protected boolean validarTexto(WebElement elemento, String textoEsperado){
    boolean validacao = true;
    String textoEncontrado = elemento.getText().trim();
    String elementoNome = "[" + elemento.toString().split(" -> ")[1];
    validacao = textoEncontrado.equals(textoEsperado);
    if(validacao)
        System.out.println("[TRUE]  Texto encontrado corresponde com texto esperado: '"+textoEsperado+"'! " + elementoNome);
    else
        System.out.println("[FALSE] Texto encontrado: '"+textoEncontrado+"'" +
                " não corresponde com texto esperado: '"+textoEsperado+"'! " + elementoNome);
    return validacao;
}
protected boolean validarTexto(By referencia, String textoEsperado){
    return validarTexto(driver.findElement(referencia), textoEsperado);
}
```

Figura 11 - Validações Texto. Fonte: Autoria própria.

Para a validação de mensagens *Alert e PopUp foi* desenvolvida, na classe _pagina, o método validarAlerta que promove facilidade na criação de caso de teste a validação de mensagens *Alert e PopUp* apenas passando como parâmetro o elemento HTML e o texto esperado sendo exemplificado na Figura 12.

```
95    /** Realiza a validação do texto interno de um popup de alerta do JavaScript
96     * @param alerta : Popup de alerta que contém o texto a ser validado
97     * @param textoEsperado : Texto que deve ser validado no alerta **/
98    protected boolean validarAlerta(Alert alerta, String textoEsperado){
99        boolean validacao = true;
100       String textoEncontrado = alerta.getText().trim();
101       validacao = textoEncontrado.equals(textoEsperado);
102       if(validacao)
103           System.out.println("[TRUE]  Texto do alerta corresponde com texto esperado: '"+textoEsperado+"'!");
104       else
105           System.out.println("[FALSE] Texto do alerta: '"+textoEncontrado+"'" +
106                   " não corresponde com texto esperado: '"+textoEsperado+"'!");
107       return validacao;
108   }
```

Figura 12 - Validação Alert PopUp. Fonte: Autoria própria.

4.3.3 Pacote *Framework*

Para o pacote *framework* foi primeiramente desenvolvida a classe SeleniumMetodos, na qual implementa os métodos do *WebDriver* do *Selenium*. Estes, por possuir certa limitação e complexidade, serão tratados para facilitar a criação de casos de teste pelos analistas. Nesta linearidade, serão abordados a forma que se utiliza os métodos no *Selenium WebDriver* frente à comparação com a forma simplificada no *framework*, exemplos ilustrativos nas Figuras 13 e 14.

Na Figura 13, exemplifica-se a forma em que o Selenium aplica o texto "Teste Cadastro" em um campo de "*imput"* em aplicações *Web* sendo que na primeira linha executa o método "c*lear*" no qual limpa o campo e em seguida através do método "*sendKeys*" executa o preenchimento.

```
driver.findElement(By.id("pf_nome_cliente")).clear();
driver.findElement(By.id("pf_nome_cliente")).sendKeys("Teste Cadastro");
```

Figura 13 - Exemplo do Método *WebDriver*. Fonte: Autoria Própria.

Na Figura 14, exemplifica a forma em que os métodos do *Selenium* serão tratados no *framework* sendo que com apenas uma chamada do método "*type*" do *framework* executa três ações do *WebDriver* sendo eles "*waitForVisible*" que espera o elemento

estar visível na página para perseguir para próxima ação evitando a quebra na execução caso de teste que é muito comum quando se utiliza diretamente, "*clear*" faz a limpeza do campo, e através do método "*sendKeys*" executa o preenchimento, podendo ser verificado na Figura 15.

```
public void type(WebElement element, String value) {
    waitForVisible(element);
    element.clear();
    element.sendKeys(value);
}
```

Figura 14 - Exemplo do método tratado no *framework* Fonte: Autoria Própria.

```
public void PrencherCampos(Sheet planilha, int coluna){
    driver.type(nome, planilha.getCell(coluna, 2).getContents());
    driver.type(dataNascimento, planilha.getCell(coluna, 3).getContents());
}
```

Figura 15 - Exemplo aplicação do método do *framework* no *PageObject.* Fonte: Autoria Própria.

4.3.4 Pacote CasoTeste

Para organização dos casos de teste foi criado pacote CasoTeste onde deve ser implementado todas as classes referentes a cada caso de teste a ser realizado de acordo com cada aplicação ou sistema a ser testado com o *framework*, sendo que cada classe é um caso de teste ou cenário de teste, podendo haver variações do mesmo exemplo Figura - 16.

