FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

RAFAEL SANTANA DE SOUZA

TESTES DE SOFTWARE PARA GARANTIR A QUALIDADE

Americana, SP

2016

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

RAFAEL SANTANA DE SOUZA

TESTES DE SOFTWARE PARA GARANTIR A QUALIDADE

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do Prof.º Me. Eduardo Antonio Vicentini.

Área de concentração: Engenharia de Software

Americana, SP

2016

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

S718t

Souza, Rafael Santana de
Testes de software para garantir a qualidade. / Rafael Santana de Souza. – Americana: 2016.
40f.

Monografia (Graduação em Tecnologia em Análise e Desenvolvimento de Sistemas). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Me. Eduardo Antonio Vicentini

1. Engenharia de software I. Vicentini, Eduardo Antonio II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.3.05

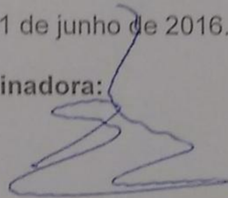
Rafael Santana de Souza

TESTES DE SOFTWARE PARA GARANTIR A QUALIDADE

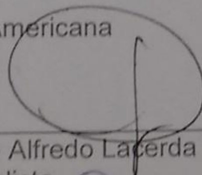
Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo CEETEPS/Faculdade de Tecnologia – Fatec/ Americana.
Área de concentração: Engenharia de Software

Americana, 21 de junho de 2016.

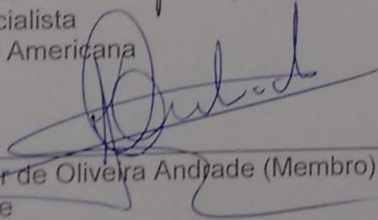
Banca Examinadora:



Eduardo Antonio Vicentini (Presidente)
Mestre
Fatec Americana



Antônio Alfredo Lacerda (Membro)
Especialista
Fatec Americana



Kléber de Oliveira Andrade (Membro)
Mestre
Fatec Americana

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer ao Centro Paula Souza e ao Governo do Estado de São Paulo por manter a Faculdade de Tecnologia de Americana, onde tive o prazer de estudar. O conhecimento obtido irá proporcionar muitas oportunidades na minha vida profissional.

Aos meus colegas de sala, onde caminhamos durante toda a trajetória da graduação e que tivemos muita cooperação e incentivo uns aos outros.

Aos professores que nos deram toda a atenção e acompanhamento em seus ensinamentos.

Ao meu orientador Eduardo Antonio Vicentini, pela paciência de me orientar durante o desenvolvimento deste trabalho.

Muito agradecido, obrigado!

DEDICATÓRIA

Aos meus pais, Osmar e Emília, que sempre me apoiaram para prosseguir com os estudos.

RESUMO

Por causa da precisão de garantir a qualidade de software, os testes de caixa branca são realizados em trechos do programa, durante a fase de desenvolvimento e pelos próprios desenvolvedores. Defeitos são encontrados e resolvidos no mesmo instante, evitando assim perca de tempo e dinheiro após a finalização de seu desenvolvimento. Para explicar sobre esse tema, esta pesquisa irá explorar fundamentos de testes de software: seus objetivos, modalidades e alguns testes. Os testes citados como exemplo são testes de caixa branca e serão apresentados com alguns exemplos como: caminho básico, fluxo de dados, condições, laços de repetição e testes de caixa branca adaptados para programação orientada à objetos. Após os estudos de algumas técnicas, é aplicado o que foi aprendido em trechos de código de um programa real, utilizando também a ferramenta de auxílio JUnit 4 e a linguagem Java, algumas imagens irão ilustrar passo-a-passo dos testes. A maior importância da pesquisa é mostrar que testes de softwares são meios importantes para garantir a qualidade de software.

Palavras Chave: Qualidade de software, Caixa branca, Desenvolvimento, Testes de Software, JUnit 4, Java

ABSTRACT

Because of the precision to ensure software quality, white-box tests of program segments during the development phase and the developers themselves. Defects are found and fixed at once, thereby avoiding waste of time and money after completion of its development. To explain on this topic, this research will explore reasons Software Testing: its objectives, methods and tests. The tests are cited as an example and white box tests with some examples will be presented as basic path, data flow conditions, repetitive loops and white box test tailored for oriented programming objects. After the studies of some techniques, which is applied has been learned in a real snippets of program code, also using JUnit support tool 4 and the Java language, some images will illustrate step-by-step test. The increased importance of the research is to show that the software testing are important means to ensure software quality.

Keywords: Software Quality, White Box, Development, Software Testing, JUnit 4, Java

