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

ANDRESSA GARCIA BARBOSA

DESENVOLVIMENTO DE UM FRAMEWORK PARA APLICAÇÃO DE TESTE UNITÁRIO ORIENTADO A DADOS

MARÍLIA
2012

ANDRESSA GARCIA BARBOSA

DESENVOLVIMENTO DE UM FRAMEWORK PARA APLICAÇÃO DE TESTE UNITÁRIO ORIENTADO A DADOS

Trabalho de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação da Fundação de Ensino “Eurípides Soares da Rocha”, mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador
Profº Ms. Fabio Lucio Meira

MARÍLIA
2012

Barbosa, Andressa Garcia
Desenvolvimento de um Framework para Aplicação de Teste Unitário Orientado a Dados / Andressa Garcia Barbosa; orientador: Fábio Lúcio Meira. Marília, SP: [s.n.], 2012.
59f.

Trabalho de Curso (Graduação em Sistemas de Informação) – Curso de Sistemas de Informação, Fundação de Ensino “Eurípides Soares da Rocha”, mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2012.

1. Teste de Software 2. Automação de Teste 3. Teste Unitário

CDD: 005.1

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Andressa Garcia Barbosa

Desenvolvimento de um Framework para Aplicação de Teste Unitário 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: 9.5 (nove e meio)

Orientador: Fabio Lucio Meira ____________________

1º. Examinador: Adriano Bezerra ____________________

2º. Examinador: Giulianna Marega Marques ____________________

Marília, 10 de dezembro de 2012.

Dedico este trabalho a Deus em primeiro lugar, por ter possibilitado a chegada até aqui, e por todos os momentos de dificuldade e superação durante a graduação.

Ao meu amado vovô, Mauro Aparecido Barbosa (in memoriam), que acompanhou somente metade dessa trajetória de uma fase tão importante em minha vida, sua ausência é muito sentida neste momento e será para sempre em toda minha vida.
Obrigada por ter sido o meu avô, pelos ensinamentos e tudo que me proporcionou em todos os momentos que passamos juntos, e principalmente, pelo seu carinho tão explicitamente demonstrado por mim desde quando nasci.
Te amo muito!
Saudades eternas.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter guiado os meus passos até aqui.

As minhas avós Rosa de Iório Garcia e Alaíde Luiz pelo cuidado comigo até os dias de hoje.

A minha mãe Rosana Garcia e meu pai Mauro André Barbosa por tudo que tem dedicado à minha vida.

Ao meu tio, Marcelo Alexandre Barbosa, pelo apoio e carinho.

As minhas irmãs, Drielle Garcia Barbosa, Maísa Garcia Barbosa e Alessandra Garcia Barbosa por todos os momentos bons e ruins que passamos juntas.

Ao meu noivo, Marcelo Santiago Bortoloci, por ter sido em vários momentos críticos peça fundamental para concretização deste trabalho.

Ao meu querido amigo Marcelo Augusto Omoto, pelo incentivo e conselhos.

Ao meu orientador Fábio Lúcio Meira, por ter desempenhado seu papel em prol dessa pesquisa.

Ao professor Adriano Bezerra por compartilhar experiências e ensinamentos em nossa área em comum.

A professora Giulianna Marega Marques, pelas valiosas dicas e incentivos.

A todos os funcionários do Univem, tesouraria, professores, secretaria, coordenação do curso, biblioteca em especial Valdir (tesouraria) e Aninha (Xerox).

“A motivação para a conquista não deve superar a motivação para o preparo.”
Pâmela Patrícia Correa da Silva

BARBOSA, Andressa Garcia. **Desenvolvimento de um framework para aplicação de teste unitário orientado a dados**. 2012. 59 f. Trabalho de Curso (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2012.

RESUMO

A presente monografia tem como objetivo o desenvolvimento de um *framework* para aplicação de teste unitário orientado a dados. O teste de *software* é uma área crítica dentro do desenvolvimento de *software*, sua necessidade é cada vez mais evidente nos cenários atuais. Existem diversos tipos de testes podem ser aplicados durante todo o processo de desenvolvimento do *software* de modo que garanta a qualidade do mesmo, contudo, o teste unitário é a maneira relativamente mais fácil e de menor custo que pode ser utilizada para aumentar a produtividade no desenvolvimento de *software*. A automação de teste no contexto de teste unitário favorece e viabiliza ainda mais sua aplicação no desenvolvimento de *software*. Ao unir a abordagem de automação de teste orientado a dados com o teste unitário, obtêm-se vantagens expressivas que refletem na qualidade do *software* e aumenta a produtividade na criação de *scripts* de testes automatizados e de casos de teste, por consequência eleva a qualidade do *software*.

Palavras-Chave: Teste de Software. Automação de Teste. Teste Unitário.

BARBOSA, Andressa Garcia. **Desenvolvimento de um framework para aplicação de teste unitário orientado a dados**. 2012. 59 f. Trabalho de Curso (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2012.

ABSTRACT

This monograph aims to study the development of a framework for implementation of data-driven unit test. Software testing is a critical area within the software development, their need is increasingly evident in the current scenarios. Several types of tests can be applied throughout the software development process in order to ensure its quality, however, the test unit is a relatively easy and low cost that can be used to increase productivity in the development software. The test automation in the context of unit testing encourages and enables further its application in software development. By joining the approach test automation with data-driven unit test, significant advantages are obtained that reflect the quality of the software and increases productivity in creating automated test scripts and test cases, consequently elevates the quality of software.

Keywords: Soffware Testing. Automation Testing. Unit Testing.

LISTA DE ILUSTRAÇÕES

Figura 1 – O custo para corrigir *bugs* pode aumentar dramaticamente ao longo do tempo.....17
Figura 2 – Modelo "V" de Teste de Software 17
Figura 3 – Exemplo de um processo básico de desenvolvimento de software e o objetivo do trabalho 19
Figura 4 – Ciclo Fundamental do TDD 24
Figura 5 – Funcionalidades do Software x Testes de Regressão.....29
Figura 6 – Pirâmide da Automação de Testes 30
Figura 7 – Fonte de Dados – *Data Driven Testing*.....33
Figura 8 – *Script* de Teste em Pseudo-linguagem 33
Figura 9 – Exemplo 1: Classe Calculadora.class.....37
Figura 10 – Exemplo 1: Classe TesteCalculadora.java 37
Figura 11 – Classes do JUnit 38
Figura 12 – Apresentação dos resultados do teste pelo JUnit 39
Figura 13 – Exemplo 2: TesteCalculadora 39
Figura 14 – Exemplo 2: TesteCalculadora – Teste parametrizado.....40
Figura 15 – Funcionamento do *framework* 41
Figura 16 – Layout da planilha de dados.....43
Figura 17 – Layout com informações adicionais que não contemplam a parametrização.....44
Figura 18 – Layout para teste de valores nulos 45
Figura 19 – Vinculando palavras aos métodos a serem testados.....45
Figura 20 – Vinculando palavras aos métodos.....46
Figura 21 – Classe de teste com execução variada de acordo com a palavra definida na planilha 46
Figura 22 – Testes Cenário 1: Método Somar.....47
Figura 23 – Testes Cenário 1: TesteCalculadora.java 48
Figura 24 – Testes Cenário 1: Fonte de Dados do Teste 48
Figura 25 – Testes Cenário 1: Resultado JUnit 49
Figura 26 – Testes Cenário 2: Classe de teste e resultado JUnit 50
Figura 27 – Testes Cenário 2: Fonte de Dados do Teste 50
Figura 28 – Teste cenário 3: Fonte de dados para validação da função de desconto 52
Figura 29 – Teste cenário 3: classe de teste 52
Figura 30 – Testes Cenário 3: resultado do JUnit 53

LISTA DE ABREVIATURAS E SIGLAS

API: Application Programming Interface, em português, Interface de Programação de Aplicativos;
IDE: Integrated Development Environment, em português, Ambiente Integrado de Desenvolvimento;
XML: eXtensible Markup Language, em português, extensão para linguagens de marcação.

SUMÁRIO

INTRODUÇÃO ... 10

CAPÍTULO 1 – TESTE DE SOFTWARE ... 13
1.1 Conceitos e Princípios ... 13
1.2 Processo e Fases de Teste ... 15
1.3 Testes Unitários ... 21
1.4 Desenvolvimento Dirigido por Testes (TDD) ... 22
1.5 Técnicas de Teste ... 24
1.5.1 Teste Funcional (Caixa Preta) ... 25

CAPÍTULO 2 – AUTOMAÇÃO DE TESTES ... 28
2.1 Conceito ... 28
2.2 Abordagens para Automação de Testes Funcionais ... 31
2.3 Data-Driven Testing ... 32

CAPÍTULO 3 – DESENVOLVIMENTO ... 35
3.1 Tecnologias Envolvidas ... 35
3.1.1 O Framework JUnit ... 36
3.2 Arquitetura e Funcionamento do Framework Proposto ... 41

CAPÍTULO 4 – APRESENTAÇÃO ... 43
4.1 Como Utilizar o Framework ... 43
4.2 Testes Realizados e Resultados Obtidos ... 47

CONCLUSÃO ... 54
REFERÊNCIAS ... 56

INTRODUÇÃO

Os sistemas de informação estão se tornando cada vez mais importante para as organizações e pessoas, na mesma proporção estão se tornando maiores e mais complexos, em consequência a importância da qualidade de *software* também se torna maior. Isso ocorre porque uma falha no *software* pode causar grandes prejuízos financeiros, de tempo, imagem e reputação e pode ferir até a integridade das pessoas. Em alguns casos o prejuízo causado por uma falha no *software* é imensurável e irreversível. Além disso, soma-se a este fator a intolerância ao tempo de entrega, tornando os cronogramas cada vez mais curtos (BURNSTEIN, 2003).

A necessidade de maior qualidade e menor tempo para entrega do *software* impacta não somente as equipes de desenvolvimento e teste de *software*, mas também toda a disciplina de engenharia de *software*, neste aspecto, pode-se constatar as mudanças de paradigmas e de metodologias de desenvolvimento que vem acontecendo nos últimos anos (CRISPIN; GREGORY, 2009).

A indústria de *software* vem respondendo às demandas e exigências do mercado com grandes obras que incluem metodologias, técnicas, ferramentas e paradigmas que envolvem todas as áreas do ciclo de desenvolvimento de *software*. Com a ascensão de metodologias ágeis, por exemplo, muitos paradigmas foram quebrados principalmente em relação às equipes de desenvolvimento e teste, ambas agora são responsáveis pela qualidade do *software*, ambas utilizam ferramentas e técnicas para reduzir o risco de falhas no *software* em produção. Programadores testam e testadores também programam testes automatizados a fim de reduzir a grande demanda por testes, possibilitando produtividade, agilidade e maior qualidade no *software*. Neste cenário, as equipes devem encontrar nas metodologias, técnicas e ferramentas a maneira mais produtiva de produzir *software*s com qualidade (CRISPIN; GREGORY, 2009).

Embora o teste de *software* seja uma atividade requerida atualmente, comumente ela não é realizada de forma sistemática devido a fatores já mencionados como limitações de tempo, recursos, qualificação técnica dos envolvidos, além da complexidade dos sistemas e sua rápida evolução (CRISPIN; GREGORY, 2009).

Com isso, a automação de teste tem sido vista como uma das principais medidas para melhorar a eficiência dessa atividade, além de aliviar sua carga de trabalho (FEWSTER;

GRAHAM, 1999). Neste contexto, esta pesquisa elencou as principais abordagens para automação de testes e ferramentas que tornam essa atividade possível.

Ao considerar o teste unitário neste contexto, é possível encontrar facilmente diversos *frameworks* para criação e execução de testes unitários automatizados, e as vantagens em seu uso são refletidas principalmente na qualidade do *software* (OSHEROVE, 2009). A relação entre desenvolvimento de *software*, qualidade, automação de testes e teste unitário é apresentada a seguir.

No ciclo de desenvolvimento de *software*, os testes estão presentes com a finalidade de encontrar erros o mais cedo possível minimizando os custos de manutenção do *software*, por esse motivo o teste de *software* é uma atividade crítica dentro do processo de desenvolvimento de *software* e necessária para qualidade de um *software* (PRESSMAN, 2001).

Atualmente, existem variadas técnicas e tipos de testes que devem ser executados ao longo do ciclo de vida de desenvolvimento de *software*, e, alguns destes testes exigem o envolvimento de usuários finais e outras exigem uma equipe de teste e qualidade de *software* dedicada e integrada à equipe de desenvolvimento, além de outros recursos que exigem investimento inicial considerável (BASTOS et al., 2007).

O teste unitário é aplicado na fase de construção do *software*, criado e executado por programadores, uma maneira relativamente de baixo custo e de fácil aplicação na produção de códigos mais eficientes, sua adequada aplicação possibilita vantagens expressivas para o sucesso do projeto, além de ser um dos pilares para o desenvolvimento ágil de *software* (HUNT; THOMAS, 2003).

A aplicação de testes unitários aumenta a produtividade na codificação do *software*, e, na abordagem de testes funcionais, possibilita maior cobertura das funções específicas do código, aumentando a confiabilidade do *software* (HUNT; THOMAS, 2003).

Além disso, através da abordagem de teste unitário orientado a dados, é possível otimizar a criação de scripts de teste, evitando duplicação de código além de aperfeiçoar a criação de cenários possibilitando também, a reutilização destes cenários (FANTINATO, 2004).

A fim de proporcionar uma alternativa às equipes de desenvolvimento de *software* no que diz respeito à qualidade, o objetivo principal deste trabalho é aperfeiçoar a criação de teste unitário por meio de um *framework* para criação de teste unitário dirigido a dados.

Para obter êxito no objetivo principal da pesquisa, foi necessário atingir as metas definidas nos objetivos específicos que contempla: domínio das metodologias, técnicas e

ferramentas envolvidas, estudo de *frameworks* existentes e desenvolvimento de um *framework* que otimize a criação de testes unitários orientado à dados.

Baseando-se nestes objetivos específicos o projeto foi organizado em tarefas que refletem na metodologia do *framework*, dividindo-o em duas fases: 1. Iniciando com a fase de revisão bibliográfica: estudos das metodologias envolvidas, estudo das tecnologias envolvidas e estudo de *frameworks* existentes para teste unitário; 2. Fase em que foi proposta a elaboração da especificação funcional do *framework*, definição da arquitetura, desenvolvimento e validação.

Com a revisão bibliográfica e estudo das metodologias e tecnologias envolvidas definidas na primeira fase do projeto, foi possível definir e refinar o escopo da pesquisa e elaborar os capítulos 1 e 2. A segunda fase do projeto foi contemplada através de estudos de *framework*s existentes, possibilitando a criação da proposta e elaboração da especificação funcional do *framework*, para em seguida, iniciar com o desenvolvimento e validação, tornando possível a elaboração dos capítulos 3 e 4. Nesta fase de desenvolvimento do *framework* utilizou-se a linguagem de programação Java em conjunto com o *framework* para teste unitário JUnit[1] além da API (Interface de Programação de Aplicativos) Java para aplicações *Microsoft:* projeto Apache POI[2] no qual possibilitou integração com o Excel.

Para que se possa apresentar detalhadamente toda a pesquisa desenvolvida, este trabalho está organizado em quatro capítulos da seguinte maneira:

Capítulo 1 – Teste de *Software*: Discorre-se sobre a problemática endereçada em termos de motivação e objetivos, bem como as peculiaridades da área de teste de *software*, definindo o escopo deste trabalho.

Capítulo 2 – Automação de Teste: Apresentação de conceitos e técnicas de automação de teste, abordando suas principais características, definindo o escopo do trabalho neste contexto.

Capítulo 3 – Desenvolvimento: Tecnologias envolvidas no projeto e arquitetura do *framework* proposto.

Capítulo 4 – Apresentação: Apresentação do *framework* e testes realizados.

---

[1] JUnit: http://www.junit.org/

[2] Apache POI: http://poi.apache.org/

CAPÍTULO 1 – TESTE DE SOFTWARE

Neste capítulo, é apresentada uma revisão bibliográfica com o objetivo de mostrar, numa ótica geral, a definição da área de conhecimento de teste de *software* e suas peculiaridades no contexto da área agregada de desenvolvimento de *software*, estabelecendo o escopo no qual este trabalho se insere.

1.1 Conceitos e Princípios

A complexidade da construção de um sistema se dá pela complexidade de suas características e dimensões, por isso, está sujeito a variados tipos de problemas que impactam no resultado final, obtendo-se um produto diferente do esperado (DELAMARO; MALDONADO; JINO, 2007, cap.1, p.1-2).