```java
package CasoTeste;

import java.io.File;
public class CadastroCliente extends TesteBase{

    @Test(description = "Cadastro de Valido")
    public void CadastroValido() throws BiffException, IOException{
        massa = Workbook.getWorkbook(new File(dirMassa+"Clientes.xls"));
        planilha = massa.getSheet("Clientes");
        cadastroCliente.PrencherCampos(planilha, 1);
        cadastroCliente.Cadastrar();
    }
    @Test(description = "Cadastro de Cliente")
    public void CadastroInvalido() throws BiffException, IOException{
        massa = Workbook.getWorkbook(new File(dirMassa+"ClientesInvalido.xls"));
        planilha = massa.getSheet("Clientes");
        cadastroCliente.PrencherCampos(planilha ,1);
        cadastroCliente.Cadastrar();
    }
}
```

Figura 16 - Exemplo da classe caso de teste. Fonte: Autoria própria.

Ainda é possível notar que em cada classe será informado o caminho da massa de dados utilizada no caso de teste, sendo uma planilha Excel a extensão .xls tomando cuidado para não ser .xlsx pois esse formato não é comportado pelo *framework*, podendo ser verificado na Figura 16 acima.

CAPÍTULO 5 APRESENTAÇÃO DA FERRAMENTA

Este capítulo tem por objetivo apresentar como se utiliza o *framework* passo a passo de forma prática, envolvendo todo o processo de abstração de informações referentes a aplicação a ser realizado o teste automatizado e também a organização dos artefatos envolvidos em cada caso de teste.

O principal objetivo desse *framework* é simplificar e otimizar a criação de casos de teste automatizados funcionais, para isso é necessário se definir a aplicação *web* que será testada, para alcançar a simplicidade e melhor aproveitamento é preciso para cada caso de teste seja criado os referentes *PageObject* e a Massa de dados.

Para se utilizar o *framework* primeiramente é preciso realizar as configurações gerais que na classe TesteBase do pacote Parametros, sendo eles *navegador*, *diretorioRaizEvidencias*, *diretorioRaizMassa*, *massaDados* e *diretorioImagens*. Após as configurações, é necessário definir a aplicação ou sistema *web* que será criado os casos de teste automatizados, de modo que para a demonstração será utilizado um sistema de anúncio de carros www.egaragens.com.br como se pode ser visto na Figura - 17.

Após definido a aplicação *web* a ser testada é preciso mapear os *PageObject* que são basicamente a separação de funcionalidades elementos do HTML da página onde o teste irá interagir, no qual passaram a ser modelos dentro do código. Assim, todo caso de teste que utilizar um elemento em comum, caso o mesmo tenha alguma alteração em seus identificadores, não será preciso alterar todos os casos de teste, somente será necessário alterar o *PageObject*, na Figura 17, a aplicação é destacada em verde, pontos no qual são definidos com *PageObject* da página.

Figura 17 - PageObject aplicação. Fonte: Autoria própria.

Após identificado os *PageObject* é preciso criar sua classe no *framework* no qual deve possuir @FindBy para cada elemento da página e será atributo um nome no qual representará esse elemento em qualquer caso de teste que utilize esse *PageObject*. Assim, quando esse elemento HTML for alterado, somente será necessário alterar o seu *PageObject*, como pode ser verificado na Figura 19. O uso da anotação @FindBy facilita a manipulação de um elemento no PageObject pois é possível atribuir um nome para elemento mais intuitivo pois em muitos casos somente é possível localizar o elemento na página através do XPATH e sua expressão não torna sua utilização fácil como exemplo (@FindBy(xpath= "id('busca2')/x:label/x:div/x:select") private WebElement tipoVeiculo;). Esse nome também será utilizado na massa de dados para identificar o elemento.

```
19 /** PageObject referente a Página de Inicial do eGaragens **/
20 public class PaginaInicial extends _Pagina{
21     @FindBy(id = "txtNome") private WebElement busca;
22     @FindBy(xpath = "id('busca2')/x:label/x:div/x:select") private WebElement tipoVeiculo;
23     @FindBy(name = "marca") private WebElement marcaVeiculo;
24     @FindBy(name = "modelo") private WebElement modeloVeiculo;
25     @FindBy(name = "cidade") private WebElement cidade;
26     @FindBy(name = "ano2") private WebElement anoInicial;
27     @FindBy(name = "anomodelo2") private WebElement anoFinal;
28     @FindBy(name = "preco") private WebElement precoInicial;
29     @FindBy(name = "preco2") private WebElement precoFinal;
30     @FindBy(xpath = "Buscar Meu Veículo") private WebElement btnBuscarMenu;
31     @FindBy(xpath = "busca-botao") private WebElement btnBuscarTexto;
32     @FindBy(id = "notfound") private WebElement naoEncontrado;
33     @FindBy(xpath = "id('cars-list')/x:ul/x:li[1]/x:a/x:div[1]/x:h4") private WebElement resultadoBusca;
34
```

Figura 18 FindBy PageObject. Fonte: Autoria própria.

Após definir os elementos foi criado os *PageObject* de interação da página na qual foi identificado a paginaInicial; buscaVeiculoNome; buscaVeiculoMenu e o validaResultado. Há métodos independentes que foram usados para criar os casos de teste, cada método utiliza os identificadores para se realizar as interações com a página através do construtor do driver do *SeleniumMedotos*. Pode-se utilizar as funções do *WebDriver* descritos no capitulo 4.2.3, também é no *PageObject* que se informa a posição das informações na massa de dados sendo passado por parâmetro a planilha e o identificador da coluna referente ao arquivo EXCEL, como exemplificado na Figura 20.