SUMÁRIO

1. INTRODUÇÃO 11

2. FUNDAMENTOS DE TESTES DE SOFTWARE 13

2.1. VERIFICAÇÃO E VALIDAÇÃO 14

2.2. OBJETIVO DOS TESTES DE SOFTWARE 14

2.3. MODALIDADES DE TESTES 15

2.3.1. TESTES DE CAIXA PRETA 15

2.3.2. TESTES DE CAIXA BRANCA 17

2.3.2.1. CAIXA BRANCA EM POO 18

3. ALGUNS TESTES DE SOFTWARE 19

3.1. CAMINHO BÁSICO 19

3.2. CONDIÇÃO 20

3.3. FLUXO DE DADOS 21

3.4. LAÇOS 24

3.5. UNIDADE EM PROGRAMAS OO 26

3.6. INTEGRAÇÃO EM PROGRAMAS OO 27

3.7. PAR-A-PAR 27

4. ESTUDO DE CASO 29

4.1. FOX (PONTO DE CAIXA) 29

4.2. FOX (PONTO DE CAIXA) - TESTES 30

5. CONSIDERAÇÕES FINAIS 38

REFERÊNCIAS 40

LISTA DE FIGURAS E DE TABELAS

FIGURA 1 - FASES DE TESTE 15
FIGURA 2 - MENOR UNIDADE: MÉTODO E CLASSE 18
FIGURA 3 - GRAFO DE FLUXO 20
FIGURA 4 - FUNÇÃO MAIN DO PROGRAMA 23
FIGURA 5 - GRAFO DEF-USO 24
FIGURA 6 - EXEMPLOS DE LAÇOS 25
FIGURA 7 - CLASSE SYMBOLTABLE 26
FIGURA 8 - MÉTODOS ENTRE CLASSES 28
TABELA 1 - PARES DO CASO DE TESTE PAR-A-PAR 28
TABELA 2 - ESTRUTURA DO SISTEMA FOX 29
FIGURA 9 - MENU DO SISTEMA FOX 30
TABELA 3 - ALGUMAS INSTRUÇÕES DO JUNIT 4 31
FIGURA 10 - CÓDIGO DO BOTÃO INSERIR PRODUTO 32
FIGURA 11 - CÓDIGO DOS MÉTODOS VERIFICACODIGO() E VERIFICAMOEDA() 33
FIGURA 12 - TESTES DOS MÉTODOS VERIFICACODIGO() E VERIFICAMOEDA() 34
FIGURA 13 - CÓDIGO DO MÉTODO EXISTEPRODUTO() 35
FIGURA 14 - TESTE DO MÉTODO EXISTEPRODUTO() 35
FIGURA 15 - TESTE DO MÉTODO EXISTEPRODUTO() COM SERVIDOR PARADO 36
FIGURA 16 - CÓDIGO DO MÉTODO INSERIRPRODUTO() 36
FIGURA 17 - TESTE DO MÉTODO INSERIRPRODUTO() 37

1. INTRODUÇÃO

Na atualidade, a demanda do mercado necessita que os produtos e serviços sejam cada vez mais eficientes. Para isso, as organizações estão cada vez mais se preocupando com a qualidade da prestação de seus produtos e serviços.

As organizações estão investindo tempo e dinheiro sobre as atividades empresariais. Elas estão entendendo e compreendendo que a tecnologia pode ser uma grande aliada para conseguir melhorar e evoluir seus processos.

Contando com a tecnologia e softwares que auxiliam no desempenho das empresas, como tomada de decisões através de informações geradas por eles, aumenta também o nível da necessidade de qualidade desses sistemas.

Com essa situação bem preocupante, as empresas de tecnologia e desenvolvimento de sistemas, precisam desenvolver bons softwares e para isso cada vez mais, é investido em atividades de teste de software. Este, que procura garantir o bom funcionamento do sistema, através de técnicas desenvolvidas especificamente para apontar e revelar defeitos, que são irregularidades, no tratamento dos programas.

Para tanto o estudo se justificou, segundo Pressman (2011) a atividade de testes é um importante e crítico elemento para garantir a qualidade do software desenvolvido. Nela consta a última revisão da especificação do projeto e da codificação.

A pergunta que se buscou responder foi: como garantir a qualidade dos sistemas computacionais, através de funções em nível de código?

O objetivo geral é aplicar algumas técnicas de teste para garantir a qualidade de um sistema computacional real.

Como objetivos específicos têm-se: a) entender através de uma pesquisa do que se tratam testes de software. b) aprender diferentes técnicas de teste.

O método científico de pesquisa utilizado foi de pesquisa aplicada. Com relação aos procedimentos técnicos, utilizou-se a pesquisa bibliográfica em livros, artigos científicos e documentos especializados.

O trabalho foi estruturado em quatro capítulos, sendo que o segundo conceitua testes de softwares, explicando sobre seus fundamentos, definindo verificação e validação, objetivos dos testes e suas modalidades, o terceiro apresentam algumas técnicas de caixa branca como: caminho básico, condição, fluxo de dados, laços, unidade em programas OO, integração em programas OO e testes par-a-par. Por fim, o quarto capítulo foi destinado à aplicação de testes de software em um estudo de caso, onde é aplicado em um software comercial de ponto de caixa.

2. FUNDAMENTOS DE TESTES DE SOFTWARE

Este trabalho tem por objetivo entender sobre a necessidade de testes de softwares e aprender algumas técnicas de testes para aplicar em um sistema real afim de encontrar possíveis defeitos, os corrigindo para garantir a qualidade.

Este capítulo apresenta fundamentos de testes de software para uma melhor compreensão do leitor, facilitando o entendimento dos próximos capítulos.

Segundo Pressman (2011, p.32) software consiste em programas de computador que, quando executadas, fornecem características, funções e desempenho desejados; estruturas de dados que possibilitam aos programas manipular informações adequadamente, e informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso dos programas.

Segundo Sommerville (2007, p. 6) um processo de software é um conjunto de atividades e resultados associados que produz um produto de software. Existem quatro atividades fundamentais de processos, que são comuns a todos os processos de software. São elas:

1. Especificação de software, onde os engenheiros e os clientes definem os requisitos funcionais do software, ou seja, o que será desenvolvido e as suas restrições.

2. Desenvolvimento de software é o processo no qual o sistema é desenvolvido.

3. Validações de software, onde o software é analisado para verificar se é o que o cliente precisa.

4. Evolução de software, o software é atualizado para se enquadrar às mudanças do cliente e do mercado.

2.1. VERIFICAÇÃO E VALIDAÇÃO

Segundo Sommerville (2007, p. 52), o processo de Verificação e Validação (V & V) está relacionado em mostrar que o sistema atendeu o que o cliente precisava ao adquirir o software e que está em conformidade com o que foi especificado.

Pressman (2011, p. 402) informa que a validação é um conjunto de atividades para garantir que o software desenvolvido é o que o cliente esperava, ou seja, as suas exigências. A verificação possui relação com atividades onde o software realiza uma função específica corretamente. Para entendermos:

Verificação: verifica se um botão, como exemplo, está funcionando corretamente.

Validação: verifica se esse botão está de acordo com o que o cliente precisa?

Durante o processo de software, existem dois tipos de testes. Teste de validação: é confirmado se o que foi desenvolvido é o que o cliente deseja. Teste de defeitos: encontra possíveis defeitos no software, para que sejam corrigidos e garantido a qualidade.

2.2. OBJETIVO DOS TESTES DE SOFTWARE

Os testes são importantes, pois estes são meios de detecção e correção de defeitos, ocasionando a qualidade do software.