Diversos fatores podem ser identificados como causas destes problemas, mas em sua maioria a origem é única: erro humano. Isto ocorre porque a construção de um *software* envolve e depende principalmente da habilidade, interpretação e execução das pessoas que o constroem, possibilitando a aparição de erros mesmo com a utilização de metodologias de engenharia de *software* (DELAMARO; MALDONADO; JINO, 2007, cap.1, p.1-2).

Atualmente é comum encontrar usuários de sistemas que tiveram alguma experiência frustrada na utilização de um *software* ou sistema, os sistemas quando não funcionam corretamente ou como esperados podem concretizar riscos de perdas financeiras, tempo, reputação e até a integridade das pessoas. Isso ocorre porque o processo de desenvolvimento de *software*, mesmo com uma série de atividades, métodos e ferramentas está sujeito a erros, pois o ser humano é sujeito a erros (engano) e passíveis a falhas. Um defeito (falha, bug) no código, *software*, sistema ou documento é produzido por um erro fazendo com que o sistema falhe ao tentar fazer o que deveria ou fazendo o que não deveria (ISTQB, 2011).

Pode-se observar que estes problemas provenientes da construção de um *software*, são divididos e diferenciados nos termos: defeito, engano, erro e falha. Segundo o padrão IEEE 610.12-1990 (IEEE, 1990), um defeito (*fault, bug*) é uma sentença, processo, passos ou definição de dados incorreta; engano (*mistake*) é a ação humana que produz um resultado incorreto; erro (*error*) é a diferença entre o valor observado e o valor esperado e falha (*failure*) é o desvio ou incapacidade do componente ou sistema de executar suas funções requeridas.

A execução rigorosa de atividades de teste para avaliação da qualidade do produto, pode reduzir consideravelmente a probabilidade de ocorrência de riscos de defeitos e problemas, identificando-os antes de implantados em produção, evitando impactos e perdas à(s) organização(ões). Além disso, os resultados da execução dos testes aumentam a confiabilidade na qualidade do *software* possibilitando medições e avaliações de suas características, requisitos funcionais e não funcionais (ABNT NBR ISO/IEC 9126-1, 2003).

A compreensão do verdadeiro significado de teste de *software* faz profunda diferença no sucesso de seus esforços, e a interpretação equivocada do termo impacta diretamente no objetivo e na eficiência dos testes, pois ao estabelecer um objetivo adequado, haverá um importante efeito psicológico que influenciará nos testes, por exemplo, na definição: "O teste é o processo de demonstrar que não existem erros no *software*", está estabelecido o objetivo de não encontrar erros no *software*, com isso, subconscientemente o responsável pelo teste será dirigido em direção a esse objetivo, ou seja, com forte tendência a selecionar dados de teste com baixa probabilidade de causar o programa ao fracasso, impactando diretamente na eficiência dos testes. Logo, se o objetivo é demonstrar que um *software* possui erros, os dados de teste terão maior probabilidade de encontrá-los (MYERS, 2004).

Ao testar um *software*, se deseja agregar valor, elevar a qualidade e confiabilidade através da identificação e remoção de erros, portanto, não se deve testar um programa para mostrar que ele funciona, mas do princípio de que o programa contém erros e que o teste (investigação) para ser bem sucedido deve-se procurar encontrar o maior número de erros possíveis (MYERS, 2004).

Podemos definir teste de *software* como sendo um processo controlado que verifica a qualidade do *software* através da identificação de erros cuja presença é assumida.

Segundo o *Syllabus* (ISTQB, 2011), temos como um dos princípios do teste de *software*: o teste demonstra a presença de defeitos, reduzindo a probabilidade de defeitos que permaneceram no *software*, mas caso não sejam encontrados, não é provado que o *software* está livre de defeitos.

Portanto, um caso de teste bem sucedido é aquele que promove a falha no programa. Dada esta definição, é apropriado determinar se é possível testar um programa para encontrar todos os seus defeitos. Em geral, é impraticável e muitas vezes impossível encontrar todos os erros de um programa (MYERS, 2004).

Ainda segundo o *Syllabus* (ISTQB, 2011), existe o princípio onde o teste exaustivo é impossível, testar todas as combinações de entradas e pré-condições não é viável, sugere-se levar em consideração riscos e prioridades com intuito de focar os esforços de teste.

A decisão de quanto teste é suficiente deve ser suposta pela equipe de teste ao estabelecer estratégias de teste considerando as variáveis: nível de risco técnico, de negócio e do projeto, restrições do projeto como tempo e orçamento, além de informações que deverão ser provenientes aos interessados (*stakeholders*) para tomada de decisões como a distribuição do *software*, próximas fases do desenvolvimento e implantação (ISTQB, 2011).

Mensurar o esforço de teste envolve implicações ao elaborar estratégias de teste, onde se consideram técnicas e critérios de teste baseando-se na cobertura desejada dos requisitos do *software* considerando os riscos dos mesmos (MYERS, 2004).

Duas das estratégias mais comuns incluem testes funcionais (caixa-preta) e testes estruturais (caixa branca), que serão tratados com mais detalhes nas próximas seções. Outra boa estratégia é o teste antecipado, conforme um dos princípios do *Syllabus* (ISTQB, 2011), a atividade de teste deve ser iniciada o mais breve possível dentro do ciclo de desenvolvimento de *software* e deve possuir seus objetivos definidos.

1.2 Processo e Fases de Teste

Comumente o processo de teste de *software* é visto como se consistisse apenas da fase de execução, por exemplo, execução do *software*. Esta fase é contemplada como uma parte do processo de teste, que consiste em atividades antes e depois da fase de execução (ISTQB, 2011).

As atividades do processo de teste podem variar de acordo com a metodologia adotada pela equipe de teste, de modo geral, o processo de teste básico, deve consistir nas seguintes atividades: planejamento de testes, modelagem e projeto de casos de teste, execução e avaliação dos resultados de teste (PRESSMAN, 2001).

No processo de teste de *software* essas atividades são organizadas com metodologia própria, divididas em fases que contemplam diferentes objetivos de teste, onde, diferentes pontos de vista levam a diferentes objetivos e a diferentes técnicas de teste para atingir estes objetivos (ISTQB, 2011).

Pode-se estabelecer de uma forma geral, como fases do processo de teste de *software*: teste de unidade, teste de integração e teste de sistemas (DELAMARO; MALDONADO; JINO, 2007, cap.2, p.4). Ainda segundo os autores, destaca-se o “teste de regressão”, fase de testes executada durante a manutenção do *software*, onde é necessário realizar testes que mostrem que as modificações efetuadas no *software* não introduziram

novos defeitos, considerando novos requisitos implementados (se for o caso) e se requisitos anteriormente testados continuam válidos.

De acordo com o *Syllabus* (ISTQB, 2011), as características de um bom teste em qualquer modelo de desenvolvimento de *software* devem envolver: 1) cada nível de teste deve possuir um objetivo específico para o nível correspondente; 2) para todas as atividades do desenvolvimento deve haver uma atividade de teste correspondente; 3) a análise e modelagem do teste para um determinado nível devem começar durante a atividade de desenvolvimento correspondente ao nível, e por fim, 4) testadores devem envolver-se na revisão de documentos antecipadamente.

Segundo Pressman (2001), os primeiros testes planejados e executados geralmente têm o objetivo de validar componentes individuais (unidades), conforme avança o teste, o foco muda em uma tentativa de encontrar erros em agrupamentos integrados de componentes (integração) e, finalmente, em todo o sistema.

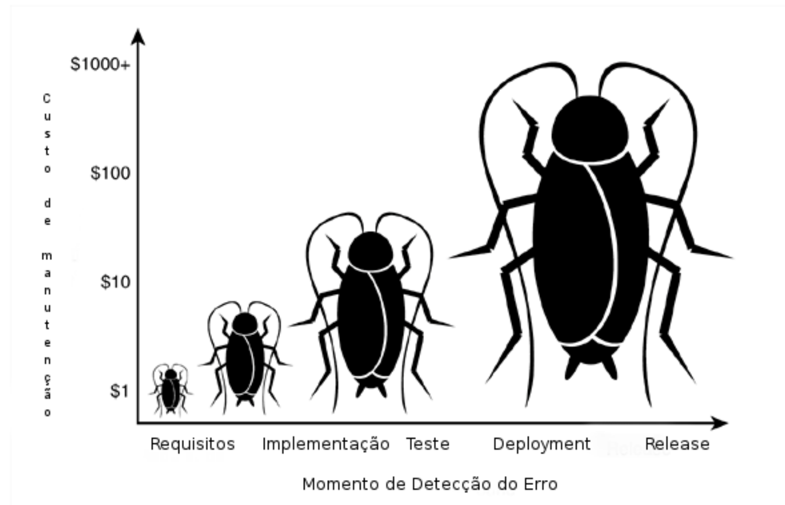
Neste contexto, de modo geral, o objetivo dos testes nas fases de teste de unidade e teste de integração pode ser causar o maior número de falhas possíveis de modo que estes defeitos possam ser identificados e solucionados. Já na fase de testes de sistemas o objetivo principal, pode ser confirmar se os requisitos foram implementados conforme a especificação (ISTQB, 2011).

As atividades provenientes do teste de *software* devem começar paralelamente às atividades do projeto de desenvolvimento, dessa forma, ao tratar os testes como um processo organizado, integrado e paralelo ao processo de desenvolvimento, os custos de manutenção serão reduzidos (BASTOS et al., 2007).

Segundo Patton (2006), os custos para correção de *bugs* são de escala logarítmica, pois aumentam em dez vezes à medida que o tempo passa e as fases do desenvolvimento e testes são concluídas. Um *bug* encontrado nas fases iniciais, por exemplo, na fase de especificação dos requisitos, pode não custar nada, ou US $ 1 no exemplo do autor (Figura 1). O mesmo *bug* se for encontrado nas fases de codificação e teste, o custo para correção pode variar de US $ 10 a US $ 100. Se um cliente encontra, o custo poderia facilmente ser de milhares ou até milhões dependendo da criticidade do *software* ou funcionalidade.

Por meio da figura 1 (próxima página) examina-se o aumento do custo de correção de defeitos à medida que as fases de desenvolvimento de *software* vão avançando, esse é um dos motivos pelo qual a atividade de teste deve iniciar paralelamente com as atividades de desenvolvimento.

Figura 1 – O custo para corrigir *bugs* pode aumentar dramaticamente ao longo do tempo.

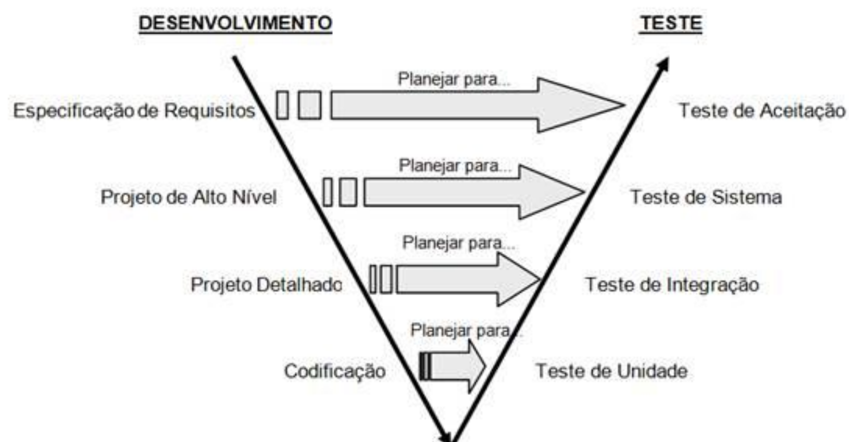


Fonte: Adaptação do livro – *Software Testing* de Ron Patton (2006)

Desta forma, o ciclo de testes pressupõe que sejam realizados testes ao longo de todo o processo de desenvolvimento, em que utilizando as mesmas informações, desenvolvimento e teste têm inicio das atividades simultaneamente, partindo do mesmo ponto, por exemplo, a equipe de desenvolvimento inicia o processo de desenvolvimento do sistema e a equipe de teste inicia o planejamento do processo de teste do *software*, definindo estratégias, ferramentas, técnicas entre outros artefatos provenientes da área de teste de *software* (BASTOS et al., 2007).

Pode-se examinar na Figura 2 o paralelismo entre as atividades de desenvolvimento e de teste de *software*, definindo o "Modelo V".

Figura 2 – Modelo "V" de Teste de *Software*



Fonte: Adaptação do livro *Systematic Software Testing* de Rick D. Craig e Stefan P. Jaskiel

Observa-se na figura acima os níveis de teste e sua ordem de planejamento e execução considerando os níveis e artefatos produzidos pelo desenvolvimento. Dessa forma, o teste de aceitação que é o último a ser executado, é o primeiro a ser planejado, pois ele é construído sobre os artefatos que são disponibilizados primeiro (especificação dos requisitos), e os testes unitários, por exemplo, são planejados com base na codificação, no projeto detalhado em alto nível, design e requisitos (CRAIG; JASKIEL, 2002).

Ainda de acordo com Craig e Jaskiel (2002), os níveis de teste podem variar de acordo com determinado ambiente considerando as variáveis: pessoas, hardware, interfaces, dados e até ponto de vista dos testadores, dessa forma, algumas empresas como exemplificado pelo autor, podem possuir apenas um nível de teste e outras 10 níveis, o que é determinado com base na complexidade do *software*, número de usuários, política, orçamento, pessoal, estrutura organizacional entre outros.

Segundo Rocha (2001, apud NETO [s.d.]), no planejamento dos testes por níveis, pode-se definir os principais níveis:

- Teste de unidade: conhecido também como testes unitários, tem o objetivo de explorar a menor unidade de código do *software*, a fim de provocar falhas por defeitos de lógica e de implementação em cada módulo separadamente;
- Teste de integração: possui o objetivo de provocar falhas às interfaces entre os módulos quando integrados para construir a estrutura do *software*;
- Teste de sistema: avalia o *software* em busca de falhas por meio da utilização do mesmo partindo do ponto de vista do usuário considerando suas ações no *software*;
- Teste de aceitação: nível de teste baseado nas necessidades dos usuários do sistema e demonstra que os requisitos foram atendidos;
- Teste de regressão: os testes de regressão não correspondem a um nível de teste, porém de suma importância, sua utilização na estratégia de teste possibilita redução significativa de eventuais falhas que podem ocorrer no *software* a cada alteração, independente da área afetada pela alteração.

Considerando as fases de desenvolvimento e teste, Rex Black em seu livro *Managing The Testing Process* (1999), avalia que ao criar um projeto de desenvolvimento de *software*, deve-se criar em paralelo o projeto de teste para validação e verificação do *software*, identificando o escopo do projeto de teste, determinando fases e seus objetivos, técnicas, ferramentas, ambiente de teste, riscos do projeto de teste entre outros.

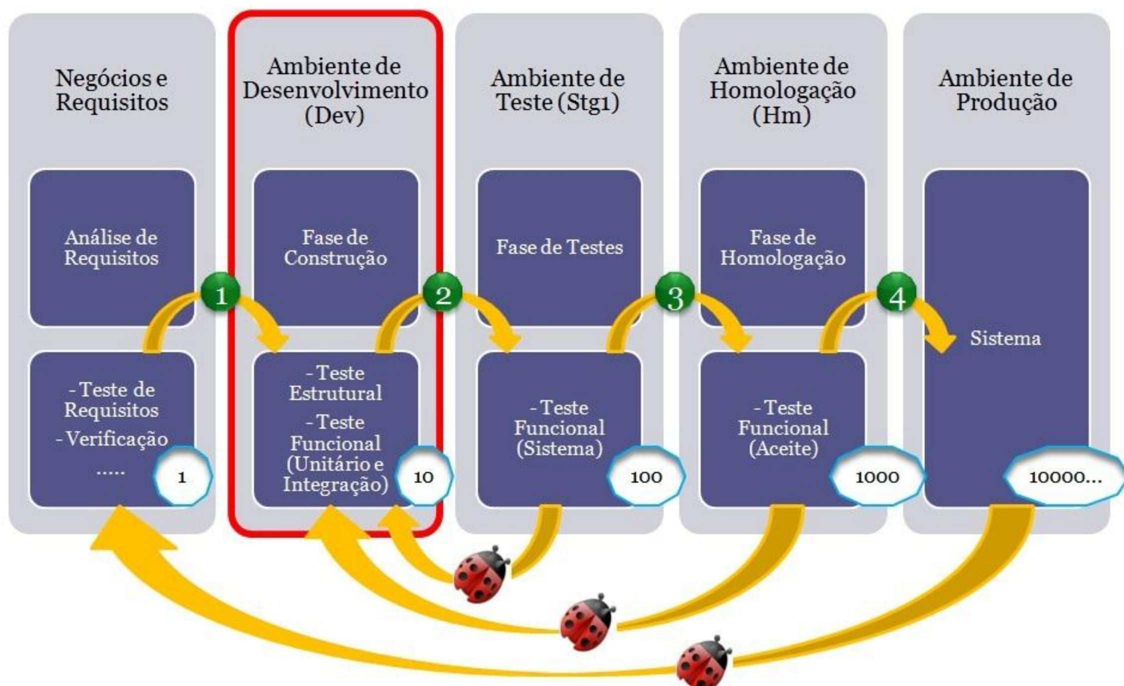
Peculiaridades e detalhamento sobre o processo de teste estenderia o escopo deste trabalho, fugindo do seu foco principal, que é atingir o processo de criação de testes funcionais para teste unitário.