```
36      public PaginaInicial(SeleniumMetodos driver) {
37          super(driver);
38      }
39      /**Metodo acesso pagina inicial **/
40      public PaginaInicial abrirPagina(){
41          driver.get("http://www.egaragens.com");
42          return this;
43      }
44      /**Metodo  que preenche valor da busca com a informacao da massa de dados **/
45      public void buscaVeiculoNome(Sheet planilha, int coluna){
46          driver.type(busca, planilha.getCell(coluna, 2).getContents() );
47          btnBuscarTexto.click();
48      }
49      /**Metodo que busca dados na massa de dados e preemche os campos do menu de veiculos**/
50      public void buscaVeiculoMenu(Sheet planilha, int coluna){
51          driver.type(tipoVeiculo, planilha.getCell(coluna, 2).getContents() );
52          driver.type(marcaVeiculo, planilha.getCell(coluna, 3).getContents() );
53          driver.type(modeloVeiculo, planilha.getCell(coluna, 4).getContents() );
54          driver.type(cidade, planilha.getCell(coluna, 5).getContents() );
55          driver.type(anoInicial, planilha.getCell(coluna, 6).getContents() );
56          driver.type(anoFinal, planilha.getCell(coluna, 7).getContents() );
57          driver.type(precoInicial, planilha.getCell(coluna, 8).getContents() );
58          driver.type(anoFinal, planilha.getCell(coluna, 9).getContents() );
59          btnBuscarMenu.click();
60      }
61      public boolean validaResultado(Sheet planilha, int coluna){
62          resultadoBusca.equals(planilha.getCell(coluna, 10).getContents());
63          return true;
64      }
65  }
```

Figura 19 PageObejcts Egaragen. Fonte: Autoria própria.

Após definido os *PageObjets* da página é preciso criar a massa de dados e o caso de teste da aplicação, primeiro deve se criar a Classe de teste no Pacote CasosTeste que deve se extender a classe TesteBase, na classe de teste cada método será um caso de teste assim o mesmo dever ser identificado pela anotação @Test exemplo (@Test(description = "Busca Veiculo Menu")) dessa forma o TestNg identifica cada execução para gerar os resultados dos testes de forma organizada e intuitiva. Ao criar um caso de teste se deve passar os parâmetros referente a massa de dados a ser utilizado para sua execução. Tem-se o exemplo no qual utiliza as configurações gerais da classe TesteBase, na Figura 21.

Na criação de cada caso de teste, se necessário, a execução de variações dos casos de teste requer a aplicação de um laço que condiciona a quantidade de variações a serem executadas, representadas por uma coluna na massa de dados.

Para exemplificar o funcionamento do *framework* foi criado 3 casos de teste referente aos *PageObjetcs* da classe PaginaInicial sendo os casos de teste, Busca veículo por menu, Busca veículo por Nome e Busca veículo Invalida. Pode-se verificar a reutilização dos *PageObjetcs* em diversos casos de teste alterando apenas a sequência de chamada dos métodos, podendo formar fluxos de execução diferentes. Dessa forma, a medida que se desenvolve mais *PageObejcts* das páginas mais simples, se cria casos de teste, havendo alguma alteração na aplicação será necessário alterar apenas o *PageObjects* especifico, como pode-se ver na Figura 21.