Pressman (2007, p. 788) expõe três objetivos principais da atividade de testes:

1. A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro;
2. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto;
3. Um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

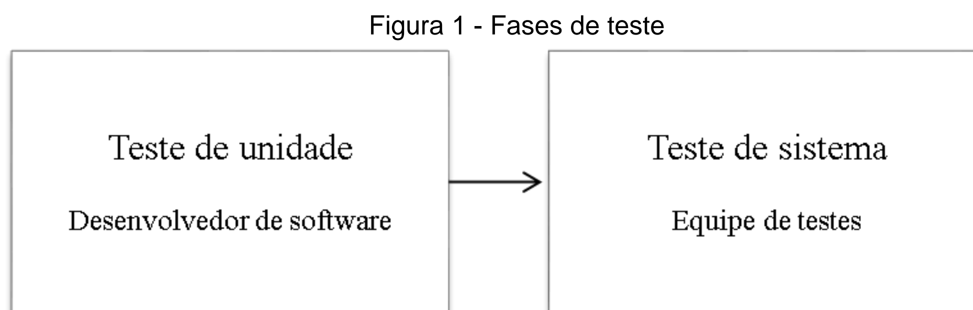
Estes três objetivos esclarecem que os testes de software bem-sucedidos são os que encontram erros ainda não descobertos.

2.3. MODALIDADES DE TESTES

Segundo Sommerville (2007), todos os tipos de testes se agrupam em duas modalidades fundamentais, o teste de sistema, onde os requisitos funcionais são testados no software completo e o teste de componentes ou teste de unidade, no qual são testados componentes separadamente. Estes testes geralmente ocorrem durante o processo de desenvolvimento pelos próprios desenvolvedores.

O objetivo do teste de sistema é realizar uma verificação para confirmar se o sistema atende os requisitos do negócio, como por exemplo: entrada e saída de dados, registro no banco de dados, telas e interface. O teste de unidade possui a função de realizar os testes em partes distintas do programa, podendo ser trechos de códigos, métodos e componentes do software. A figura 1 mostra em detalhes as responsabilidades dos dois testes.

Figura 1 - Fases de teste



Fonte: Sommerville (2007)

Para simplificar, testes de sistema são conhecidos como testes de caixa-preta e testes de unidade como testes de caixa-branca.

2.3.1. TESTES DE CAIXA PRETA

Pressman (2011, p.439) revela que os métodos de teste caixa preta (*Black-box*) concentram-se nos requisitos funcionais do software. Permitindo assim que seja criado pelo Engenheiro de Software, conjuntos de entrada de dados para testar os

requisitos como por exemplo: o cadastro de dados sendo validado por caracteres específicos. Segundo o autor, os erros nesta categoria de testes são os seguintes:

1. Funções incorretas ou ausentes;
2. Erros de interface;
3. Erros de estrutura de dados e no acesso a bancos de dados;
4. Erros de desempenho;
5. Erros de inicialização e término.

Já Sommerville (2007, p. 359) diz que o sistema é tratado como uma verdadeira caixa preta, cujo comportamento pode ser determinado por meio do estudo de suas entradas e saídas relacionadas. Também é conhecido como teste funcional.

Existem alguns métodos para este teste de sistema, são eles:

- **Teste de desempenho,** o qual o sistema é testado em carga máxima para tratamento de erros e verificar seu comportamento, de forma que não cause travamento total do sistema ou inconsistência de dados;
- **Particionamento de equivalência,** (Pressman, 2011) escreve que é um método de caixa preta que divide o domínio de entrada de um programa em classes de dados a partir das quais os casos de teste podem ser derivados. Por exemplo, o processamento incorreto dos dados do tipo data (date);
- **Análise de valor limite,** que propõe valores de entrada nos extremos dos limites permitidos, pois ainda segundo o autor, por razões que não são completamente claras, um número maior de erros tende a ocorrer nas fronteiras do domínio de entrada do que no “centro”, por exemplo, um campo que permita valores positivos menores que 1000 deve ser testados com valores que fiquem em torno de -1, 0, 1 e 999, 1000 e 1001;
- **Testes de comparação** são executados em sistemas críticos que necessitam de maior atenção na sua confiabilidade de informações, como um software aeronáutico, por exemplo. Dois sistemas geralmente

trabalham em conjunto para validação de dados em redundância. O teste neste caso tem por objetivo comparar os resultados de ambos, normalmente feitos através de ferramentas automatizadas;

- **Técnicas de grafo de causa-efeito** são utilizadas no projeto de casos de teste. Elas oferecem uma representação das condições lógicas de um programa e das ações correspondentes. Um grafo de causa-efeito é projetado, em seguida é convertido em uma tabela de decisão e somente a partir daí o engenheiro desenvolve um caso de teste.

Os métodos caixa preta não serão descritos em detalhes, pois este estudo concentra-se em métodos de teste caixa branca.

2.3.2. TESTES DE CAIXA BRANCA

Os métodos de caixa branca garantem que os caminhos independentes dentro de um determinado trecho de código do sistema sejam executados ao menos uma vez; teste as estruturas lógicas para valores falsos ou verdadeiros; percorra todos os laços de repetição em seus limites e abrangência; e valide as estruturas de dados internas em nível de linguagem de software.

Porém, segundo Delamaro, Maldonado e Jino (2007) o teste baseado nestes métodos mostram-se pouco eficaz porque os programas, mesmo que pequenos, podem possuir um número de caminhos lógicos infinitos.

Mas, o autor aponta que essa desvantagem é vista como complementar às demais técnicas de teste, isso porque revela classes de defeitos diferentes e indetectáveis por outros padrões de teste.

Este trabalho trata de testes de caixa branca em um contexto de POO (programação orientada a objetos), logo, uma pesquisa bibliográfica será realizada para adaptar os conceitos dos métodos e aplicá-los.

2.3.2.1. CAIXA BRANCA EM POO

O teste de caixa branca em programas orientado a objetos podem ser divididos em duas fases: teste de unidade e teste de integração (Franchin, 2007, p. 34).

O teste de unidade, também é conhecido como teste **intramétodo** em OO, trata-se de testar a menor unidade de um sistema: os métodos individualmente. No teste **intermétodo**, referido ao teste de integração, são realizadas interações entre métodos dentro de uma mesma classe. Assim também como o teste **intraclasse**, e a sua diferença é que neste tipo, relaciona com chamadas de métodos públicos dentro de uma classe. O teste **interclasse** verifica o funcionamento da relação de polimorfismo, herança e acoplamento dinâmico entre classes. Uma das técnicas para o teste interclasse é a **par-a-par**.

Alguns autores utilizam conceitos diferentes com relação a menor unidade a ser testada em um teste de unidade: classe ou método Franchin (2007, p. 45). Este trabalho apresenta os critérios de tese estrutural para POO partindo do princípio do método como sendo a menor unidade. A figura 2 ilustra a diferença entre estes dois pontos de vista.