Considerando todas as informações levantadas até este ponto, conclui-se que a melhor fase para encontrar defeitos no *software* é a fase de codificação, que possibilita o teste unitário. O teste unitário quando aplicado para encontrar a maior quantidade de defeitos possíveis, aperfeiçoa todo o processo subsequente de teste e evita constantes fluxos entre equipe de desenvolvimento e teste.

Estrategicamente, o objetivo deste trabalho atinge a fase de teste unitário, ao aperfeiçoar esta atividade estaremos otimizando todas as atividades subsequentes, viabilizando esta prática no desenvolvimento de *software*, evitando retrabalho, grandes fluxos entre equipes, e aproximando equipe de desenvolvimento e testes, fazendo com que ambos, desde o princípio do projeto trabalhem juntos para produção de um *software* de qualidade.

A fim de exemplificar um processo básico de desenvolvimento de *software* e destacar o objetivo deste trabalho neste processo, pode-se examinar na Figura 3 (abaixo) que existem fluxos entre as fases, atividades e ambientes. Observa-se que entre os ambientes e equipes há um fluxo natural do desenvolvimento do *software* e este fluxo pode repetir à medida que um defeito é encontrado.

Figura 3 – Exemplo de um processo básico de desenvolvimento de *software* e o objetivo do trabalho



Fonte: Autoria própria

Considera-se o fluxo:

1. Envia-se os artefatos da análise de requisitos para fase de construção do *software*;

2. ao receber a documentação de requisitos inicia-se a fase de construção, em seguida os artefatos produzidos pelo desenvolvimento é enviado para fase de testes;

3. inicia-se a fase de testes e investigação no intuito de identificar defeitos nos artefatos produzidos pelo desenvolvimento;

4. os artefatos testados são enviados para homologação com o cliente;

5. homologados, os artefatos são enviados ao ambiente de produção.

Observa-se que quanto mais tarde o defeito é encontrado maior é o fluxo, por exemplo, ao identificar um *bug* na fase de testes, o mesmo é reportado para a equipe de desenvolvimento, após a resolução o teste será reexecutado, ao verificar que o problema foi solucionado e que não foram encontrados mais defeitos, pode-se seguir para as demais fases, do contrário o fluxo será repetido até que se obtenha o resultado esperado do artefato de desenvolvimento, neste exemplo, existe o custo de retrabalho e tempo, no caso de encontrar um *bug* na fase de homologação, o mesmo é reportado ao desenvolvedor, que após a identificação e correção irá enviar para equipe de testes validar, após validado, será enviado novamente para homologação, observa-se neste caso que o fluxo, tempo e retrabalho é maior.

Se o *software* está em ambiente de produção e neste ambiente for encontrado um defeito, o mesmo será reportado para equipe de análise de negócio que por sua vez constatará o problema e enviará por meio de documentação a equipe de desenvolvimento, para que o problema seja solucionado, a partir daí, segue o fluxo novamente de desenvolvimento, testes e homologação para enfim a solução ser disponibilizada em produção.

Neste contexto, pode-se examinar no exemplo de um processo básico de desenvolvimento citado, que grandes fluxos possibilitam retrabalho, ferem a confiança entre equipes e até com o cliente, justificando a avaliação de grandes autores mencionados neste trabalho da área de engenharia e teste de *software*, em que se afirma: quanto mais tarde o defeito é encontrado maior o custo para sua correção.

Pode-se constatar também que ao executar atividades que identificam erros na fase de análise de requisitos e de teste unitário, não há fluxos, pois no caso da análise de requisitos, o defeito pode ser identificado e solucionado em uma reunião com os *stakeholders* por exemplo, já o teste unitário, por ser executado pelo próprio desenvolvedor em seu ambiente, o mesmo possui *feedback* em tempo real, com possibilidade de depurar e corrigir erros assim que encontrados no código.

Na próxima seção, é apresentada uma visão geral sobre teste unitário.

1.3 Testes Unitários

Uma unidade é um método ou função testável de uma aplicação que exerce uma pequena área específica da funcionalidade do código a ser testado (HUNT; THOMAS, 2003).

Um teste de unidade é um código executado de forma automatizada, que através de passagem de mensagem ao componente que está sendo testado, verifica suposições sobre seu comportamento lógico. O código para teste de unidade é escrito utilizando a mesma linguagem de programação do componente a ser testado, pode ser escrito através de uma estrutura determinada por um *framework* para testes de unidade, é facilmente escrito, rapidamente executado, totalmente automatizado, confiável, legível e de fácil manutenção (OSHEROVE, 2009).

Os testes de unidade são realizados para validar e provar se a unidade faz o que o desenvolvedor espera, isolando-a do restante do código, permitindo determinar seu comportamento individual.

Sua prática é extremamente útil na programação ou codificação de um *software*, é uma das melhores maneiras que um desenvolvedor pode adotar para melhorar a qualidade do seu código, aperfeiçoa depuração de código (processo de identificação e correção de um erro descoberto), proporciona compreensão mais aprofundada dos requisitos funcionais de uma classe ou método, em paralelo, aumenta a confiabilidade para futuras alterações (HUNT; THOMAS, 2003).

O teste unitário é uma técnica essencial para o aperfeiçoamento na codificação de um sistema, pois proporciona em tempo real, *feedback* no momento em que o código é escrito, além disso, garante cobertura completa e máxima detecção de erros ao utilizar técnicas para teste de unidade (HUNT; THOMAS, 2003).

O desenvolvedor é o responsável pela criação e execução de testes unitários, garantindo a exatidão e eficiência do código através de simulações com diversas condições, entradas e saídas esperadas.

Segundo Hunt e Thomas (2003), o teste unitário é feito por programadores para programadores, com intuito de facilitar as atividades do desenvolvimento de um sistema, este é o principal benefício, possibilitando projetos com mais qualidade e redução drástica da quantidade de tempo gasta na depuração de código.

Usualmente, o teste unitário é executado na fase de codificação do *software*, sua aplicação, possibilita maior produtividade nas fases seguintes do processo de teste e validação

como: teste de integração e teste de sistema, dentre outros. Segundo, Myers (2004) “os testes unitários podem remover entre 30% e 50% dos defeitos dos programas”.

Além disso, o teste unitário possibilita a correção de erros na fase onde o custo da correção é menor (BASTOS et al., 2007).

Os casos de teste para teste de unidade, são projetados a partir da especificação do módulo a ser testado e o código do módulo (unidade). A especificação tipicamente define os parâmetros de entrada e de saída considerando sua função (MYERS, 2004).

Segundo Pressman (2001), os casos de teste para teste de unidade devem ser projetados para descobrir erros provenientes de cálculos errados, comparações incorretas ou fluxo de controle inadequado.

Como mencionado anteriormente, os testes unitários são automatizados, ou seja, são métodos criados e executados para testar outros métodos, avaliando os parâmetros de entrada e de retorno. É possível criar testes unitários automatizados sem a utilização de *frameworks* conforme exemplifica Roy Osherove (2009) em seu livro *The art of Unit Testing*, porém atualmente, para a maioria das linguagens de programação existem *frameworks* que facilitam a criação e execução de testes unitários, viabilizando ainda mais sua aplicação no desenvolvimento de *software*.

Os *frameworks* para teste unitário possibilitam facilidade e agilidade na criação de testes unitários através de conjuntos de APIs, execução automatizada dos testes e revisão de resultados. Os testes são escritos utilizando bibliotecas do *framework,* que por sua vez, são executados a partir de uma ferramenta de teste de unidade do próprio *framework,* e os resultados são exibidos e revistos pelo desenvolvedor (OSHEROVE, 2009).

Mais detalhes sobre automação de testes e *frameworks* para teste automatizado serão tratados nos capítulos 2 e 3.

1.4 Desenvolvimento Dirigido por Testes (TDD)

O objetivo desta seção é mencionar rapidamente o desenvolvimento dirigido por testes, abordagem que também se beneficia com as ferramentas que são produzidas para testes de unidade.

O TDD (*test driven development*), ou desenvolvimento dirigido por testes, é uma técnica que cria uma discussão sobre a questão: quando os testes unitários devem ser escritos? Antes ou depois da codificação do *software*?

Diversos desenvolvedores acreditam que o melhor momento para criação de teste de unidade é depois que o *software* foi escrito, mas atualmente, um número crescente de desenvolvedores prefere criar testes de unidade antes da codificação do *software*. A partir daí, surgiu o conceito do TDD (OSHEROVE, 2009).

De modo geral, o TDD consiste na criação de códigos para teste visando à validação do método a ser criado para determinada funcionalidade, em seguida avança para escrita do código correspondente, ao verificar os resultados do teste, o código escrito para funcionalidade é refatorado até que se obtenham os padrões desejados (OSHEROVE, 2009).

Neste estilo de desenvolvimento, os testes que determinam o código a ser escrito, onde nenhum código entra em produção a menos que esteja associado a testes, pois primeiramente se escreve os testes, ou seja, escreve o código que irá testar a funcionalidade a ser implementada antes de implementar a funcionalidade em si (ASTELS, 2003).

Dessa forma, o TDD garante a testatibilidade do *software* ou código, pois mantém código e testes em sincronia e ajuda na cobertura de teste exaustivo, ou seja, se um erro é introduzido durante a depuração, o teste irá encontrá-lo imediatamente e apontará sua localização sem atrasos entre a descoberta de um *bug* e sua reparação (ASTELS, 2003).

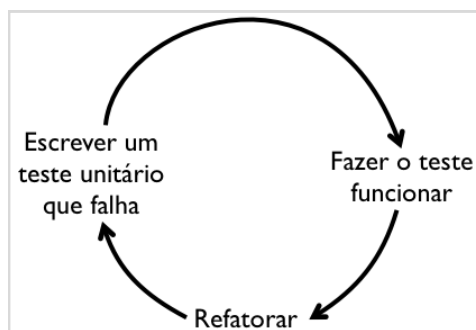
Os próprios testes ajudam a descrever o sistema e embora todos os participantes do projeto se beneficiem com a aplicação de TDD, o mais imediato e principal benefício é exclusivamente do programador, pois lhe dá frequentes *feedbacks* positivos, e possibilita a percepção de progresso na codificação do *software* (ASTELS, 2003).

Esta abordagem proporciona um processo disciplinado que, incentiva o bom desenho, ajuda a evitar erros de programação, ajuda de forma produtiva construir *software* de baixo acoplamento, altamente coesos, taxas de defeitos baixas e reduz o custo de manutenção do *software* (BECK, 2002).

O ciclo fundamental do desenvolvimento de *software* com TDD consiste em: a) escrever um teste unitário que falhe (certamente falhará, pois o código a ser testado ainda não foi criado), b) escrever o código, o mínimo necessário para fazer o teste funcionar, e por fim, c) refatorar o código tornando-o mais simples possível para implementação das características testadas (FREEMAN; PRYCE, 2010).

É possível analisar o ciclo fundamental do TDD através da Figura 4 (próxima página):

Figura 4 – Ciclo Fundamental do TDD



Fonte: Adaptado do livro *Growing object-oriented software, guided by tests* de Steve Freeman e Nat Pryce – Pág. 6

Ainda segundo Freeman e Pryce (2010), à medida que é desenvolvido o *software*, o TDD é utilizado para obter *feedback* sobre a qualidade da implementação, resolvendo os questionamentos sobre seu funcionamento e estrutura, isso é possível pois a refatoração possibilita otimizar o código de modo que não altere seu comportamento, e que represente melhor as características que implementa, tornando-o mais sustentável.

Observa-se, contudo, que existe uma pequena diferença entre teste unitário e TDD, em que teste unitário é um estágio da atividade de teste que exerce a prática de testar a unidade verificando se ele se comporta da maneira como esperado, já o TDD é a disciplina do uso de teste unitário para projetar a codificação do *software*.

Dessa forma, conclui-se que *frameworks* para teste unitário, beneficiam e viabilizam a aplicação do TDD no desenvolvimento de *software*, inserindo automaticamente este tema no escopo deste trabalho.

1.5 Técnicas de Teste

Como já mencionado anteriormente, um bom caso de teste é aquele que possibilita maior probabilidade de encontrar defeitos no *software*. Neste contexto o caso de teste deve ser projetado para este fim, mecanismos para criação de casos de teste devem ser utilizados para garantir a integridade dos testes e fornecer a maior probabilidade de encontrar erros no *software* (PRESSMAN, 2001, p.443-444).

Visto que o *software* é um produto da engenharia, existem duas perspectivas que podem ser usadas como estratégias para projetar os casos de teste: o teste estrutural também

conhecido como teste caixa branca ou *white box* e o teste funcional conhecido também como teste caixa preta ou *black box* (BURNSTEIN, 2003, p.63-65).

Classificar o projeto de caso de teste é uma forma de derivar e selecionar condições e critérios de teste para cada perspectiva.

Os testes estruturais devem garantir que os *softwares* sejam estruturalmente sólidos considerando o aspecto técnico, implica na robustez e funcionamento da estrutura, em que é considerada toda a sua extensão (BASTOS et al., 2007, p.48).

Os testes funcionais devem garantir os requisitos especificados para o *software*, considerando o comportamento das funções do *software* ao estabelecer entradas e garantir que as saídas são produzidas corretamente. Neste contexto, se examina o aspecto fundamental do sistema, sem considerar a estrutura lógica interna do *software* (PRESSMAN, 2001, p.443-444).

Ambas as abordagens se complementam e provavelmente descobrem uma classe diferente de erros. Os testes funcionais podem descobrir erros nas seguintes categorias: funções incorretas ou ausentes, erros de interface, erros em estrutura de dados ou acesso à base de dados externos, erros de comportamento ou desempenho, erros de inicialização ou de encerramento entre outros. Já o teste estrutural pode descobrir erros de design, sintaxe, caminhos lógicos, conjuntos de condições ou laços e estrutura de controle do projeto procedimental (PRESSMAN, 2001, p.459-460).

Dentre as técnicas tradicionais de teste aqui apresentadas, enfatiza-se neste trabalho a técnica funcional, pois essa técnica permite o analisar o comportamento de uma função do sistema através de dados de entrada e saída que são fornecidos durante a execução do teste.

Na seção a seguir, o teste funcional é tratado com mais detalhes.

1.5.1 Teste Funcional (Caixa Preta)

O teste funcional ou teste caixa preta possui foco nos requisitos funcionais do programa ou componente, possibilitando derivar conjuntos de condições de entrada que irá exercer plenamente todos os requisitos funcionais do programa (PRESSMAN, 2001).

Nesta abordagem, não é considerado o comportamento interno e estrutura do programa, em vez disso, concentra-se em encontrar circunstâncias em que o programa ou componente não se comporta de acordo com as suas especificações (MYERS, 2004).

A utilização desta técnica consiste em fornecer entradas e avaliar as saídas geradas a fim de verificar se estão em conformidade com os objetivos especificados, onde o *software* é avaliado segundo o ponto de vista do usuário (FABBRI; VINCENZI; MALDONADO, 2007, cap.2, p.9-10).

O sistema é uma “caixa preta”, pois seu comportamento é avaliado estudando suas entradas e saídas relacionadas, a preocupação neste caso, é somente com a funcionalidade e não com a implementação do *software* ou componente. Aplicável a sistemas que são organizados como funções ou como objetos, utilizando esta técnica, o teste detecta com sucesso um problema no *software* ou componente, quando é apresentadas entradas e se examina que as saídas correspondentes não são as previstas (SOMMERVILLE, 2003).