```
11  public class BuscaVeiculo extends TesteBase{
12      @Test(description = "Busca veículo por Menu")
13      public void BuscaVeiculoMenu() throws BiffException, IOException{
14          int numVariacoes = 6;
15          massa = Workbook.getWorkbook(new File(dirMassa+"BuscaVeiculo.xls"));
16          for(int i=1; i<=numVariacoes; i++){
17              planilha = massa.getSheet("Menu"); /**Define aba da planilha **/
18              if(i<planilha.getColumns()){  /** condição para execuar a quantidade de campos**/
19                  paginaInicial.buscaVeiculoMenu(planilha, i);
20                  paginaInicial.validaResultado(planilha, i);
21          }
22        }
23      }
24      @Test(description = "Busca veículo por Nome")
25      public void BuscaVeiculoNome() throws BiffException, IOException{
26          int numVariacoes = 6;
27          massa = Workbook.getWorkbook(new File(dirMassa+"BuscaVeiculo.xls"));
28          for(int i=1; i<=numVariacoes; i++){
29          planilha = massa.getSheet("Nome");
30          if(i<planilha.getColumns()){
31              paginaInicial.buscaVeiculoNome(planilha, i);
32              paginaInicial.validaResultado(planilha, i);
33          }
34         }
35      }
36      @Test(description = "Busca veículo Invalido")
37      public void BuscaVeiculoInvalida() throws BiffException, IOException{
38          int numVariacoes = 6;
39          massa = Workbook.getWorkbook(new File(dirMassa+"BuscaVeiculo.xls"));
40          for(int i=1; i<=numVariacoes; i++){
41          planilha = massa.getSheet("BuscaInvalida");
42          if(i<planilha.getColumns()){
43              paginaInicial.buscaVeiculoNome(planilha,i);
44              paginaInicial.validaResultado(planilha, i);
45          }
46         }
47      }
```

Figura 20 - Caso de Teste Busca. Fonte: Autoria própria.

Para utilizar o benefício da massa de dados na criação de variações de execução dos casos de testes descrito acima, foi criada a planilha EXCEL “BuscaVeiculo.xls” na

qual possui 3 abas, sendo que cada uma delas é destina a um caso de teste. A planilha é estruturada, sendo a primeira coluna representada pelo nome dos elementos HTML do *PageObject*, na qual o conteúdo de cada variação será inserido ou validado na execução do deste na aplicação web, dessa forma cada coluna será uma variação de teste, como demonstra a Figura - 21.

	A	B	C	D	E	F	G
1	Campo	Variacão 1	Variacão 2	Variacão 3	Variacão 4	Variacão 5	Variacão 6
2	tipoVeiculo	Carro	Moto	Carro	Carro	carro, moto, caminhão	Carro
3	marcaVeiculo	Fiat	Honda - Motos	Ford	Chevrolet		Fiat
4	modeloVeiculo	Uno	CB 300 R	Fiesta	Astra		Palio
5	cidade	Todas	Todas	Todas	Todas	Marilia	Tupã
6	anoInicial	1995	2010	1998		1985	
7	anoFinal	2010	2013		2002	2005	2008
8	precoInicial	3000	7500			50000	
9	precoFinal	16500	11000		13500		14000
10	resultadoBusca	Fiat - UNO	Honda - Motos - Cb 300 R	Ford - FIESTA	Chevrolet - Astra	Ford - KA	Fiat - Palio

	A	B	C	D	E	F	G	H	I
1	Campo	Variaçao 1	Variaçao 2	Variaçao 3	Variaçao 4	Variaçao 5	Variaçao 6		
2	busca	Gol	Fiesta	Cg 125	Corsa	Vectra	Astra		
3	resultadoBusca	Volkswagen - GOL	Ford - FIESTA	Honda - Motos - Cg 125	Chevrolet - Corsa	Chevrolet - Vectra	Chevrolet - Astra		
4									

	A	B	C	D	E	F	G	H	I
1	Campo	Variaçao 1	Variaçao 2	Variaçao 3	Variaçao 4	Variaçao 5	Variaçao 6		
2	busca	YBR 150	CBX 350	Uno 2.0	Gol 2.4	Omega 1.0	Astra 1.0		
3	resultadoBusca	Desculpe n	Desculpe nã	Desculpe nã	Desculpe nã	Desculpe não encont	Desculpe não encontramos o veículo que		
4									

Figura 21 - Massa de dados casos de teste. Fonte: Autoria própria.

Para se executar os casos de teste desenvolvidos, é preciso criar um arquivo XML no qual se deve informar a classe de teste e os métodos no qual queria executar. A ação forma um suíte de teste na qual será interpretada pela TestNG, segue o exemplo na Figura - 22.

BuscaVeiculo.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Busca Veiculo">
    <listeners>
        <listener class-name="Parametros.Saidas" />
    </listeners>

    <test name="Cadastro cliente" preserve-order="false">
        <classes>
            <class name="CasosTeste.BuscaVeiculo">
            <methods>
             <include name="BuscaVeiculoMenu"/>
             <include name="BuscaVeiculoNome"/>
             <include name="BuscaVeiculoInvalida"/>
            </methods>
            </class>
        </classes>
    </test>

</suite>
```

Figura 22 XML Suíte de Teste. Fonte: Autoria própria.

Para executar a suíte de teste bastar executar o arquivo via Run As TesteNg Suite da IDE eclipse, que automaticamente irá chamar o navegador onde o webdriver irá aplicar seus comandos de acordo com o caso de teste definido. No momento da execução através dos tratamentos desenvolvidos da classe saídas já é possível acompanhar os resultados dos testes exemplificado na Figura 23 e o resultado apresentado pelo TestNG Figura 24.

Problems @ Javadoc Declaration Console Results of running suite

```
<terminated> BuscaVeiculo.xml [TestNG] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (21/11/2014 20:41:02)
[TestNG] Running:
  C:\Users\Thales\workspace\flamework\BuscaVeiculo.xml


[INICIO] Execução do teste "Busca veículo Invalido" iniciada.

[SUCESSO] Teste "Busca veículo Invalido" executado com Sucesso!

[INICIO] Execução do teste "Busca veículo por Menu" iniciada.

[SUCESSO] Teste "Busca veículo por Menu" executado com Sucesso!

[INICIO] Execução do teste "Busca veículo por Nome" iniciada.

[SUCESSO] Teste "Busca veículo por Nome" executado com Sucesso!

===============================================
Busca Veiculo
Total tests run: 3, Failures: 0, Skips: 0
===============================================
```

Figura 23 Saídas Execução. Fonte: Autoria própria.