Figura 2 - Menor unidade: Método e Classe

Menor Unidade: Método

Fase	*Teste de Software Orientado a Objetos*
Unidade	Intra-método
Integração	Inter-método, Intra-classe e Inter-classe
Sistema	Toda a aplicação

Menor Unidade: Classe

Fase	*Teste de Software Orientado a Objetos*
Unidade	Intra-método, Inter-método e Intra-classe
Integração	Inter-classe
Sistema	Toda a aplicação

Fonte: Franchin (2007, p. 45)

3. ALGUNS TESTES DE SOFTWARE

Este capítulo apresenta algumas técnicas de testes de caixa branca. Pressman (2011) se refere a elas como testes de estrutura de controle, são elas:

1. Caminho básico;
2. Condição;
3. Fluxo de dados;
4. Laços (loops);
5. Unidade em programas orientados a objetos;
6. Integração em programas orientados a objetos;
7. Par-a-Par.

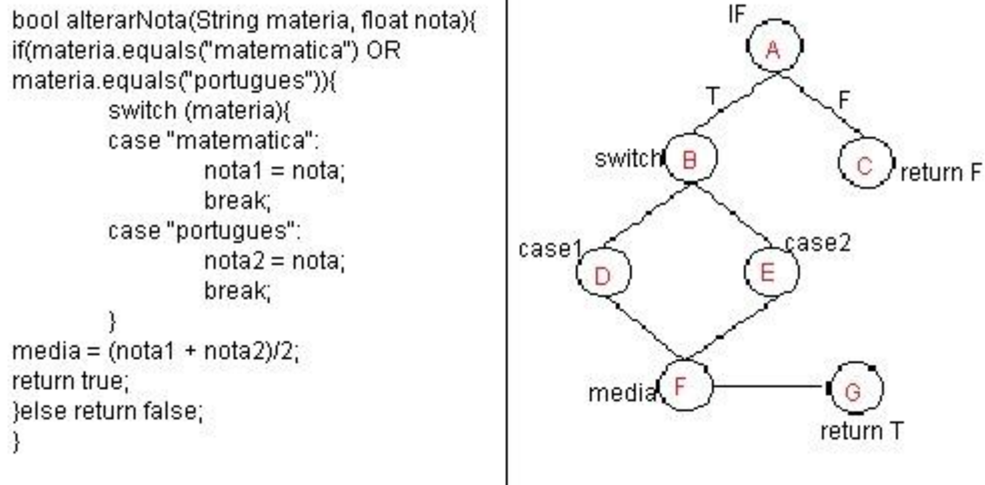
3.1. CAMINHO BÁSICO

Segundo Sommerville (2007), esta técnica de teste de caixa branca, consiste em testar todos os caminhos de um determinado trecho de código pelo menos uma vez, de forma que todas as possibilidades lógicas sejam testadas.

Entretanto, não é possível testar todas as combinações possíveis, porque existem programas com loops. Por este motivo, o teste de caminho básico é considerado como forma complementar em busca de defeitos na fase de desenvolvimento.

Pressman (2011) diz que para que o teste seja introduzido (mas não obrigatoriamente), utiliza-se um fluxograma (grafo de fluxo) para representar o trecho de código que será exercitado. O grafo de fluxo é composto por círculos, denominados nós, e setas, denominadas ramos, onde cada nó representa instruções procedimentais ou processamentos que são sequenciados logicamente pelos ramos. As áreas delimitadas pelos ramos e nós são denominadas regiões, conforme ilustrado na figura 3.

Figura 3 - Grafo de fluxo



Fonte: Autoria própria.

Através da métrica de software *complexidade ciclomática*, pode-se encontrar o número de caminhos independentes de um grafo de fluxo.

No exemplo acima, o nó A e B são chamados de nós predicativos, pois contém uma condição e são caracterizados por dois ou mais ramos que saem deles. Podemos calcular o número de caminhos independentes, C, seguindo a fórmula: C = P +1. Onde P é o número de nós predicativos do grafo.

C = 2 + 1 => C = 3 caminhos independentes

3.2. CONDIÇÃO

Este teste é utilizado para detectar defeitos nas condições lógicas. São elaborados casos que obtem o máximo de combinações de valores de entrada possíveis.

Existem dois tipos de condições: as simples e as compostas. As *condições simples* são representadas por variáveis booleanas ou expressões relacionais (>, ≥, =, ≠, < e ≤), como V1 <operador relacional> V2, onde V1 e V2 são variáveis numéricas ou expressões aritméticas. As *condições compostas* comportam duas ou mais condições simples separados por operadores booleanos: AND (&), OR (|) e NOT (!), como no exemplo: (V1 ≤ V2) ||(V2 ≠ V3)(PRESSMAN, 2011, p. 437).

Basicamente dois tipos de testes de condições são executados para exercitar o código em busca de erros: os *testes de domínio* e os *testes de ramos*. O teste de domínio precisa que sejam planejados três ou quatro valores para uma expressão relacional,

V1<operador relacional>V2

de forma que V1 e V2 sejam exercitados com valores menores, maiores ou equivalentes (<, >, =).

No teste de ramos, para uma condição C composta, os ramos verdadeiro e falso de C e todas as condições simples em C precisam ser executadas pelo menos uma vez.

Condição: (E1 > E2) & (E3)

No exemplo acima, supõe-se que E1 e E2 representem expressões aritméticas e E3 uma variável booleana. Desta forma temos uma condição composta formada por duas condições simples. Pressman (2011, p. 437), sugere uma técnica que utiliza restrições de condição para exercitar todas as combinações possíveis de uma condição composta. Esta técnica baseia-se em definir uma restrição de saída (*true* ou *false*) para cada condição simples. Para exercitar a condição do exemplo acima se utiliza as restrições: {t, t}, {f, t} e {t, f}.

Na primeira condição simples E1 > E2, substitui-se a restrição *true* pelo símbolo > e a restrição *false* pelos símbolos < e =, ficando assim: {(>, t), (<, t), (=, t), (>, f)}. Seguindo a técnica, a condição é testada de forma que possa garantir a detecção de erros de operadores relacionais.