Em princípio, a técnica de teste funcional, submete o programa a todas as entradas possíveis, no entanto, o domínio de entradas pode ser infinito ou muito grande tornando a prática inviável. Com isso, se faz necessário a definição de técnicas e critérios de teste (FABBRI; VINCENZI; MALDONADO, 2007, cap.2, p.9).

Existem diversos critérios da técnica de teste funcional, os mais conhecidos são: partição de equivalência, análise do valor limite e grafo causa e efeito, estes critérios são brevemente tratados com mais detalhes abaixo:

1. Partição de equivalência: com o objetivo de determinar subconjuntos tornando a quantidade de dados de entrada de um programa finito e mais viável para atividade de teste, o particionamento de equivalência divide o domínio de entradas do *software* em classes ou partições de equivalência, onde de acordo com a especificação do programa, possuem comportamento similar, podendo ser tratados da mesma forma (FABBRI; VINCENZI; MALDONADO, 2007, cap.2, p.11-12).

As partições podem ser identificadas por dados válidos ou inválidos, ou para valores de saídas entre outros. Os testes utilizando esta técnica são elaborados para cobrir as partições, e são aplicáveis a todos os níveis de testes (ISTQB, 2011).

2. Análise do valor limite: este critério é utilizado em conjunto com a partição de equivalência, onde são considerados os limites da partição. Os valores limites de uma partição são seu máximo e seu mínimo, e podem ser valores válidos e inválidos, onde o limite para uma partição válida é um valor limite válido e o limite para uma partição inválida é um valor limite inválido. Ao se projetar casos de teste, é escolhido um valor em cada limite da partição. Este critério pode ser aplicado em todos os níveis de teste, sua capacidade de encontrar defeitos é alta, e valores limite podem ser usados para selecionar dados de teste (ISTQB, 2011).

3. Grafo causa-efeito e tabela de decisão: este critério explora combinações dos dados de entrada e pode derivar um conjunto eficaz de casos de teste que revelem inconsistências em uma especificação. Inicialmente, a especificação deve ser transformada em um grafo que se assemelha com um circuito digital, e através da lógica booleana as entradas são combinadas e entendidas como verdadeiras ou falsas (BURNSTEIN, 2003).

O grafo deve ser convertido em uma tabela de decisões para que o testador utilize-a para criação dos casos de teste, em que deve conter as condições que disparam ações, ou seja, combinações (verdadeiras e falsas) para as condições de entrada e ações resultantes para cada combinação de condições. O grande ganho neste critério é a combinação de condições que geralmente não foram exercitadas durante a utilização dos critérios anteriormente mencionados. Aplicável em todos os níveis de teste, principalmente quando a execução do *software* ou componente depende de muitas decisões lógicas (ISTQB, 2011).

CAPÍTULO 2 – AUTOMAÇÃO DE TESTES

O objetivo deste capítulo é mostrar, através de referências bibliográficas, a definição e objetivo da área de automação de teste de *software*, metodologias e conceitos que foram utilizados para o desenvolvimento do *framework* proposto.

2.1 Conceito

Os testes manuais muitas vezes ficam repetitivos, tornando essa atividade entediante, suscetível a cometer erros e a ignorar erros mesmo que simples, além disso, etapas e testes são ignorados quando a equipe enfrenta um curto prazo, esta situação ocorre porque muitas vezes o teste manual é lento. Ao automatizar os testes elimina-se a possibilidade de erros, pois cada teste é executado exatamente da mesma maneira o tempo todo e em poucos minutos dependendo da ferramenta de automação (CRISPIN; GREGORY, 2009, p.259-260).

De acordo com Bernardo e Kon (2008, p.54), em artigo publicado na revista Engenharia de *Software* Magazine, temos a definição sobre testes automatizados:

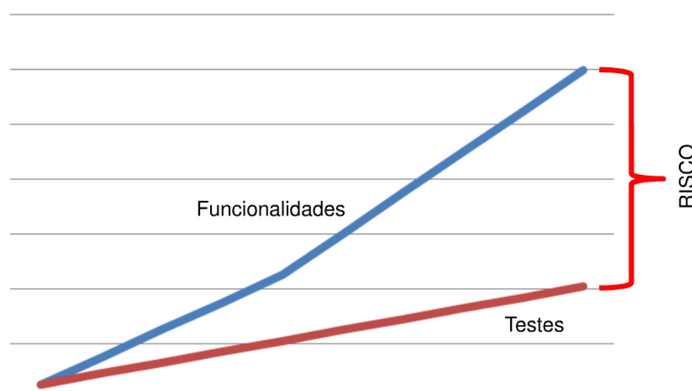
> Testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos resultados obtidos, a principal vantagem dessa abordagem é que todos os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço.

Dessa forma, automatizar testes pode reduzir significativamente o esforço necessário para um teste adequado, ou aumentar significativamente a quantidade de testes que podem ser realizados em tempo limitado. Os testes automatizados podem ser executados repetidas vezes e em minutos ou até segundos, o que poderia levar até horas para ser executado manualmente (FEWSTER; GRAHAM, 1999, p.3-4).

A automação de testes beneficia principalmente na fase de testes de regressão, ou seja, casos de teste que devem ser executados em quase todas as versões do *software*. De acordo com o consultor de teste de *software* Cristiano Caetano (2010) em uma de suas apresentações publicadas em sua página na *web*, os casos de teste para testes de regressão são acumulativos e aumentam à medida que são adicionadas novas funcionalidades no *software*, criando um gargalo para equipe de testes mesmo quando há maturidade no processo de teste, fazendo com que os testes não cubram todas as funcionalidades, possibilitando risco de ocorrência de erros em produção.

É possível examinar na Figura 5 o risco de ocorrência de erros quando os testes não cobrem todas as funcionalidades do *software*:

Figura 5 – Funcionalidades do *Software* x Testes de Regressão



Fonte: http://www.slideshare.net/cristianocaetano/automao-de-testes-mitos-e-verdades-qualister

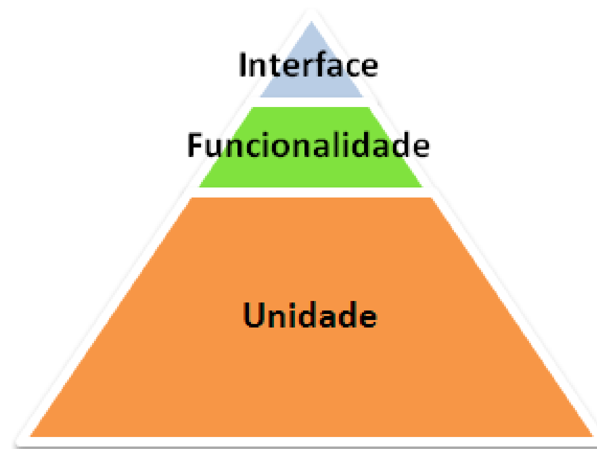
Contudo, considera-se que os casos de teste para teste de regressão depois de um período se tornam uma pilha crescente para serem executados em cada versão e em um tempo limitado. Ao automatizar testes de regressão, o esforço manual para estes testes é eliminado, evitando também, erro humano na execução repetida de testes tornando essa atividade monótona (AHMED, 2010, p.107-108).

Além dos testes de regressão a automação de teste beneficia diversos tipos e técnicas de teste como: teste de unidade, teste de integração, análise estática de código, teste de desempenho entre outros, logo, a automação de teste deve ocorrer somente quando há uma oportunidade ou necessidade, que envolve redução do esforço de teste, redução do período de execução, cobertura adicional, entre outros. Os benefícios da automação devem envolver estes fatores, o que não seria possível com o teste manual, para que se obtenha de fato retorno do investimento na automação de testes (BLACK; MITCHELL, 2011, p.461-462).

Neste contexto, os autores Tim Riley e Adam Goucher (2010, p.103-104) exemplificam em seu livro *Beautiful Testing*, que tentativas de automação falham porque testadores gastam muito tempo automatizando ou na tentativa de automatizar comportamentos ou cenários que não compensam, o mesmo ocorre quando não são automatizados principais alvos de automação, decidir quais os testes automatizar é difícil e cabe a equipe de teste e as partes interessadas determinarem o escopo da automatização. Ainda segundo o autor, a automação é uma solução perfeita e mutuamente aceitável para desenvolvedores escreverem teste unitário.

Segundo exemplifica Mike Cohn (2010, p.311-312), uma das razões pelo qual as equipes tiveram dificuldade em automatizar testes: automatização no nível errado. De acordo com o autor, uma estratégia eficaz de automação de teste é automatizar os testes em três níveis: unidade, funcionalidade ou serviço e interface. Através da Figura 6 pode-se analisar que estes níveis formam a pirâmide de automação:

Figura 6 – Pirâmide da Automação de Testes



Fonte: Adaptação do livro - *Succeeding with Agile* de Mike Cohn

Observa-se que o nível unidade está em maior proporção, pois segundo o autor, o teste unitário deve ser a base sólida da estratégia de automação de teste, isso porque o teste de unidade proporciona muitas vantagens em relação aos demais tipos de testes e possibilita confiança às fases seguintes.

O nível mais baixo da pirâmide é o alicerce que suporta todo o restante, dessa forma, testes unitários e de componentes representam a maior parte dos testes automatizados. No desenvolvimento ágil a tendência é concentrar os testes automatizados principalmente nesta camada, por ser o meio mais rápido e dispendioso para escrita dos testes automatizados, fornecem o mais rápido *feedback*, além disso, possibilitam maior retorno do investimento em relação aos demais tipos de teste (CRISPIN; GREGORY, 2009, p.276-277).

A camada intermediária inclui automação de testes para testes funcionais que verificam conjuntos maiores de funcionalidade do que a camada de unidade. Estes testes focam nos serviços ou funcionalidades por traz da interface, testando-as diretamente sem passar pela interface. Já a camada superior representa o que deve ser o menor esforço de automação, pois os testes geralmente proporcionam menor retorno do investimento. Isso porque estes testes são escritos utilizando elementos de interface para manipulação da

interface, que tendem a ser alterados muitas vezes, fazendo com o que o teste "quebre" e tenha que ser ajustado ou até refeito. Por esse motivo, nesta camada, os testes geralmente são automatizados para teste de regressão, em que a interface raramente sofre mudanças (CRISPIN; GREGORY, 2009, p.277).

Com a ascensão de metodologias ágeis e diversas vantagens no uso da automação de testes principalmente nessa abordagem, a indústria de *software* tem visto grandes obras, diversos conceitos e metodologias neste segmento para testes de *software*.

A seguir, serão apresentadas as principais abordagens para automação de testes, a fim de introduzir a abordagem utilizada neste trabalho.

2.2 Abordagens para Automação de Testes Funcionais

Existem diferentes abordagens para automação de teste, cada qual com suas características específicas, a seguir, serão apresentadas brevemente as principais abordagens.

A abordagem *Record and Playback* consiste na utilização de uma ferramenta de automação de testes para gravar ações executadas por um usuário sobre a interface gráfica da aplicação a ser testada. As ações executadas são convertidas em *scripts* pela ferramenta, para que as ações gravadas sejam repetidas como na primeira execução. Para cada caso de teste é criado um *script* de teste que inclui os dados de entrada e resultados esperados. Seu uso é simples e prático, apropriado para testes executados poucas vezes, porém, não é viável para um grande conjunto de casos de teste devido à dificuldade de manutenção, baixa taxa de reutilização, não possui flexibilidade a mudanças e o tempo de vida é curto (FANTINATO, 2004).

Programação de scripts, é uma abordagem que estende a *record & playback*, pois trata-se da alteração dos scripts que são gerados por esta ferramenta, afim de se obter um comportamento diferente do script original da primeira execução. Para isso, é necessário que a ferramenta de gravação de scripts possibilite a edição dos mesmos. Com isso, os *scripts* contemplam maior quantidade de verificações, possibilitando maior taxa de reutilização, maior tempo de vida e mais robustez dos *scripts*. A principal desvantagem dessa abordagem, é que para cada caso de teste dever programado um *script* de teste, incluindo os dados e procedimentos de teste, produzindo grande número de *scripts* (FANTINATO, 2004).

Visando a diminuição da quantidade de *scripts* de teste e melhor definição e manutenção dos casos de teste automatizados, a abordagem *data-driven testing* (teste

orientado a dados), consiste em manter os dados de teste armazenados em arquivos, mantendo-os separados dos *scripts*. Os *scripts* devem contemplar apenas os procedimentos de teste, lógica de execução e ações sobre a aplicação que normalmente são genéricos para um conjunto de casos de teste. Dessa forma, os dados de teste são acessados pelo *script* quando necessário e de acordo com o procedimento de teste implementado, possibilitando uma importante vantagem, adicionar, modificar ou remover dados e casos de teste sem ou com pequena manutenção dos *scripts* de teste (FANTINATO, 2004).

Diferente da abordagem *data-driven testing*, a abordagem *keyword-driven testing* (teste orientado a palavras chave), consiste em extrair dos *scripts* de teste o procedimento de teste que representa a lógica de execução. Dessa forma, os procedimentos de teste são armazenados em uma fonte externa, na forma de um conjunto ordenado de palavras-chave e respectivos parâmetros. Os *scripts* de teste nesta abordagem passam a conter apenas ações específicas de teste sobre a aplicação, as quais são identificadas pelas palavras-chave da fonte externa (arquivo). Estas ações de teste representam funções de um programa, podendo receber parâmetros, que são ativados pelas palavras-chave a partir da execução de diferentes casos de teste. Assim, o *script* de teste não mantém os procedimentos de teste no código, obtendo-os diretamente dos arquivos de procedimento de teste. A principal vantagem é a facilidade de adicionar, modificar e remover passos da execução no arquivo de procedimento de teste com necessidade mínima de manutenção dos *scripts* (FANTINATO, 2004).

O escopo e principal alvo deste trabalho é a automatização de testes unitários utilizando a abordagem *data-driven testing*, por isso a próxima seção é dedicada ao detalhamento dessa abordagem.

2.3 Data-Driven Testing

Comumente ocorrem situações em que casos de testes tendem a fazer as mesmas coisas porém usando dados diferentes, esta é uma situação que pode ser conduzida através de testes automatizados orientado a dados (*data-driven testing*) (BLACK; MITCHELL, 2011, p.502).

Data-driven é uma técnica para desenvolvimento de scripts de teste que consiste no armazenamento das entradas e saídas de teste em um arquivo (fonte de dados) externo ao *script* de teste. Quando o teste é executado, o *script* de teste lê o arquivo com os dados do

teste, e utilizando estes dados os procedimentos do teste são executados através do *script* (FEWSTER; GRAHAM, 1999, p.83-84).

Uma ferramenta para testes automatizados baseado em dados de teste pode reduzir a manutenção do *script* de teste e permitir envolvimento com testadores que efetuam testes manuais, auxiliando na especificação de entradas a serem inseridas em uma fonte de dados (planilha ou tabela), para que o *script* percorra cada valor, combinando os resultados esperados com os resultados reais (CRISPIN; GREGORY, 2009, p.182-183).

Abaixo, é apresentado exemplo de um arquivo (planilha) com dados a serem utilizados no teste:

Figura 7 – Fonte de Dados – *Data Driven Testing*

	A	B	C	D
1	Caso de Teste	valor 1	valor 2	resultado esperado
2	CT01 - Somar valores positivos	1	2	3
3	CT02 - Somar com valor negativo	1	-2	-1
4	CT03 - Somar com valores reais	3	0,02	3,02
5	CT04 - Somar com valores negativos	-3	-2	-5

Fonte: Autoria Própria.

É importante observar que com esta técnica, é possível criar diferentes casos de teste utilizando o mesmo *script*, abaixo é apresentado um exemplo de script de teste para a fonte de dados acima utilizando pseudo-linguagem:

Figura 8 – *Script* de Teste em Pseudo-linguagem

```
Abrir planilha dadosCalculadoraTeste.xls;
Ler dados da planilha;
Obter valor1, valor2;
Obter resultado esperado;
Passar valor1, valor2 para o método somar da classe calculadora;
Obter resultado da soma;
Comparar se o resultado retornado é igual ao resultado esperado;
```

Fonte: Autoria Própria

Dessa forma, cada linha da tabela representa um único teste, que parametriza a função a ser testada, em seguida o resultado obtido é comparado com o resultado esperado.