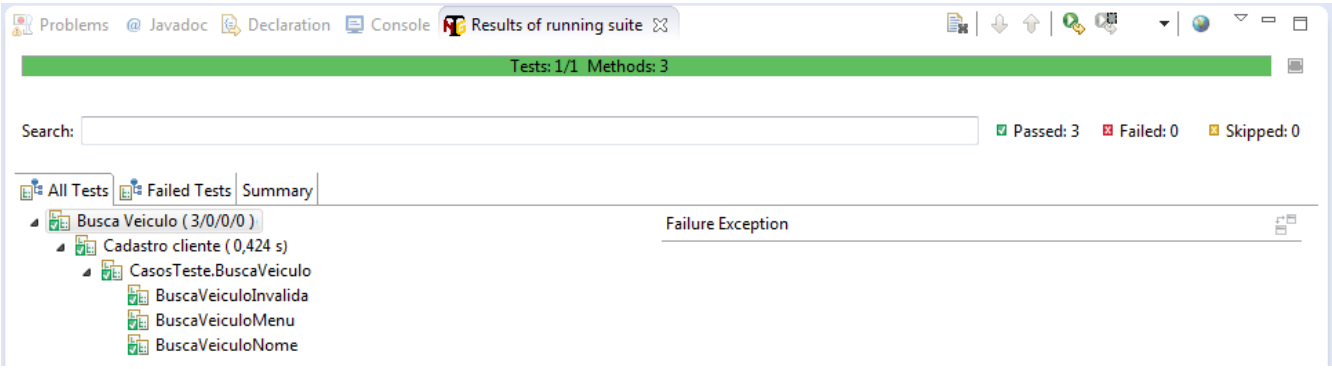

Figura 24 - Resultado TestNG. Fonte: Autoria própria.

Para demonstrar melhor o funcionamento do framework é demonstrado na Figura – 25, o diagrama de sequência no qual é possível ver a interação entres as classes do *framework.*