3.3. FLUXO DE DADOS

Pressman (2011, p. 438), mostra que o método de fluxo de dados seleciona os caminhos válidos de um grafo de fluxo de acordo com suas variáveis chaves, identificando a sua definição e o seu uso no programa para gerar os casos de teste:

DEF(S) = {X} – A instrução S contém uma definição da variável X

USE(S) = {X} – A instrução S contém um uso da variável X

Uma **definição** de variável ocorre quando um valor é armazenado em uma posição de memória. Isto pode ocorrer quando a variável está do lado esquerdo de um comando de atribuição (var V = “valor”), em um comando de entrada (var X = textBox.getText()) ou em passagens como parâmetros por valor, referencia ou nome (function media (var X, var Y)).

Um **uso** de variável ocorre de duas maneiras: *uso computacional de variável* c-uso e *uso predicativo de variável* p-uso. O primeiro se refere a visualização de uma definição ou de uma computação realizada; o segundo afeta o fluxo de controle do programa, como em comandos *if, then, else* e *while*, *do*, *switch*. Observa-se que c-uso está relacionado com os nós de um grafo e p-uso com seus ramos. (Delamaro, Maldonado e Jino, 2007, p. 52).

Na figura 4, a função *main* do programa *Identifier* é responsável por determinar se um identificador é valido ou não e faz uso das funções *valid_s (char)* e *valid_f (char)* para isso, porém as duas funções não serão citadas para este exemplo:

Figura 4 - Função main do programa

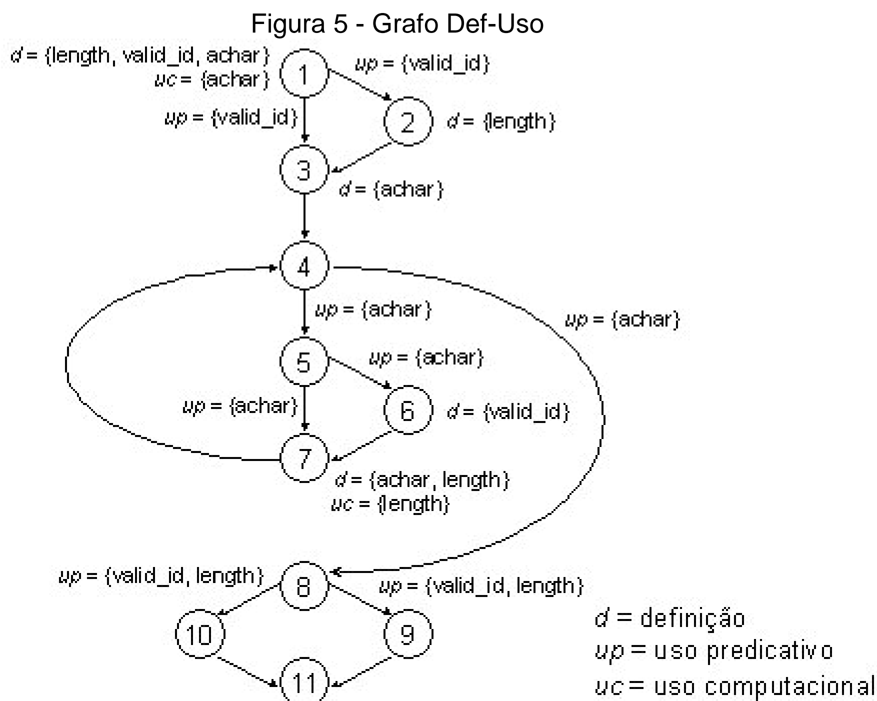
```
*01*  {
*01*     char achar;
*01*     int length, valid_id;
*01*     length = 0;
*01*     printf("Identificador:");
*01*     achar = fgetc(stdin);
*01*     valid_id = valid_s(achar);
*01*     if(valid_id)
*02*             length = 1;
*03*     achar = fgetc(stdin);
*04*     while(achar != 'n');
*05*     {
*05*             if(!(valid_f(achar)))
*06*                     valid_id = 0;
*07*             length++;
*07*             achar = fgetc(stdin);
*07*     }
*08*     if(valid_id && (length >= 1) && (length < 6))
*09*             printf("Valido\n");
*10*     else
*10*             printf("Invalido\n");
*11*  }
```

Fonte: Delamaro, Maldonado e Jino (2007, p. 52)

Nota-se que as linhas do código na figura 4 estão numeradas em blocos que vão do número 1 ao 11. Cada bloco representa um ou mais comandos que mudam o rumo da execução do programa. A partir daí forma-se os nós do grafo denominado Grafo Def-Uso, conforme apresentado na figura 5. Através do grafo Def-Uso é possível verificar os caminhos que deverão ser percorridos para exercitar a variável testada, ou seja, formular o caso de teste.

Pelo menos um caminho que percorra desde a definição de uma variável até o seu uso (não importa se c-uso ou p-uso) deve ser verificado sem que haja uma redefinição da mesma no meio do fluxo.

Figura 5 - Grafo Def-Uso



Fonte: Delamaro, Maldonado e Jino (2007, p. 61)

3.4. LAÇOS

O teste de laços (loops), ou teste de ciclos é uma técnica de caixa branca que tem a função de validar a construção de laços no código do programa. Segundo Pressman (2011, p. 438), quatro classes de laços podem ser definidas, são elas: laços *simples*, laços *concatenados*, laços *aninhados* e laços *não estruturados.* A figura 6 ilustra os tipos de laços.

Os **laços simples** são as estruturas mais comuns quando se trata de laços de repetição dentro de um programa, nele deve-se testar:

1. Pular o laço inteiramente;
2. Apenas uma passagem;
3. Duas passagens;
4. X passagens, sendo X < n, onde n é o número máximo de passagens;
5. n – 1, n, n + 1 passagens.