Apesar das diversas vantagens no uso dessa abordagem, para escrita dos *scripts* de teste é necessário conhecimento em programação, é preciso um automatizador para construir a

capacidade de leitura e armazenamento dos dados da fonte externa para parametrização das funções a serem testadas (BLACK; MITCHELL, 2011, p.503).

Neste caso, muitas vezes os testadores poderiam auxiliar somente na elaboração dos casos de teste e especificação das entradas do teste, possibilitando a construção dos dados da planilha e consequentemente proporcionando maior cobertura dos testes.

No contexto deste trabalho, esta não é uma desvantagem, pois na abordagem de teste unitário, o próprio desenvolvedor escreve os testes para validação das unidades do código, seu uso é produtivo tanto para desenvolvedores como testadores.

Durante a criação de testes unitários, geralmente, para um bom conjunto de testes, muitas vezes se reutiliza o mesmo código, alterando-se somente as entradas e resultados esperados, no intuito de validar valores limites, valores válidos e inválidos, de forma que seja possível verificar o comportamento da unidade para os diversos tipos de entrada (AHMED, 2010, p.120).

Dessa forma, ao unir teste unitário com a abordagem de teste orientado a dados, é possível obter economia de códigos que efetuam o teste de unidade, pois as diversas combinações de entradas e saídas são lidas de uma fonte externa ao código para o teste, possibilitando um meio que abrange todos os casos de teste especificados para o mesmo *script* de teste.

Os testadores quando familiarizados com uma fonte de dados para produção dos casos de teste, auxiliam os desenvolvedores no uso de técnicas para testes funcionais, garantindo maior cobertura dos testes unitários, possibilitando encontrar erros precocemente e otimizando as demais atividades de testes subsequentes.

CAPÍTULO 3 – DESENVOLVIMENTO

Este capítulo tem por objetivo apresentar a proposta do *framework*, tecnologias utilizadas, características e conceitos utilizados por meio da revisão bibliográfica apresentada até este ponto.

3.1 Tecnologias Envolvidas

Para o desenvolvimento do *framework*, se fez necessário além do levantamento bibliográfico teórico, o estudo das principais tecnologias que envolvem automatização de testes e teste unitário.

Primeiramente foi escolhida a linguagem de programação *Java* por ser uma linguagem orientada a objetos, possuir código aberto, boa documentação, possuir IDEs (*Integrated Development Environment*) gratuitas entre outras vantagens em sua utilização.

Existem diversos *frameworks* desenvolvidos para teste unitário em Java, os mais conhecidos atualmente são JUnit e TestNG, a seguir uma breve descrição de cada *framework*:

1. JUnit: ferramenta *open-source* amplamente utilizada, tornou-se rapidamente o framework padrão para o desenvolvimento de testes de unidade no Java, desde seu lançamento em 2001.

2. TestNG: ferramenta *open-source* que abrange não só testes unitários, mas também testes de integração, funcionais e aceitação, por isso suporta mais recursos que o JUnit.

Existe uma forte discussão e comparação em diversos fóruns na *web* sobre a escolha do uso de uma das duas ferramentas, ambas possuem fiéis seguidores (desenvolvedores) e possuem boa documentação e suporte que pode ser encontrado nas páginas *www.junit.org* e *http://testng.org*.

Nestes fóruns de discussões podemos facilmente encontrar dentre as diferenças entre JUnit e TestNG a abordagem de testes orientado a dados, o TestNG suporta essa abordagem e possibilita seu uso através de arquivos XML, já o JUnit a partir da versão 4 evoluiu principalmente na abordagem de testes parametrizados, porém sem suporte à fonte de dados externos ao *script,* no JUnit, mesmo com parametrização os dados do teste ainda ficam estáticos no código.

A fim de facilitar a criação de testes unitários orientado a dados no JUnit, através da *Apache Maven Project* foi criado um projeto disponibilizado pelo repositório de códigos fonte

open source, a SourceForge, um projeto nomeado DDTUnit[3], que possibilita a criação de testes unitários orientado a dados através de arquivos XML.

Ambas as ferramentas são semelhantes em sua utilização para criação de testes unitários, o que se pode destacar é que o JUnit é projetado para aprimorar teste de unidade, enquanto a proposta do TestNG abrange também o alto nível de testes, possibilitando flexibilidade em conjuntos de teste de grande porte.

Para este trabalho, foi escolhido o JUnit por ser o framework mais utilizado por desenvolvedores para teste unitário e possuir menor curva de aprendizagem para sua utilização.

O framework proposto neste trabalho tem a finalidade de possibilitar uma alternativa para utilização de fonte de dados em arquivos XML, optou-se em utilizar planilhas do Excel para consulta dos dados, unindo a popularidade e facilidade de uso do Excel com a simplicidade do JUnit.

A seguir será apresentada rapidamente o funcionamento do framework JUnit a partir da versão 4, a fim de se obter informações sobre sua API e formas de uso e integração.

3.1.1 O Framework JUnit

É possível encontrar facilmente na *web* documentação sobre a API do JUnit, tutoriais entre outros, nesta seção é apresentada somente as funções do JUnit que motiva o desenvolvimento deste trabalho.

O framework JUnit fornece uma API (*Application Programming Interface*) para criar testes e aplicações para executá-los, este *framework* faz parte de uma família de arquitetura de testes conhecida como xUnit. Este termo xUnit é usado para se referir a família de *frameworks* de automação de testes, onde a maioria das linguagens de programação difundidas hoje possuem uma aplicação xUnit para criação de *scritps* para teste unitário automatizado (MESZAROS, 2007, p.75-76).

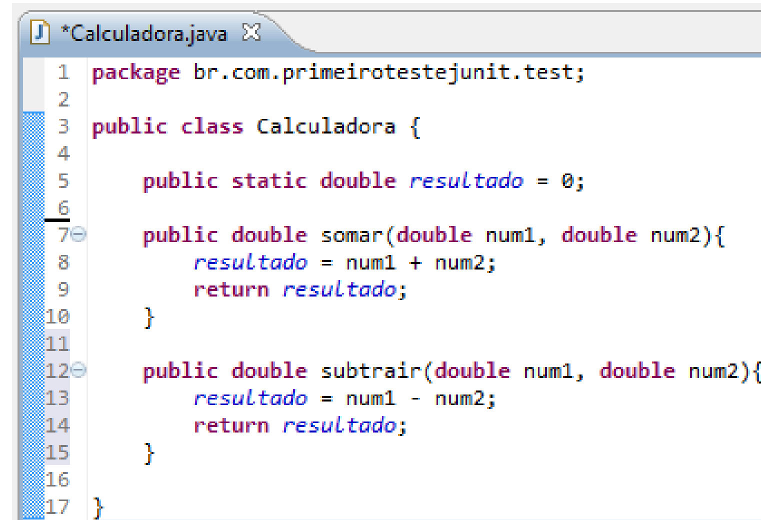
O JUnit já vem instalado nas versões recentes de IDE’s como Eclipse, NetBeans, JBuilder, BlueJ entre outros, mais informações sobre a instalação e configuração do JUnit, pode ser encontrada no site *http://junit.sourceforge.net/doc/cookstour/cookstour.htm*.

---

[3] Projeto DDTUnit – Apache: http://ddtunit.sourceforge.net/

A partir da versão 4.x o JUnit permite a utilização de anotações para a especificação de casos de teste, para exemplificar de uma forma prática, considere a classe Calculadora.class:

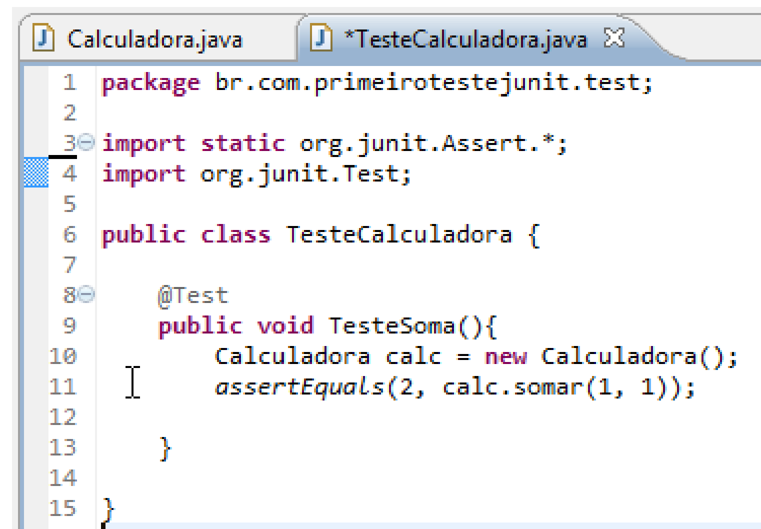
Figura 9 – Exemplo 1: Classe Calculadora.class



Fonte: Autoria própria.

Uma classe de teste para testar o método soma da classe Calculadora.class utilizando o JUnit é exemplificado abaixo:

Figura 10 – Exemplo 1: Classe TesteCalculadora.java

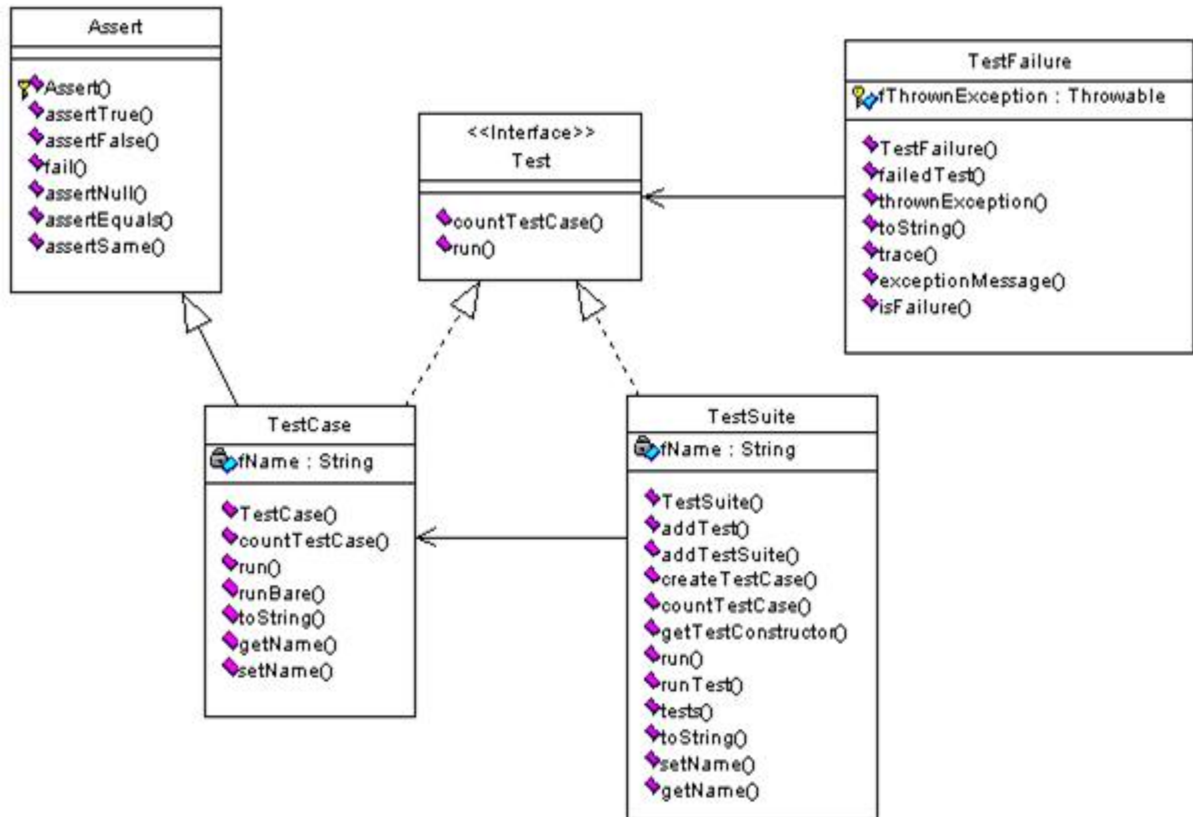


Fonte: Autoria própria.

Através da figura 10 é possível examinar a notação @Test, esta notação é a principal do JUnit, pois ela indica que o método anotado possui testes, é através dessa notação que o JUnit irá executá-los.

A classe assertEquals compõe o conjunto de classes do JUnit para fazer avaliações sobre o resultado esperado e o resultado obtido pelo método, na figura abaixo, pode-se avaliar as principais classes do JUnit:

Figura 11 – Classes do JUnit



Fonte: http://siep.ifpe.edu.br/anderson/blog/?page_id=976

O assertEquals utilizado na classe TesteCalculadora.class é uma das diversas assertivas disponibilizadas pelo JUnit, o assertEquals por exemplo compara dois valores de igualdade, o teste passa se os valores são iguais. No exemplo mencionado, é comparado o valor ‘2’ ao resultado que deve ser retornado pelo método soma da classe Calculadora.

Ao alterar a assertiva para:

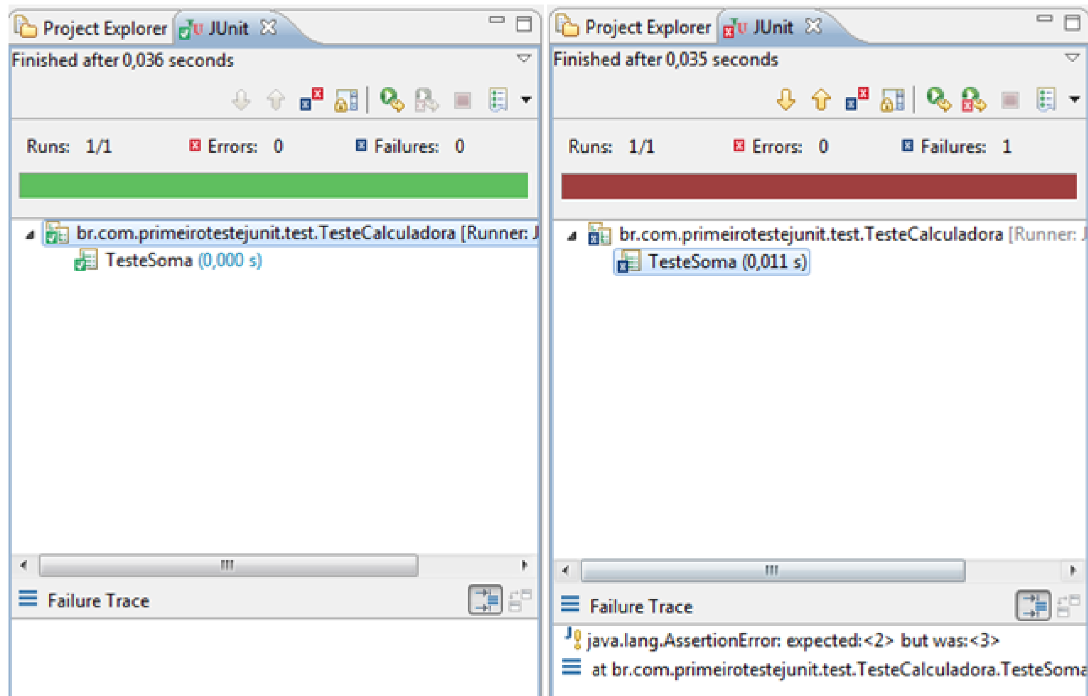
assertEquals(2, calc.somar(1, 2));

o teste vai falhar, pois o resultado esperado é diferente do resultado retornado pelo método testado. O JUnit exibe o resultado do teste e sinaliza a condição que falhou no teste, por padrão ao apresentar os resultados do teste, em modo gráfico o JUnit, o método pode apresentar a cor verde, indicando que o teste passou e a cor vermelha quando o teste falhou.

Para verificar se o teste passou ou falhou, é possível executar o teste através das opções *Run As* e JUnit Test da IDE utilizada, após executar o teste o JUnit exibe o resultado.

Utilizando o exemplo do TesteCalculadora.class, pode-se analisar na figura 12 a exibição do resultado do teste pelo *framework*.