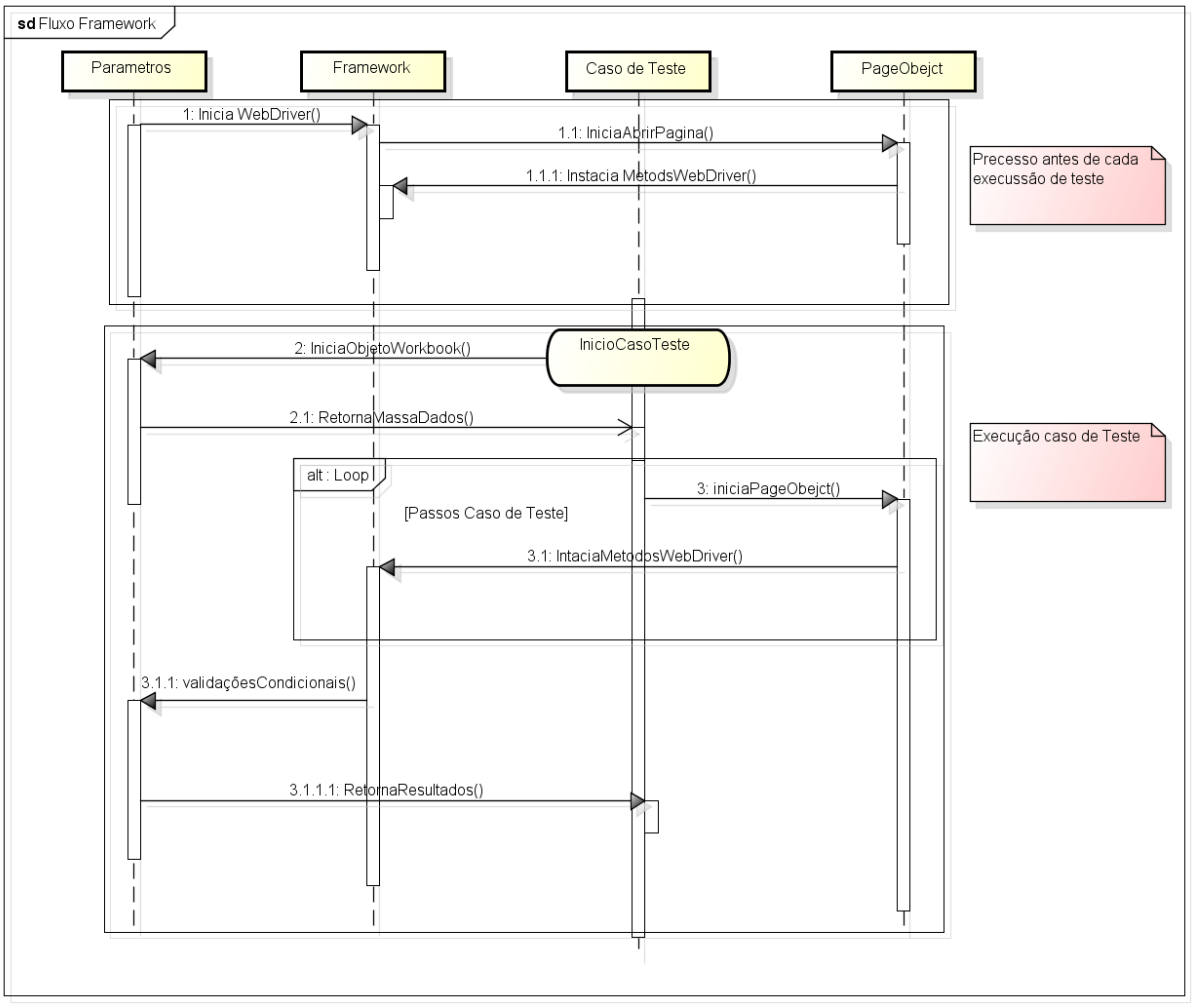

Figura 26 - Diagrama de sequência *Framework Fonte: Autoria própria.*

Esse *framework* foi desenvolvido especificamente para desenvolvimento de teste automatizados em aplicações *web* se limitando a realização de testes funcionais diretamente executado nas páginas da aplicação através do navegador virtual do Selenium, não atendendo o desenvolvimento de teste de unidade.

Para se utilizar esse framework o analista de teste ou programador deve possui conhecimentos básicos na linguagem de programação JAVA e conceitos de Orientação a Objetos afim de se fazer o uso dos benefícios da estrutura e classes desenvolvidas no framework, para alterações mais complexas nas funções do framework é necessário que que se tenha um conhecimento mais avançado em programação e nas funções específicas do WebDriver, estima-se que o analista e desenvolvedor esteja utilizando o framework com perfeição com 30 dias.

Essa pesquisa deixa embasamento para se desenvolver uma interface gráfica para o framework tornado a ferramenta mais intuitiva para o usuário, também a possibilidade de diversificar tipo de massa de dados a ser utilizada como conexão com banco de dados, arquivos XML e arquivos de texto.

Benefícios do uso do *framework* é dado através do uso do massa de dados que possibilita a separação dos dados do entrada do código e a possibilidade de se criar variações dos teste apenas adicionando dados de entrada, também possui ganho pela reutilização de código através do uso de funções genéricas, o *framework* também apresenta métodos do WebDriver simplificados no qual facilita o desenvolvimento dos casos de teste e redução de código, apresenta benefícios na organização dos casos de teste através da orientação a objetos aplicada na estrutura do framework e o principal benefício do framework é através do uso correto do concito de PageObject que abstrai a interface da aplicação a ser testada do caso de teste.

Para comparação do benéfico do uso do *framework* proposto foi criado um caso de teste pela ferramenta Selenium IDE que é uma extensão do navegador Firefox muito utilizada no mercado para a criação de casos de teste automatizados em aplicações web, trabalhado com método Record And Play, a Figura 26 – Selenium IDE apresenta a interface da ferramenta nessa o caso de teste criado, nota-se que é apenas um caso de teste e um variação do mesmo cenário demostrado acima no *framework* proposto, como não utiliza dados externos e não permite a criação de classes diretamente, para cada variação do caso de teste deve se criar um teste apenas alterando os dados de entrada, o Selenium IDE mesmo sendo muito rápido e eficiente na criação de casos de teste

simples não tem o mesmo benéfico quando se é preciso aplicar muitas variações de casos teste, outra vantagem do *framework* proposto está na possibilidade de se criar funções especificas como gerador de número de documentos e e-mails válidos para dados de entrada, já no Selenium IDE os dados são estáticos.