Os **laços aninhados** são um conjunto de laços um dentro do outro. Isso faz com que o número de testes se multiplique cada vez mais, dependendo da estrutura do laço. Para simplificar, Pressman (2011, p. 439) sugere uma abordagem que

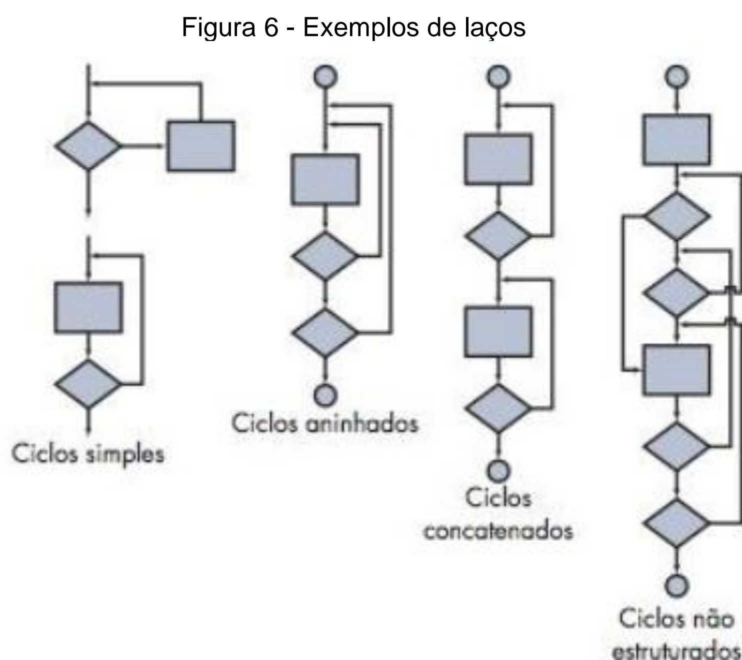
diminui a quantidade de testes. A técnica consiste em começar exercitando os laços mais internos da estrutura com os mesmos testes sugeridos para os *laços simples*, fixando os laços externos com valores mínimos, de forma que as repetições externas sejam pequenas. Esta técnica deve ser repetida de dentro para fora, passando de um laço para o outro, até que todos eles sejam exercitados.

Os **laços concatenados** podem ser tratados de duas formas. Se os laços forem independentes um do outro, deve-se usar a técnica apresentada para o teste de *laços simples*. Se houver um contador em comum entre os laços ou se este contador for usado como valor inicial para o outro, deve-se utilizar a abordagem apresentada no teste de *laços aninhados*.

Para os **laços não estruturados**, Pressman (2011, p. 439) recomenda que estes sejam *reprojetados* sempre que possível. Deve-se usar o bom senso e aplicar as técnicas apresentadas nesta mesma sessão.

A figura 6 representa graficamente os tipos de laços:

Figura 6 - Exemplos de laços



Fonte: Pressman (2011, p.438)

3.5. UNIDADE EM PROGRAMAS OO

A definição deste e dos seguintes critérios de teste estrutural apresentados neste capítulo foram desenvolvidos originalmente para testes em programas procedimentais, todavia ao longo do tempo eles vêm sendo adaptados para a programação orientada a objetos (Delamaro, Maldonado e Jino, 2007, p. 133).

A primeira e mais básica unidade a ser testada no teste estrutural em programação orientada a objeto é o método, ou seja, cada método deve ser testado individualmente dentro de uma classe. Segundo Franchin (2007, p. 34), este critério denomina-se teste **intramétodo**.

Figura 7 - Classe SymbolTable

```
01 // symboltable.h: definition
02 #include "symbol.h"
03
04 class SymbolTable {
05 private:
06   TableEntry *table;
07   int numentries, tablemax;
08   int *Lookup(char *);
09 public:
10   SymbolTable(int n) {
11      tablemax = n;
12      numentries = 0;
13      table = new TableEntry[tablemax]; };
14   SymbolTable() { delete table; };
15   int AddtoTable(char *symbol, char *syminfo);
16   int GetfromTable(char *symbol, char *syminfo);
17 };
18
19 // symboltable.c: implementation
20 #include "symboltable.h"
21
22 int SymbolTable::Lookup(char *key, int index) {
23   int saveindex;
24   int Hash(char *);
25   saveindex = index = Hash(key);
26   while (strcmp(GetSymbol(index),key) != 0) {
27     index++;
28     if (index == tablemax) /* wrap around */
29       index = 0;
30     if (GetSymbol(index)==0 || index==saveindex)
31       return NOTFOUND;
32   }
33   return FOUND;
34 }
35
```

```
36 int SymbolTable::AddtoTable(char *symbol,
                               char *syminfo) {
37   int index;
38   if (numentries < tablemax) {
39     if (Lookup(symbol,index) == FOUND)
40       return NOTOK;
41     AddSymbol(symbol,index);
42     AddInfo(syminfo,index);
43     numentries++;
44     return OK;
45   }
46   return NOTOK;
47 }
48
49 int SymbolTable::GetfromTable(char *symbol,
                                 char **syminfo) {
50   int index;
51   if (Lookup(symbol,index) == NOTFOUND)
52     return NOTOK;
53   *syminfo = GetInfo(index);
54   return OK;
55 }
56
69 void SymbolTable::AddInfo(syminfo,index)
70   ...
73   strcpy(table[index].syminfo,syminfo);
75 }
76
77 char *SymbolTable::GetInfo(index)
78   ...
82   return table[index].syminfo;
83 }
```

Fonte: Franchin (2007, p. 46)

No exemplo da figura 7, a classe SymbolTable implementa o método Lookup, onde a definição da variável *índex* é realizada na linha 27, e seu uso na linha 28.

Para analisar o fluxo de dados em programação OO, utiliza-se o conceito Def-Uso (Capítulo 2.3). Se um método M é chamado dentro de uma classe, e nele

exercita-se (v1, v2) como definição e uso, então (v1, v2) é um par def-uso intramétodo.

3.6. INTEGRAÇÃO EM PROGRAMAS OO

Depois do teste de unidade, Franchin (2007, p. 53) sugeriram os testes **intermétodo**, **intraclasse** e **interclasse** como testes estruturais de integração.

O teste de integração, assim como o de unidade (Capitulo 2.5), utiliza, o método de Def-Uso (Capitulo 2.3) para verificar as relações no fluxo de dados.

No teste **intermétodo**, as chamadas entre métodos são testadas em conjunto dentro de uma mesma classe. Como exemplo, o método AddtoTable (Figura 7) é testado percorrendo também os métodos AddSymbol, Lookup e AddInfo.

No teste **intraclasse** é feito chamadas a métodos públicos em sequências diferentes, tal como exemplo SymbolTable, AddtoTable e GetfromTable.

3.7. PAR-A-PAR

Ainda como um teste estrutural de integração, o teste par-a-par testa métodos aninhados entre classes. Dado um programa que possui um conjunto de classes C estruturadas com heranças ou polimorfismos que interagem entre si, C é organizado agrupando pares de métodos de onde são gerados os dados para a análise de fluxo de controle.