Figura 12 – Apresentação dos resultados do teste pelo JUnit



Fonte: Autoria Própria

É importante observar no exemplo da classe TesteCalculadora.class que os parâmetros de entrada para o teste são estáticos, para variar esses parâmetros é necessário repetir as asserções porém com parâmetros diferentes, pode-se examinar este cenário na figura 13:

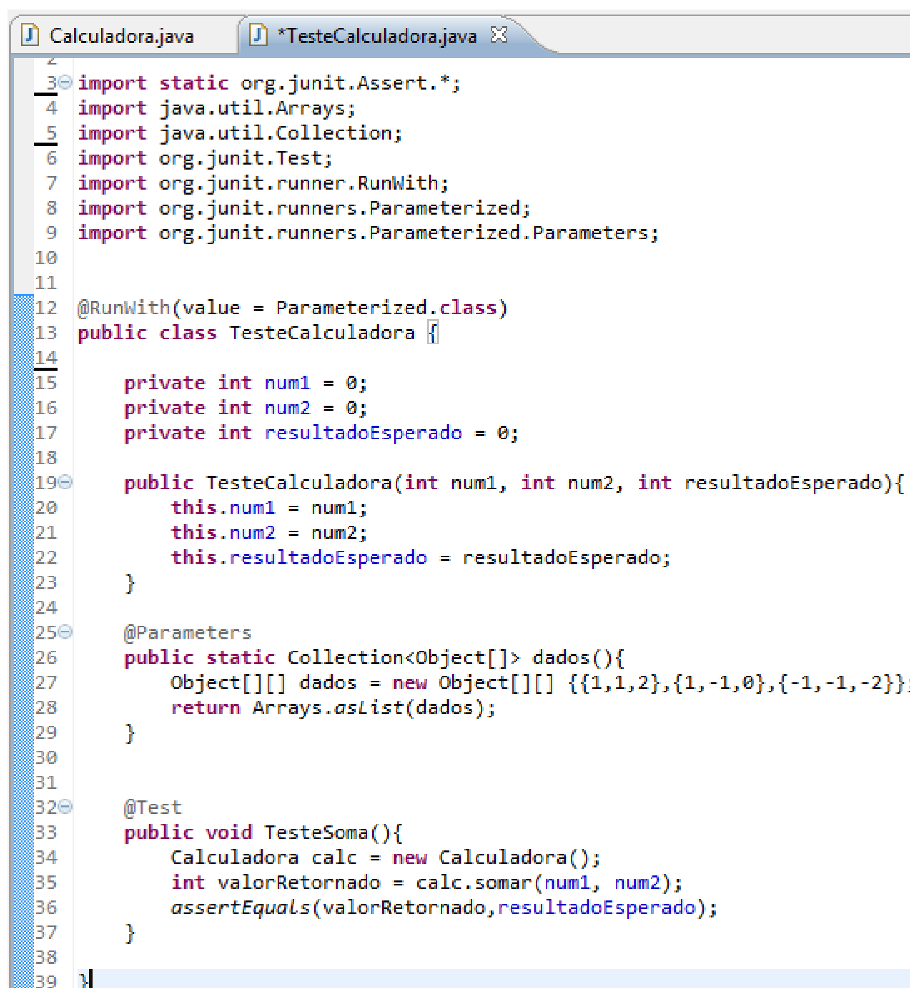
Figura 13 – Exemplo 2: TesteCalculadora

```
Calculadora.java    *TesteCalculadora.java
1  package br.com.primeirotestejunit.test;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5
6  public class TesteCalculadora {
7
8      @Test
9      public void TesteSoma(){
10         Calculadora calc = new Calculadora();
11         assertEquals(2, calc.somar(1, 1));
12         assertEquals(0, calc.somar(1, -1));
13         assertEquals(-2, calc.somar(-1, -1));
14     }
15
16 }
```

Fonte: Autoria própria.

Tendo visto este cenário, o JUnit a partir da versão 4 possibilita os testes parametrizados, que consiste em variar o valor do parâmetro do teste de unidade. As anotações @RunWith e @Parameter são usadas para fornecer valor ao parâmetro do teste, a anotação @Parameter deve retornar uma lista de parâmetros e passá-los como argumento para o construtor da classe de teste, pode-se observar o uso dessas anotações na figura 14:

Figura 14 – Exemplo 2: TesteCalculadora – Teste parametrizado



Fonte: Autoria Própria

Ao examinar o exemplo acima, é possível notar que as assertivas não são mais repetidas no código de teste, porém os dados para o teste (parâmetros) continuam acoplados no código, caso necessite alterar, excluir, ou inserir algum parâmetro é necessário alterar o código de teste para que o teste seja executado com as alterações efetuadas.

Visando este problema, propõe-se o *framework* com arquitetura funcional mencionada na próxima seção.

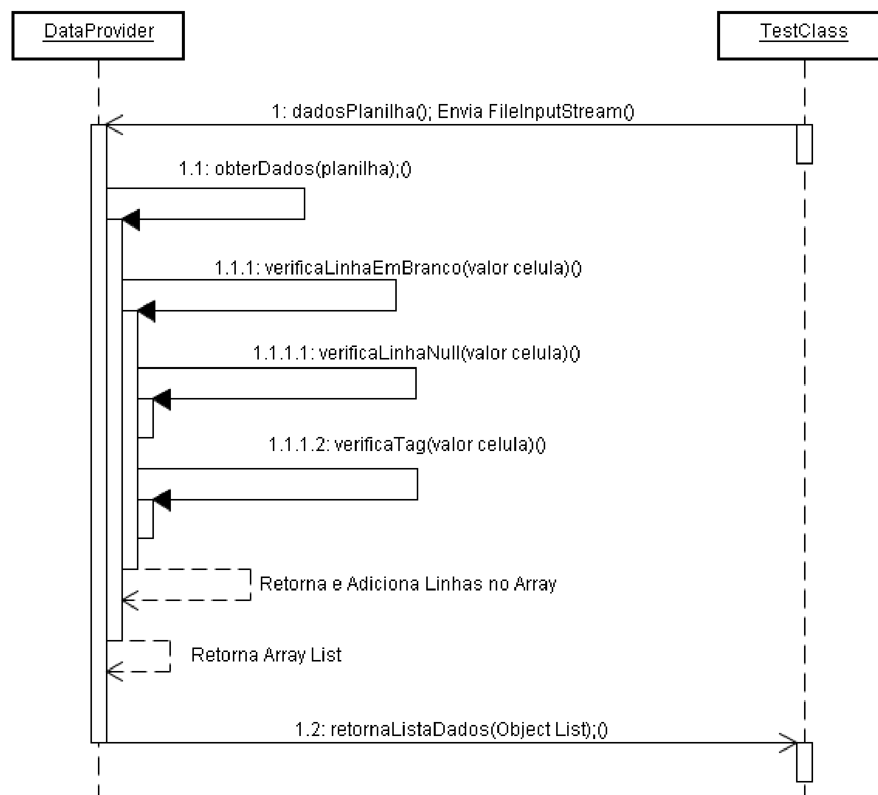
3.2 Arquitetura e Funcionamento do Framework Proposto

Para desacoplar os dados do teste do código de teste, primeiramente é necessário definir qual o tipo e formato de fonte externa ao código que será utilizada para leitura dos dados. Ao definir a fonte de dados, foi considerado principalmente a popularidade e facilidade de uso, para que desenvolvedores e testadores possam utilizá-la para especificação das entradas e saídas do teste, contudo, escolheu-se o Excel como fonte de dados para o teste.

Para possibilitar a leitura de dados em uma planilha e criação do *framework* que terá o papel de provedor de dados (*data provider*), se fez necessário o uso da API Apache Poi, este projeto da Apache tem o objetivo de manter APIs Java para manipular diversos arquivos baseados no *Office* da *Microsoft* incluindo Word, Excel, Power point e Outlook, considerando inclusive suas extensões correspondentes a versões dos aplicativos.

Em uma visão macro da funcionalidade e principais verificações do *framework,* é possível analisar na figura 15 suas principais características:

Figura 15 – Funcionamento do *framework*



Fonte: Autoria Própria

As verificações foram criadas para que se retornem corretamente os dados que devem fazer parte do teste. Por exemplo, através da verificação se a linha é vazio ou está em "branco" é possível evitar que linhas vazias sejam inseridas no teste, considerando somente as linhas que contém informações.

O conteúdo das células também é verificado se possuem a informação *null* (nulo) ou se possuem informações entre *tags*, essa última para que informações que não devem fazer parte do teste (comentários e informações adicionais) não sejam enviados para a classe de teste, a verificação do valor *null* significa que deve ser inserido um valor "vazio" no *array*, a fim de se testar algum parâmetro sem informação, por exemplo, testar um *login* sem a senha requerida. Mais informações sobre a utilização do *framewok* a fim de utilizar todos os seus recursos será tratado no capítulo 4.

Além de prover os parâmetros que de fato se tem intenção de enviar ao teste, o *framework* é composto por verificações de tipo da célula, retornando se os valores que estão inseridos na planilha são numéricos, *strings* ou fórmulas, possibilitando maior flexibilidade em sua utilização.

É importante ressaltar novamente que para que o JUnit interprete que os dados da lista retornados devem ser os parâmetros do teste é imprescindível que no construtor do teste estes parâmetros sejam declarados, para que sejam corretamente utilizados nos métodos de teste, além da utilização das notações @RunWith e @Parameter.

CAPÍTULO 4 – APRESENTAÇÃO

Este capítulo apresenta o framework proposto considerando suas características, exemplos de uso e testes realizados no mesmo.

4.1 Como Utilizar o Framework

O principal objetivo do framework é fornecer informações de uma planilha no Excel para que o JUnit parametrize os métodos de teste baseado nos dados da planilha.

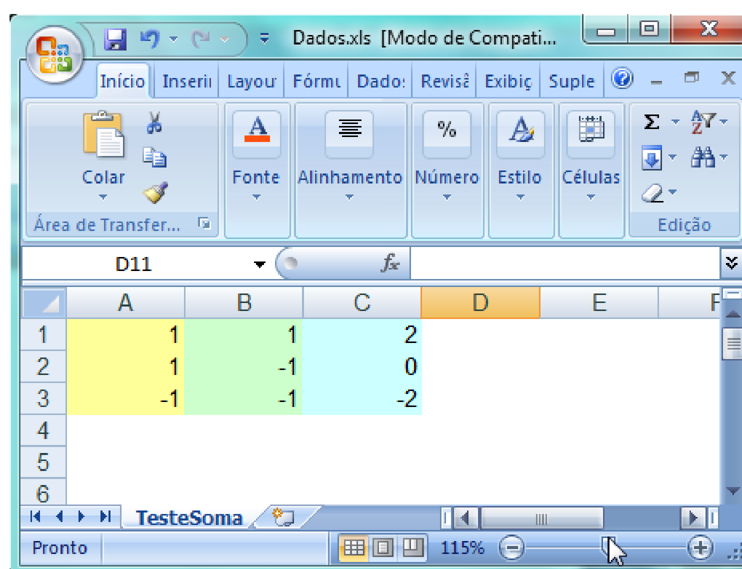
Para utilização do framework além de possuir o JUnit a partir da versão 4, é necessário configurar o arquivo com extensão .jar do *framework* no *build path* do projeto de teste.

Com o objetivo de possibilitar clareza na representação do funcionamento do framework, utilizaremos o mesmo exemplo da calculadora mencionado no capítulo anterior.

Para que o *framework* possa identificar os dados a serem inseridos no teste, é preciso estabelecer um *layout* da fonte de dados *(excel)* que possibilite a correta utilização do *framework* e correta parametrização do teste. As possibilidades básicas de layout para o uso correto do *framework* serão apresentadas a seguir.

1. *Layout* básico: constituído somente com as informações a serem inseridas no teste (figura 16):

Figura 16 – Layout da planilha de dados



Fonte: Autoria Própria

Seguindo e exemplo da Calculadora mencionado no capítulo anterior, pode-se observar na figura 16 que os parâmetros de entrada são os valores das células correspondentes as colunas ‘A’ e ‘B’, e na coluna ‘C’ o valor expresso corresponde ao resultado esperado para o teste.

2. *Layout* com informações adicionais que não devem fazer parte do conjunto de parâmetros a serem enviados ao teste (figura 17):

Figura 17 – Layout com informações adicionais que não contemplam a parametrização

A	B	C	D
<Caso de Teste>	<num 1>	<num 2>	<resultado esperado>
<CT01: Soma valores válidos>	1	1	2
<CT02: Soma com valor negativo>	1	-1	0
<CT03: Soma de valores negativos>	-1	-1	-2

Fonte: Autoria própria.

Neste layout, as informações inseridas entre *tags* serão ignoradas pelo provedor de dados do teste, será enviado para o método de teste somente os parâmetros que não estão entre *tags*.

O uso deste *layout* facilita a identificação dos casos de teste bem como seu principal objetivo, este modelo pode ser utilizado por testadores para criação de diversos casos de teste a fim de garantir maior cobertura dos testes possibilitando maior probabilidade de encontrar erro nas unidades de código.

Como já mencionado no capítulo anterior, o provedor de dados ignora linhas vazias ou em branco, em casos onde o teste contempla parâmetros vazios, a fim de testar parâmetros de entrada obrigatórios entre outros, é necessário utilizar a palavra *null* para o campo em que se deseja testar, veja um exemplo na figura 18:

Figura 18 – Layout para teste de valores nulos

	A	B	C	D
1	<Caso de Teste>	<Usuário>	<Senha>	<Mensagem Esperada>
2	<CT01: Login e Senha Válidos>	andressa.garcia	123456	Login Efetuado com Sucesso
3	<CT02: Login e Senha Inválidos>	fabio.meira	654321	Usuário ou Senha não Cadastrada
4	<CT03: Informa Login e não Informa Senha>	andressa.garcia	NULL	A senha é obrigatória para o login
5	<CT04: Informa Senha e não Informa Login>	NULL	123456	O usuário é obrigatório para o login

Fonte: Autoria própria.

Dessa forma, o provedor de dados interpreta que deve ser enviado um valor vazio como parâmetro para o método de teste.

Utilizando o provedor de dados é possível vincular palavras aos métodos que se deseja testar, definindo quais funções devem ser executadas para os parâmetros determinados na planilha, veja o exemplo na figura 19:

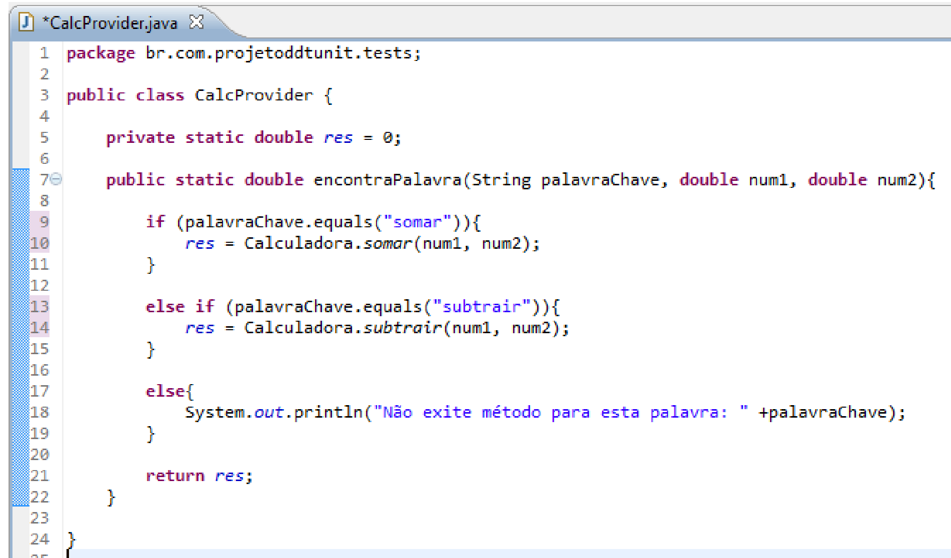
Figura 19 – Vinculando palavras aos métodos a serem testados

	A	B	C	D
1	<Test Case>	<num 1>	<num 2>	<resultado esperado>
2	somar	1	1	2
3	somar	1	-1	0
4	somar	-1	-1	-2
5	subtrair	1	2	-1
6	subtrair	1	1	0

Fonte: Autoria Própria

Para contemplar este caso, deve-se criar uma classe que efetua o vínculo das palavras informadas na planilha com os métodos a serem testados. Veja nas figuras 20 e 21 uma maneira simples de fazer este vínculo e de executar os testes baseando-se nas palavras da planilha:

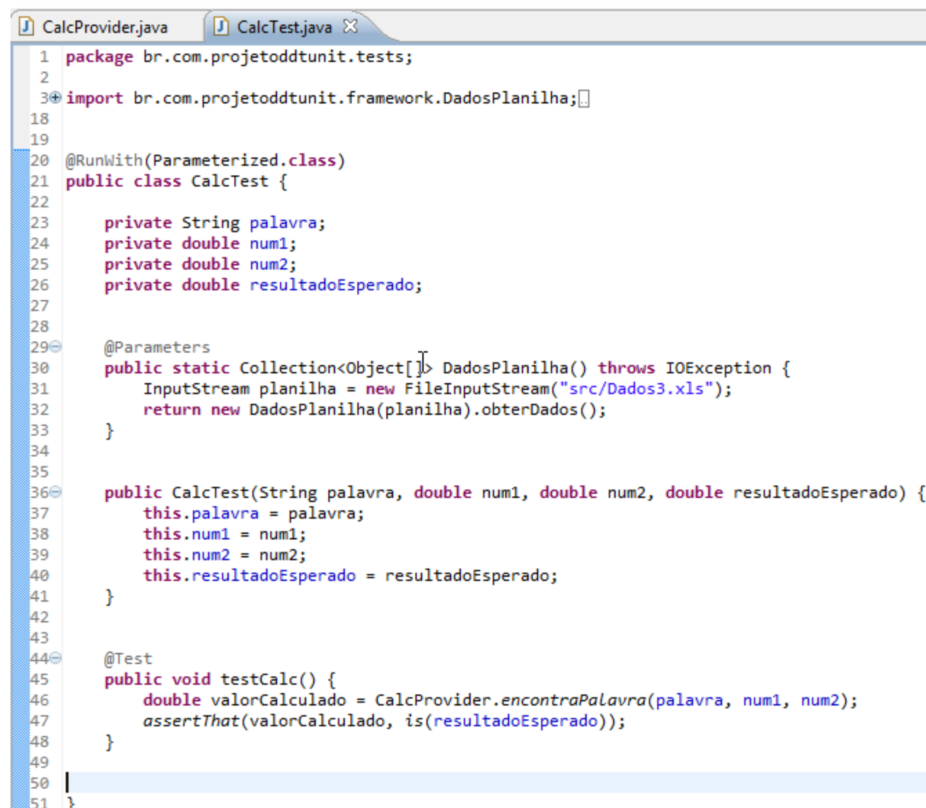
Figura 20 – Vinculando palavras aos métodos



```
*CalcProvider.java
1  package br.com.projetoddtunit.tests;
2
3  public class CalcProvider {
4
5      private static double res = 0;
6
7      public static double encontraPalavra(String palavraChave, double num1, double num2){
8
9          if (palavraChave.equals("somar")){
10             res = Calculadora.somar(num1, num2);
11         }
12
13         else if (palavraChave.equals("subtrair")){
14             res = Calculadora.subtrair(num1, num2);
15         }
16
17         else{
18             System.out.println("Não exite método para esta palavra: " +palavraChave);
19         }
20
21         return res;
22     }
23
24 }
```

Fonte: Autoria Própria

Figura 21 – Classe de teste com execução variada de acordo com a palavra definida na planilha



```
CalcProvider.java    CalcTest.java
1  package br.com.projetoddtunit.tests;
2
3  import br.com.projetoddtunit.framework.DadosPlanilha;
18
19
20 @RunWith(Parameterized.class)
21 public class CalcTest {
22
23     private String palavra;
24     private double num1;
25     private double num2;
26     private double resultadoEsperado;
27
28
29     @Parameters
30     public static Collection<Object[]> DadosPlanilha() throws IOException {
31         InputStream planilha = new FileInputStream("src/Dados3.xls");
32         return new DadosPlanilha(planilha).obterDados();
33     }
34
35
36     public CalcTest(String palavra, double num1, double num2, double resultadoEsperado) {
37         this.palavra = palavra;
38         this.num1 = num1;
39         this.num2 = num2;
40         this.resultadoEsperado = resultadoEsperado;
41     }
42
43
44     @Test
45     public void testCalc() {
46         double valorCalculado = CalcProvider.encontraPalavra(palavra, num1, num2);
47         assertThat(valorCalculado, is(resultadoEsperado));
48     }
49
50
51 }
```

Fonte: Autoria Própria

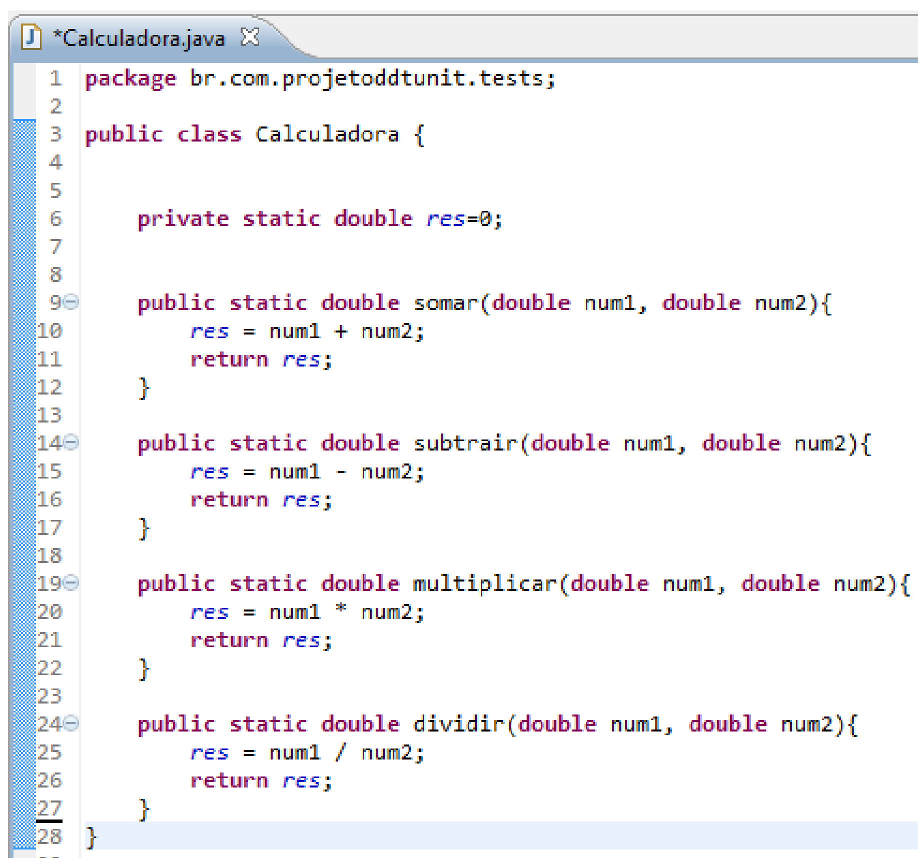
4.2 Testes Realizados e Resultados Obtidos

Para garantir que as funcionalidades propostas pelo *framework* foram atendidas, foram efetuados diversos testes a fim de se obter resultados que devem refletir no atendimento do objetivo dessa pesquisa e promover possíveis melhorias. Abaixo relatório de dois teste efetuados:

Cenário 1: Validação de layout com tags

Descrição: Classe Calculadora – Teste do método somar:

Figura 22 – Testes Cenário 1: Método Somar



Fonte: Autoria Própria

Figura 23 – Testes Cenário 1: TesteCalculadora.java

Calculadora.java | TesteCalculadora.java

```java
import br.com.projetoddtunit.framework.ProvedorDeDados;

@RunWith(Parameterized.class)
public class TesteCalculadora extends TestCase {

    private double num1;
    private double num2;
    private double resultadoEsperado;

    @Parameters
    public static Collection<Object[]> DadosPlanilha() throws IOException {
        InputStream planilha = new FileInputStream("src/Dados2.xls");
        return new ProvedorDeDados(planilha).obterDados();
    }

    public TesteCalculadora(double num1, double num2, double resultadoEsperado) {
        this.num1 = num1;
        this.num2 = num2;
        this.resultadoEsperado = resultadoEsperado;
    }

    /**
     * Estes testes vão passar pois o resultado esperado é o resultado de uma soma */
    @Test
    public void testeSomar() {
        double valorCalculado = Calculadora.somar(num1, num2);
        assertThat(valorCalculado, is(resultadoEsperado));
    }

    /**
     * Estes testes vão falhar pois o resultado esperado não é o resultado de uma subtração */
    @Test
    public void testeSubtrair() {
        double valorCalculado = Calculadora.subtrair(num1, num2);
        assertThat(valorCalculado, is(resultadoEsperado));
    }
}
```

Fonte: Autoria Própria

Figura 24 – Testes Cenário 1: Fonte de Dados do Teste

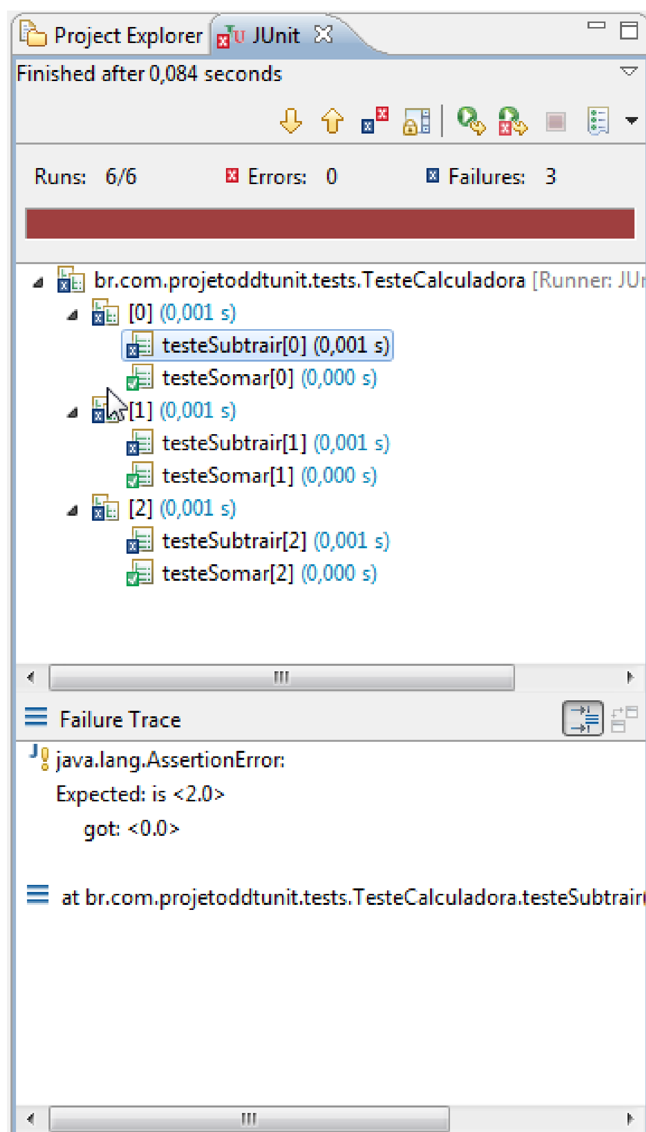
Dados2.xls [Modo de Compatibilidade] - Microsoft ...

	A	B	C	D
1	<Caso de Teste>	<num 1>	<num 2>	<resultado esperado>
2	<CT01: Soma valores válidos>	1	1	2
3	<CT02: Soma com valor negativo>	1	-1	0
4	<CT03: Soma de valores negativos>	-1	-1	-2

Teste_Calculadora

Fonte: Autoria Própria

Figura 25 – Testes Cenário 1: Resultado JUnit



Fonte: Autoria própria

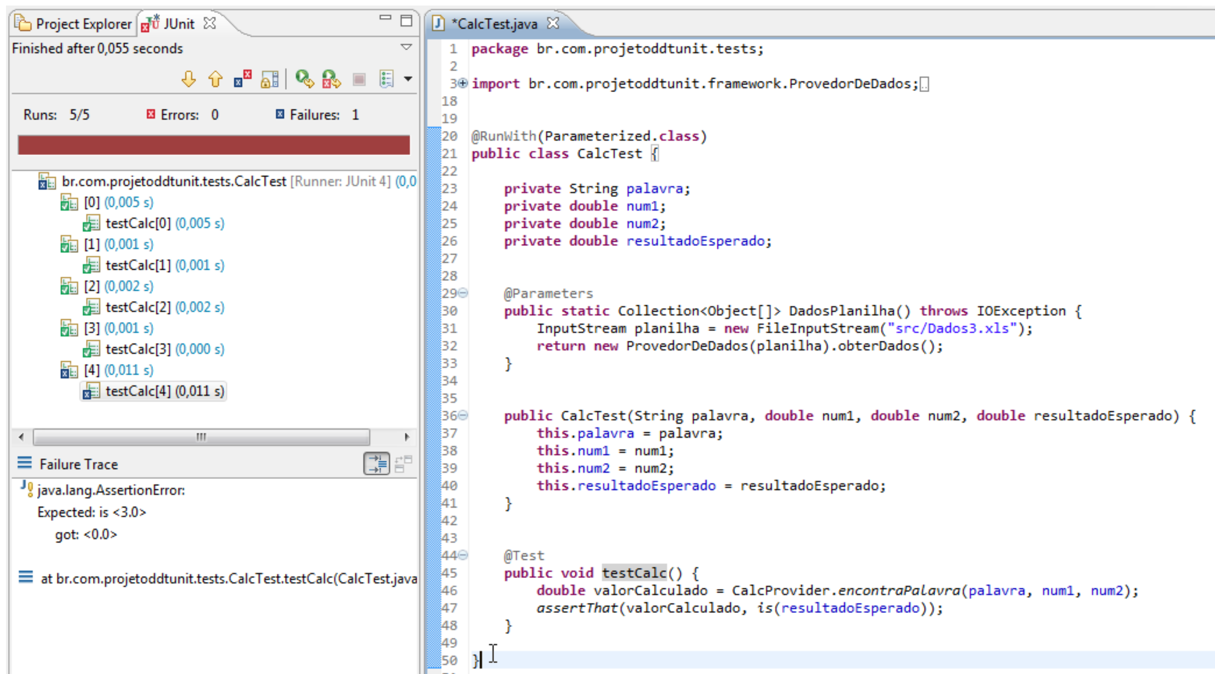
Observa-se na figura acima que para cada linha da planilha foi criado uma linha de teste que testa os métodos soma e subtrair utilizando os mesmos parâmetros, devido a isso, o método subtrair falhou em todos os testes, pois o resultado esperado (determinado na planilha) não corresponde ao resultado retornado pelo método. É possível ainda, verificar a falha exibida pelo *Failure Trace* que mostra que o resultado esperado é diferente do resultado obtido.

Cenário 2: Validação de layout utilizando vínculo de palavras com métodos

Descrição: Classe Calculadora – Teste do método somar e subtrair:

Analise a figura 26 e 27 (próxima página):

Figura 26 – Testes Cenário 2: Classe de teste e resultado JUnit



Fonte: Autoria Própria

Figura 27 – Testes Cenário 2: Fonte de Dados do Teste

Dados3.xls [Modo de Compatibilidade] - Microsoft ...

	A	B	C	D
1	<Test Case>	<num 1>	<num 2>	<resultado esperado>
2	somar	1	1	2
3	somar	1	-1	0
4	somar	-1	-1	-2
5	subtrair	1	2	-1
6	subtrair	1	1	3

Teste_Calculadora

Fonte: Autoria própria.

Pode-se observar neste cenário, que falhou um teste, devido ao resultado esperado não ser igual ao resultado obtido, porém, na apresentação dos resultados do JUnit, não é possível verificar qual o método que falhou (somar ou subtrair), é possível fazer esta identificação somente analisando as linhas da planilha e os índices das linhas exibidas no resultado do JUnit.

Cenário 3: Utilização do framework para validação da unidade por meio de técnicas de teste funcional

Este cenário foi criado no intuito de mostrar a principal vantagem na utilização do *framework*. No exemplo da calculadora, por possuir poucos cenários e casos de teste, o uso do *framework* pode ser questionado, dessa forma, pode-se considerar o cenário a seguir:

1. Cliente pode possuir o status novo, ativo, *master* e inadimplente;
2. O sistema (função) deve calcular desconto de acordo com as seguintes regras:

2.1 Clientes com status novo:

- não recebem desconto se nº de itens comprados for Inferior a 10;
- recebem 5% desconto para compras entre 10 e 99 itens;
- recebem 10% de desconto acima de 100 itens.

2.2 Clientes com status ativo:

- recebem 3% de desconto para compras abaixo de 10 itens;
- 10% de desconto entre 10 e 99 itens;
- 15% de desconto acima de 100 itens.

2.3 Clientes com status master:

- recebem 5% de desconto para compras abaixo de 10 itens;
- 15% de desconto entre 10 e 99 itens;
- 25% de desconto acima de 100 itens.