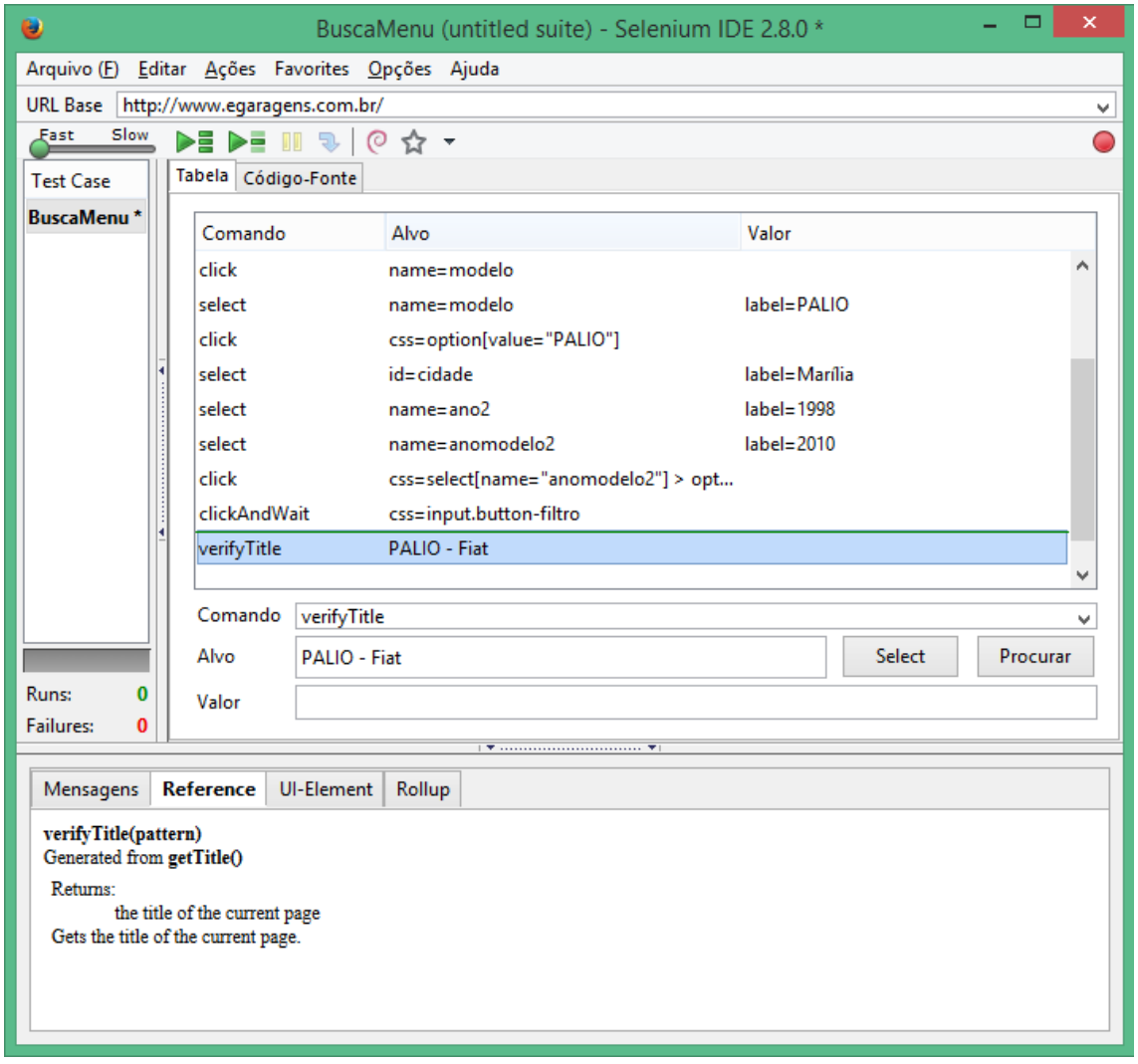

Figura 26 - Selenium IDE Fonte Autoria própria.

CAPÍTULO 6 CONCLUSÃO

Após revisão bibliográfica dos temas envolvidos verificou-se que a atividade de teste de *software* é de suma importância, e que a automação de teste funcional por meio de um *framework* orientado a dados obtém-se um ganho significativo no tempo de desenvolvido dos casos e planos de teste automatizado.

Este trabalho teve sua contribuição por meio do estudo das referentes áreas de teste de *software*, destacando sua importando no cenário de desenvolvimento visando buscar a qualidade. Mostrando como é possível através de tecnologias *open-source* aplicar teste funcionais e enfatizando a utilização da massa de dados para obter um ganho significativo no desenvolvimento de testes funcionais para aplicações *web*.

Levantado estratégicas, características e boas práticas para o desenvolvimento de um *framework* e aplicado na ferramenta desenvolvida no trabalho, que alcançou os objetivos de ser de fácil utilização, possuir uma estrutura lógica afim de organizar as funções do *framework*, que possui uma fácil manutenção, foi possível simplificar a utilização da linguagem de programação oferecendo um legibilidade ao analista de teste, a separação dos dados de teste do script que melhora a manutenibilidade do código e a criação de funções independentes que agiliza a criação de casos de teste.

O *framework* proposto é reutilizável e oferece uma forma simples e prática de se desenvolver casos e planos de teste, oferece uma simples forma de se aplicar variações de caos de teste somente através da na massa de dados.

Ao aplicar o *framework* proposto, é esperado que o analista de teste possa ter um ganho significativo no desenvolvimento dos casos de teste e suas variações, também que após um grande volume de caso de teste e *PageObject* seja desenvolvido se tenha facilidade para mantê-los atualizados e funcionando corretamente.

O *framework* não se limita a utilização por analistas de teste também podendo ser utilizado por desenvolvedores para se criar seus *scripts* de geração de dados e validação de alterações durante o desenvolvimento de uma aplicação.

Para uma futura atualização do framework, destaca-se a possibilidade de se utilizar outros tipos de massa de dados como banco de dados e arquivos texto, também a possibilidade de disponibiliza-lo como código *open-source* e a criação de funções e métodos que dê suporte a criação de casos de teste.

REFERÊNCIAS

BEIZER, B. **Software testing techniques.** New York: Van Nostrand Reinhold Co.1990

BURNSTEIN, Ilene. **Practical Software Testing: a process-oriented approach**. New York: Springer-Verlag, 2003.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAND, P. & STAL, M (1996): **Pattern-Oriented Software Architectur A System Of Patterns**, JOHN WILEY & SONS, 1996

CAETANO, Cristiano. **Automação de Testes:** Mitos e Verdades. São Paulo: Qualister. Disponível em: <http://www.slideshare.net/cristianocaetano/automao-de-testes-mitos-e-verdades-qualister>. Acesso em: 09 jun. 2014.