Como exemplo, a figura 8 ilustra um programa básico em que os métodos são aninhados entre classes. A classe A possui dois métodos, onde o primeiro chama o segundo, que por sua vez chama um terceiro método público da classe B (A.m1() -> A.m2() -> B.m3()). O método m3() da classe B faz a instancia de um objeto da classe C chamando um construtor. Depois executa o método m4() que também está dentro da classe C.

Figura 8 - Métodos entre classes

```
public classe A {
  public void m1() {
    ...
    m2();
    ...
  }
  public void m2() {
    ...
    B.m3();
    ...
  }
}
```

```
public classe B {
  public static void m3() {
    ...
    C o = new C();
    o.m4();
    ...
  }
}

public classe C {
  public void m4() {
    ...
  }
}
```

Fonte: Franchin (2007, p. 57)

São definidos os pares das menores unidades do teste interclasse, ou seja, os métodos, para a geração dos requisitos de teste Franchin (2007, p. 56). Baseado na estrutura de classes da figura 8, os pares de métodos são:

Tabela 1 - Pares do caso de teste par-a-par

Classe	Par das unidades
A	m1() e m2()
A e B	m2() e m3()
B e C	m3() e construtor C
B e C	m3() e m4()

Fonte: Franchin (2007, p. 57)

O teste par-a-par não é detalhado e aplicado nos capítulos sequentes para cumprir melhor com a organização desta monografia.

4. ESTUDO DE CASO

Este capítulo aplica os testes aprendidos no capítulo anterior em trechos do programa de um software desenvolvido em linguagem Java. Esta aplicação está relacionada somente a um determinado trecho do programa para que o objetivo do trabalho seja completado sem estender de forma desnecessária.

Para realizarmos os testes será utilizado o IDE NetBeans 8.0.2, que é a ferramenta utilizada para o desenvolvimento do software em linguagem Java. E o plug-in integrado do framework de testes JUnit 4.

4.1. FOX (PONTO DE CAIXA)

Este programa foi desenvolvido em linguagem Java. É um sistema de controle de caixa, controla estoque, possui módulo de vendas por código de barras, impressão de cupom e relatórios de movimentações.

É utilizado o banco de dados MySQL. O banco de dados deste software utiliza três tabelas principais: **Produto**, onde é armazenado informações de estoque e valor por quilo; **Venda**, que registra a data, valor total e o usuário do sistema que efetuou a venda; **VendaProduto**, onde é detalhado os produtos vendidos, quantidade e valor, unidos pela chave primária Venda.id.

Na tabela 2 são apresentadas as principais classes do projeto separadas pelo modelo de desenvolvimento MVC (*Model, View, Controller*). A tela principal do sistema é apresentada na figura 9.

Tabela 2 - Estrutura do sistema Fox

Model	Controller	View
Produto	Util	JFrameProduto
Venda		JFrameVendas
VendaProduto		JFrameRelatorio

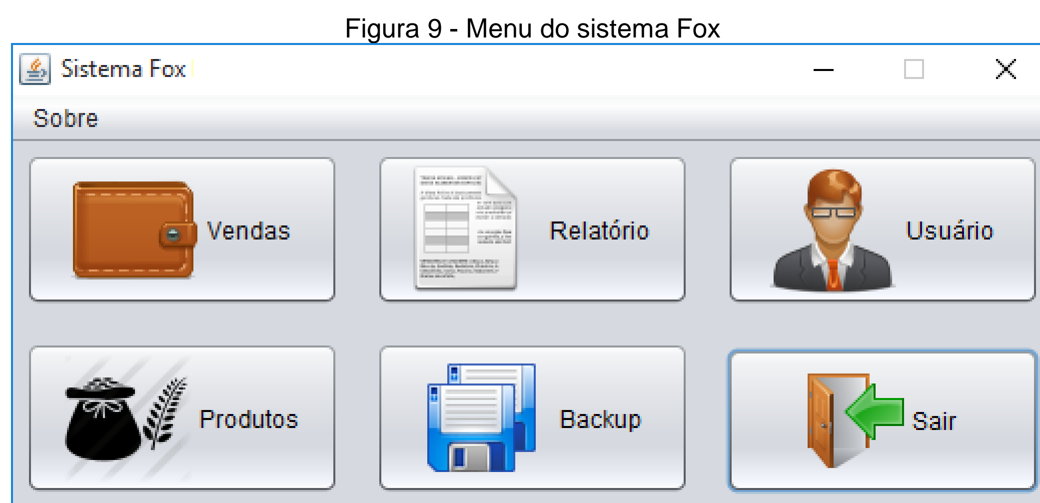
Fonte: Autoria própria.

As classes *Models*são classes de persistência. Ou seja, os seus objetos têm atributos que serão armazenados no banco de dados. Os métodos são responsáveis por visualizar, armazenar e alterar dados dos registros.

A classe Util é responsável pelas regras de negocio, como validações de valores, e pela impressão do cupom não fiscal.

Por último, as classes *Views* representam a interface gráfica do programa. As telas com as caixas de texto, botões e *labels* são tratadas por estas classes.

Figura 9 - Menu do sistema Fox



Fonte: Autoria própria.

4.2. FOX (PONTO DE CAIXA) - TESTES

O JUnit é um framework de teste de unidade em Java que proporciona de forma prática a execução de testes, comparando valores de entrada e saída, trabalhando de forma automática. Ao ser executado, os resultados dos testes são exibidos através de um relatório. O JUnit mostra a origem do erro caso alguma saída inválida, exceção ou erro não esperado ocorrer. (Delamaro, Maldonado e Jino, 2007, p. 171).

É adicionado uma classe de teste JUnit para cada classe testada. Aconselha-se criar um pacote apenas para as classes de teste, como prática de um bom desenvolvimento. O JUnit cria a estrutura de todos os métodos que serão executados da classe selecionada. Ao executar a classe de testes, uma tela de

resultados aparece para indicar e descrever caso algum teste falhe (Devmedia, acesso em 01/03/2016).

Na tabela 3 são descritos alguns dos comandos que são utilizados nas classes de testes.