2.3 Clientes com status inadimplente:

- não recebem desconto.

Utilizando a técnica de teste funcional partição de equivalência, foi possível criar vinte e três casos de teste para validação da função para cálculo do desconto no cenário proposto. Na figura 28, pode-se observar os casos de teste criados e os dados (parâmetros) que serão enviados para função a ser testada, bem como o resultado esperado para cada caso de teste.

Por meio das respectivas figuras 29 e 30 é possível verificar o código fonte da classe que irá testar a função utilizando o *framework* e o resultado do teste exibido pelo JUnit.

Figura 28 – Teste cenário 3: Fonte de dados para validação da função de desconto

<caso de teste>	<status cliente>	<qtde itens>	< % desconto esperado>
<ct01 - cliente novo sem desconto>	novo	1	0
<ct02 - cliente novo sem desconto>	novo	9	0
<ct03 - cliente novo com 5% desconto>	novo	10	5
<ct04 - cliente novo com 5% desconto>	novo	99	5
<ct05 - cliente novo com 10% desconto>	novo	100	10
<ct06 - cliente novo com 10% desconto>	novo	13517	10
<ct07 - cliente comum com 3% desconto>	ativo	1	3
<ct08 - cliente comum com 3% desconto>	ativo	9	3
<ct09 - cliente comum com 10% desconto>	ativo	10	10
<ct10 - cliente comum com 10% desconto>	ativo	99	10
<ct11 - cliente comum com 15% desconto>	ativo	100	15
<ct12 - cliente comum com 15% desconto>	ativo	5000	15
<ct13 - cliente master com 10% desconto>	master	1	5
<ct14 - cliente master com 10% desconto>	master	9	5
<ct15 - cliente master com 15% desconto>	master	10	15
<ct16 - cliente master com 15% desconto>	master	99	15
<ct17 - cliente master com 25% desconto>	master	100	25
<ct18 - cliente master com 25% desconto>	master	30000	25
<ct19 - cliente com desconto bloqueado>	inadimplente	1	0
<ct20 - cliente com desconto bloqueado>	inadimplente	9	0
<ct21 - cliente com desconto bloqueado>	inadimplente	10	0
<ct22 - cliente com desconto bloqueado>	inadimplente	99	0
<ct23 - cliente com desconto bloqueado>	inadimplente	100	0

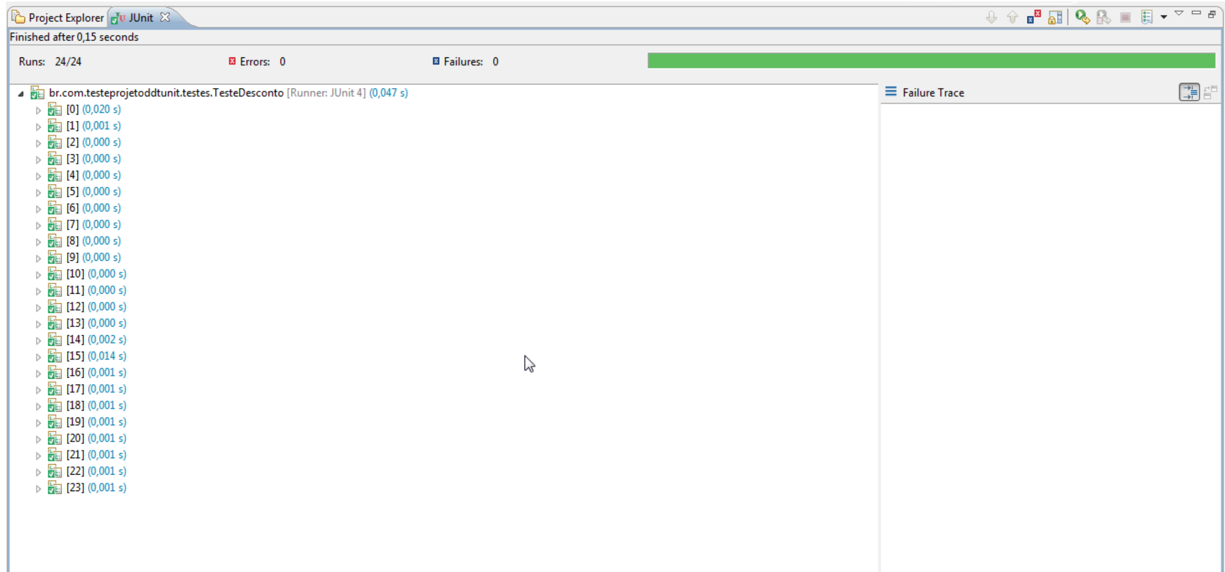
Fonte: Autoria Própria

Figura 29 – Teste cenário 3: classe de teste

```java
@RunWith(Parameterized.class)
public class TesteDesconto {

    private String tipoCliente;
    private double qtdeProduto;
    private double descontoEsperado;

    @Parameters
    public static Collection<Object[]> DadosPlanilha() throws IOException {
        InputStream planilha = new FileInputStream("src/DadosTesteDesconto.xls");
        return new ProvedorDados(planilha).obterDados();
    }

    public TesteDesconto(String tipoCliente, double qtdeProduto, double desconto) {
        this.tipoCliente = tipoCliente;
        this.qtdeProduto = qtdeProduto;
        this.descontoEsperado = desconto;
    }


    @Test
    public void testeDesconto() {
        double descontoRetornado = Desconto.calculaDesconto(tipoCliente, qtdeProduto);
        assertThat(descontoRetornado, is(descontoEsperado));
    }
}
```

Fonte: Autoria Própria.

Figura 30 – Testes Cenário 3: resultado do JUnit



Fonte: Autoria Própria

Por meio deste cenário, é possível notar que a utilização de uma fonte externa tornou o código para teste simples e legível, além disso, ao utilizar uma fonte externa para validação da função foi possível de uma maneira usual, utilizar técnicas de teste funcional para criação de vários casos de testes sem afetar o código fonte para o teste unitário.

CONCLUSÃO

Tendo em vista a importância da qualidade de *software* no cenário atual, a condução deste trabalho visou contribuir com a área de teste de *software*, visando otimizar e qualificar as atividades dentro dessa disciplina.

A contribuição desse trabalho foi dada por meio de estudos sobre a área de teste de *software*, levantando seu papel dentro do desenvolvimento de *software*, destacando o paralelismo entre desenvolvimento e testes de *software* e a importante integração que deve ocorrer entre essas duas equipes. Em seguida, enfatizou-se a fase de teste unidade, e como as atividades nessa fase refletem na qualidade do *software*, ainda podendo ser aplicadas nessa fase, técnicas de testes funcionais para a garantia de da qualidade do *software*, além da menção a metodologia de desenvolvimento dirigido a testes que se beneficia com as técnicas e ferramentas voltadas para o teste unitário.

Neste contexto, introduziu-se a automação de teste que contribui com o aperfeiçoamento das atividades de teste e desenvolvimento, as metodologias e ferramentas aqui apresentadas viabilizam ainda mais a aplicação de teste unitário, por possibilitar facilidade na criação de testes, custo relativamente baixo, maior e mais rápido retorno do investimento em relação aos recursos para automatização, menor curva de aprendizagem e por possuir diversas ferramentas que suportam o teste unitário. As vantagens aumentam ao inserir a abordagem de teste unitário orientado a dados, por possibilitar que as atividades de desenvolvimento se unam com as atividades de teste a fim de encontrar falhas precocemente, otimizando todo o processo de desenvolvimento, e garantindo a qualidade do *software*.

Dessa forma, ao estudar e analisar as ferramentas disponíveis que dão suporte ao teste unitário, o objetivo desse trabalho foi definido ao propor uma alternativa que otimiza a criação de testes unitários orientado a dados. E por meio das ferramentas existentes atualmente e estudos efetuados das mesmas foi possível desenvolver um *framework* que permite suporte ao desenvolvedor durante os testes unitários, possibilitando legibilidade nos códigos de teste, desacoplamento de dados de teste em *scritps* de teste, maior cobertura dos testes, facilidade na manutenção dos dados de teste e scripts, aumentando assim a probabilidade de encontrar erros na fase onde sua correção custa menos.

O *framework* proposto é reutilizável e proporciona de uma maneira simples a execução automatizada de testes unitários orientado a dados, unindo a simplicidade do JUnit a popularidade e facilidade de uso do Excel.

Ao utilizar o *framework* proposto, espera-se que o desenvolvedor tenha maior produtividade na criação dos testes unitários e na manutenção dos mesmos, bem como na criação e especificação dos dados que serão utilizados no teste, neste aspecto o testador poderá auxiliar na criação dos testes, possibilitando eficácia e maior cobertura.

A aplicação do *framework* como ferramenta de teste na fase de teste de unidade é possível em qualquer metodologia de desenvolvimento em que se considera a fase de teste de unidade.

Após o desenvolvimento do *framework*, concluiu-se que o JUnit na abordagem de teste unitário orientado a dados, possui limitações e afeta a flexibilidade do framework por necessitar passar os argumentos do teste no construtor da classe de teste.

Como ações futuras para a evolução do *framework*, pode-se destacar o estudo para utilização do mesmo em conjunto com o *framewok* de teste unitário TestNG, estudo sobre a viabilidade de aplicar a abordagem de automação *keyword data driven* ao *framework* e melhoria no *core* para utilização de documentos com as novas extensões do Office.

REFERÊNCIAS

ABNT. ABNT – Associação Brasileira de Normas Técnicas. **Engenharia de Software – Qualidade de Produto. Parte 1: Modelo de Qualidade**. NBR ISO/IEC 9126-1. 2003

AHMED, Ashfaque. **Software testing as a service**. United States: Taylor & Francis Group, 2010.

ASTELS, David. **Test-Driven Development: A Practical Guide**. New Jersey: Prentice Hall PTR, 2003.

BASTOS, Aderson et al. **Base de Conhecimento em Teste de Software**. 2.ed. São Paulo: Martins, 2007.

BECK, Kent. **Test-Driven Development By Example**. Boston: Pearson Education, 2003.

BERNARDO, Paulo C.; KON, Fabio. **A Importância dos Testes Automatizados**. Engenharia de Software Magazine, 3.ed., p. 54-57, 2008.

BEUST, Cédric; SULEIMAN, Hani. **Next Generation Java Testing: TestNG and Advanced Concepts**. Boston: Pearson Education Inc, 2008.

BLACK, Rex; MITCHELL, Jamie. **Advanced software testing: guide to the ISTQB advanced certification as an advanced technical test analyst**. v.3. United States: Rocky Nook, 2011.

BLACK, Rex. **Managing The Testing Process**. Washington: Microsoft Press, 1999.

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

CAETANO, Cristiano. **Automação de Testes: Mitos e Verdades**. Qualister. Disponível em: <http://www.slideshare.net/cristianocaetano/automao-de-testes-mitos-e-verdades-qualister>. Acesso em: 20 out. 2012

CRAIG, Rick D.; JASKIEL, Stefan P. **Systematic Software Testing**. London: Artech House Publishers, 2002.

CRISPIN, Lisa; GREGORY, Janet. **Agile Testing: a practical guide for testers and agile teams**. Boston: Pearson Education, 2009.

COHN, Mike. **Suceeding With Agile**. Software Development Using Scrum. United States: Addison Wesley, 2010.

DELAMARO, M. E.; MALDONADO, J.C.; JINO, M. Conceitos Básicos. In: _____. **Introdução ao Teste de Software**. Rio de Janeiro: Elsevier, 2007. cap. 1. p.1-7.

DDTUnit. **DDTUnit – A Data Driven Approach to Unit Testing**. SourceForge. 2007. Disponível em: < http://ddtunit.sourceforge.net/ddtunit.pdf>. Acesso em: 2 jun. 2012.

FABBRI, S. C. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Teste Funcional. In: DELAMARO, M. E.; MALDONADO, J.C.; JINO, M. **Introdução ao Teste de Software**. Rio de Janeiro: Elsevier, 2007. cap. 2. p.9-25.

FANTINATO, Marcelo et.al. **AutoTest – Um Framework Reutilizável para a Automação de Teste Funcional de Software**. Campinas. Disponível em: <http://www.each.usp.br/fantinato/files/sbqs24_Fantinato.pdf>. Acesso em: 8 set 2012.

FREEMAN, Steve; PRYCE, Nat. **Growing object-oriented software, guided by tests**. Boston: Pearson Education, 2010.

FEWSTER, Mark; GRAHAM, Dorothy. Software Test Automation: effective use of test execution tools. Boston: ACM Press, 1999.

GLOVER, Andrew. **In pursuit of code quality: JUnit 4 vs TestNG**. IBM – Developer Works. 2006. Disponível em: <http://www.ibm.com/developerworks/java/library/j-cq08296/>. Acesso em: 4 ago. 2012.

HETZEL, Bill. **The Complete Guide to Software Testing**. 2.ed. United States: John Wiley & Sons, 1998.

HUNT, Andrew; THOMAS, David. **Pragmatic Unit Testing**: In Java with JUnit. United States: The Pragmatic Bookshelf, 2003.

IEEE. **IEEE Standart Glossary of Software Engineering Terminology**. IEEE Std 610.12-1990, p. 1, 1990.

ISTQB. **Certified Tester – Foundation Level Syllabus**. 2011. Disponível em: <http://www.bstqb.org.br/uploads/docs/syllabus_ctfl_2011br.zip > Acesso em: 18 mai 2012;

JUNIT. Java Documentation. Disponível em: < http://junit.sourceforge.net/javadoc/>. Acesso em: 10 jun. 2012

KOSKELA, Lasse. **Test Driven. Practical TDD and Acceptance TDD for Java Developers**. United States: Manning Publications Co, 2008.

LI, Kanglin; WU, Menqi. **Effective Software Test Automation: developing an automated software testing tool**. London: Sybex Inc, 2004.

MASSOL, Vincent; HUSTED, Ted. **JUnit in Action**. United States: Manning Publications Co, 2004.

MESZAROS, Gerard. **xUnit Test Patterns: refactoring Test Code**. Boston: Pearson Education, 2007.

MOREIRA, Anderson. **Testes Unitários com JUnit**. Disponível em: <http://siep.ifpe.edu.br/anderson/blog/?page_id=976>. Acesso em: 3 out. 2012

MYERS, Glenford J. **The Art of Software Testing**. 2.ed. New Jersey: John Wiley & Sons, 2004.

NETO, Arilo Claudio Dias. **Introdução a Teste de Software**. Disponível em: < http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acesso em: 01 out 2012

LAUKKANEN, Pekka. **Data-Driven and Keyword-Driven Test Automation Frameworks**. Disponível em: < http://eliga.fi/Thesis-Pekka-Laukkanen.pdf>. Acesso em: 7 jul. 2012.

OSHEROVE, Roy. **The Art of Unit Testing with Examples in .NET**. United States: Manning Publications Co, 2009.

PATTON, Ron. **Software Testing**. United States: Sams Publishing, 2006.

POI API Documentation. Disponível em: < http://poi.apache.org/apidocs/index.html>. Acesso em: 20 out. 2012

PRESSMAN, Roger S. **Software Engineering: a practitioner’s approach**. 5.ed. United States: McGraw-Hill, 2001.

RAINSBERGER, J. B. **JUnit Recipes: practical methods for programmer testing**. United States: Manning Publications Co, 2005.

RILEY, Tim; GOUCHER, Adam. **Beautiful Testing**. United States: O’ Reilly Media, 2010.

SAKURAI, Rafael. **TDD – Desenvolvimento Guiado por Testes**. Disponível em: < http://www.universidadejava.com.br/docs/tdd-desenvolvimentoguiadoportestes> Acesso em: 10 out. 2012.

SMART, John Ferguson. **Data-driven tests with JUnit 4 and Excel**. 2009. Comunidade Java.net. Disponível em: < http://weblogs.java.net/blog/johnsmart/archive/2009/11/28/data-driven-tests-junit-4-and-excel>. Acesso em: 12 mai. 2012.

SOMMERVILLE, Ian. **Engenharia de Software**. 6.ed. São Paulo: Addison Wesley, 2003.

VELOSO, Janielton de Sousa; NETO, Pedro de Alcântara dos S.; SANTOS, Ismayle de Sousa; BRITTO, Ricardo de Sousa. **Avaliação de Ferramentas de Apoio ao Teste de Sistemas de Informação**. Disponível em: < http://www.seer.unirio.br/index.php/isys/article/view/1295/1179>. Acesso em: 13 out. 2012