CALDAS, A.G.M. **Automação de testes funcionais a aplicação Java Swwing**. Portugal: FEUP. 2009.

COELHO, R.; KULESZA, U.; STAA, A. V.;LUCENA, C. **Unit testing in multi-agent systems using mock agents and aspects**. In: **SELMAS '06**: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems, pages 83–90, New York, NY, USA, 2006. ACM Press.

COELHO, R. Teste de Software. In: **XIX Simpósio Brasileiro de Engenharia de Software:** Mini curso. Uberlândia. 2005. Disponível em: <http://www.sbbdsbes2005.ufu.br/arquivos/minicursoSBES2005_TestesSoftware.pdf> Acesso em: 03 fev. 2014.

CHAIM, M. J. **Potential Uses Criteria Tool for Program Testing**. Campinas: DCA/FEE/UNICAMP. 1991.

DIAS NETO, A. C. **Uma Infra-estrutura Computacional para Apoiar o**

Planejamento e Controle de Testes de Software. Rio de Janeiro: COPPE/UFRJ. 2006.

DELAMARO, M. E.; MALDONADO, J.C.; JINO, M. Conceitos Básicos. In: _____. **Introdução ao Teste de Software**. Rio de Janeiro: Elsevier, 2007.

Eclipse, Documentação < http://www.eclipse.org/org/>. Acesso em: 8 maio 2014.

FEWSTER, Mark; GRAHAM, Dorothy. **Software Test Automation: Effective use of test execution tools**. Eua: Addison-wesley Professional, 1999.

IEEE Standard Glossary of Software Engineering Terminology. ANSI/IEEE Std 610.12, 1990.

FEUP CIQS - Inovação em Qualidade de Software. Disponível em < http://sigarra.up.pt/feup/pt/NOTICIAS_GERAL.VER_NOTICIA?P_nr=8514>.

Fayad, M. E., Schimidt & D. C., Johnson, R. E. (1999a): **Implementing application frameworks: object-oriented frameworks at work**. New York: J. Wiley, c1999. p 29.

Fayad, M. E., Schimidt & D. C., Johnson, R. E. (1999b): **Building application frameworks: object-oriented foundations of framework design**. New York: J. Wiley, c1999. p 664.

LEAL, G. CL. **Uma abordagem integrada de desenvolvimento e teste de software para equips distribuídas**. Maringá: UEM. 2010.

Mattsson, M. (1996): **Object-oriented Frameworks** - A survey of methodological issues", Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LUCS-TR: 96-167, Department of Computer Science, Lund University, 1996.

Maven Documentação. disponivel em < http://maven.apache.org/guides/index.html >. Acesso em: 15 maio de 2014.

MESQUITA, A. M. **Qualidade de conjuntos de testes de software de código aberto**: uma análise baseada em critérios estruturais. Goiás: UFG. 2011.

MODESTO, L. R. **Teste functional baseado em diagramas de UML**. Marília: UNESP. 2006

MYERS, G. J. **The Art of Software Testing, Wiley**. New Jersey: John Wiley and Sons. 2004.

Myers, G. **The Art of Software Testing**. New York: John Wiley & Sons. 1976.

NÓBREGA, R. O. **Framework para automação de testes funcionais utilizando o Rational Functional Tester**. Recife: UFP. 2006

PageObjects, Documentação. Disponível em < https://code.google.com/p/selenium/wiki/PageObjects>. Acesso em: 8 maio 2014.

PRESSMAN, R. S. **Engenharia de Software**. 5. ed. Rio de Janeiro: McGraw-Hill, 2002.

PRESSMAN, R. S. **Software Engineering**: A Practitioner's Approach. Nova York: McGraw-Hill, 2005.

PETTICHORD, Bret. Success with Test Automation, 2001.

Rational Unified Process: Visão Geral. Disponível em < http://www.wthreex.com/rup/process/ovu_proc.htm.>. Acesso em 10 maio 2014.

Rios, E., **Análise de Riscos em Teste de Software.** São Paulo: s.n.2012.

ROTTA NETO, João; SANTOS, Maria C. dos. **Testes de Software**: Uma Introdução e Exemplos. São Paulo: s.n. 2001.

Selenium, Documentação Introdução ao Selenium. Disponivel em < http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp>. Acesso em: 8 maio 2014.

SILVA, D. A. G. **EvolUniT**: **Geração e Evolução de Testes de Unidade em Java utilizando Algoritmos Genéticos**. 130 F. 2008. Dissertação de Mestrado (*Mestrado em ciência da computação) - CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO, Pernambuco, 2008.*

TestNG, Documentação. Disponiel em<http://testng.org/doc/documentation-main.html#introduction>. Acesso em: 8 maio 2014.