Tabela 3 - Algumas instruções do JUnit 4

INSTRUÇÃO	DESCRIÇÃO
@before	Indica os métodos que serão executados antes da pilha de testes. Um exemplo seria a inicialização de objetos ou instancia do banco de dados.
@after	Métodos que serão executados depois da pilha de testes, como encerramentos de instancia e arquivos.
@test	Os métodos que o JUnit executará devem ser notados com esta expressão para indicar que ele será testado.
@test(expected=Exception.class)	Para o teste passar, o valor de saída esperado deve ser a excessão descrita no argumento.
@test(timeout=xxx)	Se o tempo de execução do método de teste for maior que o valor do argumento, o JUnit indicará um erro.
assertEquals(objeto esperado, objeto atual, arredonamento)	Sendo um dos comandos mais comuns, assert compara se o resultado de uma execução retornou o valor esperado. O terceiro argumento é opcional e utilizado para arredondamento de valores numéricos.

Fonte: www.junit.org

Para cadastrar um produto no sistema Fox, o usuário precisa fornecer as seguintes informações: código, nome, valor e estoque. O método *btnInserirActionPerformed* apresentado a seguir, é chamado quando o botão *Inserir* é pressionado pelo usuário na tela *Produtos*. O método irá fazer as verificações e validações de texto e números através dos métodos **verificaCodigo(string)** e **verificaMoeda(string)**; irá verificar se não existe nenhum produto com o mesmo código através do método **existeProduto(int)**; irá inserir o produto na base de dados através do método **inserirProduto(Produto)** e por fim fazer a atualização da tabela de produtos que existe na janela e limpar todos os campos através dos métodos **atualizaProdutos()** e **limpar()**. Os dois últimos métodos não serão exercitados neste trabalho por trabalharem diretamente com campos do formulário Java.

Figura 10 - Código do botão Inserir Produto

```
    private void btnInserirActionPerformed(java.awt.event.ActionEvent  evt) {

    //verifica se os campos preenchidos  são válidos
          if ((!Util.verificaCodigo(txtCodigo.getText())) ||txtNome.getText().equals("") ||
             (!Util.verificaMoeda(txtValorUnitario.getText()))  ||  (!Util.verificaMoeda(txtEstoque.getText()))) {
    showMessageDialog("Dados inválidos.");
             return;}

    //verifica se o produto  que será  inserido  não  existe  na  base  de  dados.
    int codigo = Int txtCodigo.getText();
    int resultado = Produto.existeProduto(codigo);

          if (resultado == 1 || resultado == -1) {
    showMessageDialog("Ocorreu uma falha ao inserir este produto");
             return;  }

    //tenta inserir o produto  na base de dados,  atualiza  o grid com os produtos e limpa as caixas de texto
          if (Produto.inserirProduto(new Produto(Int txtCodigo.getText(), Double txtValorUnitario.getText(), Double
txtEstoque.getText(), txtNome.getText())) {
    showMessageDialog("Produto inserido com sucesso.");
    atualizaProdutos();
    limpar();
          } else
    showMessageDialog("Ocorreu uma falha ao inserir este produto.");
       }
```

Fonte: Autoria própria.

Para exercitar o método do botão Inserircom o JUnit, todos os métodos terceiros serão testados. Desta forma exemplifica-se o método de teste estrutural de integração, ou teste intermétodo e interclasse. Primeiramente os métodosverificaCodigo(string)e verificaMoeda(string)da classe Util verificam se o valor passado por parâmetro não é negativo, conforme apresentado a seguir.

Figura 11 - Código dos métodos verificaCodigo() e verificaMoeda()

```
public static boolean verificaCodigo(String pValor) {
      try {
int valor = Int (pValor);
          if (valor < 0) return false;
} catch (Exception e) return false;
return true;
}
```

```
public static boolean verificaMoeda(String pValor) {
     try {
       double valor = Double (pValor);
          if (valor < 0) return false;
     } catch (Exception e) return false;
     return true;
}
```

Fonte: Autoria própria.

Utiliza-se o comando ***assertTrue()*** e ***assertFalse()*** do JUnit 4 para testar o retorno booleano do método verificaCódigo(). O comando é uma variação do assertEquals(), ou seja, o comando mostra após a sua execução se o valor retornado é verdadeiro(*assertTrue*), ou falso (*assertFalse*). Se falhar e o resultado não for o esperado, o JUnit acusa uma falha no seu relatório.

Segundo o teste de domínio (capítulo 2.2), quatro valores de teste são definidos para testar a condição relacional simples do método verificaCodigo(). Para estasituação os valores são 1, 999, -1, -999. O resultado do teste através do JUnit é apresentado na figura 12.

O método verificaMoeda(string) pode ser testado com os mesmos valores de entrada utilizados no teste do método verificaCodigo(). A única diferença entre os dois é que a conversão do valor passado por parâmetro é para Double, no caso do método verificaCodigo() o valor convertido é Integer, conforme apresentado no código acima.

Figura 12 - Testes dos métodos verificaCodigo() e verificaMoeda()

```
47      @Test
48      public void testVerificaCodigo() {
49          System.out.println("Valor inserido: 1");
50          assertTrue(Util.verificaCodigo("1"));
51          System.out.println("Valor inserido: 999");
52          assertTrue(Util.verificaCodigo("999"));
53          System.out.println("Valor inserido: -1");
54          assertFalse(Util.verificaCodigo("-1"));
55          System.out.println("Valor inserido: -999");
56          assertFalse(Util.verificaCodigo("-999"));
57      }
58      @Test
59      public void testVerificaMoeda() {
60          assertTrue(Util.verificaMoeda("1"));
61          assertTrue(Util.verificaMoeda("999"));
62          assertFalse(Util.verificaMoeda("-1"));
63          assertFalse(Util.verificaMoeda("-999"));
64      }
```

FoxPDV.control.UtilTest

Resultados do Teste

FoxPDV.control.UtilTest

100,00 %

Ambos os testes foi(foram) aprovado(s).(0,149 s)

- FoxPDV.control.UtilTest aprovado
  - testVerificaCodigo aprovado (0,005 s)
  - testVerificaMoeda aprovado (0,0 s)

```
Valor inserido: 1
Valor inserido: 999
Valor inserido: -1
Valor inserido: -999
```

Fonte: Autoria própria.

O método existeProduto(int) da classe Model Produto, faz uma pesquisa na base de dados em busca do produto com o código passado por parâmetro e retorna um valor inteiro que representa o resultado. Para testar os retornos inteiros, utiliza-se o comando assertEquals(). As três possibilidades de retorno serão forçadas no teste, sendo o retorno 1 para produto existente, -1 para erro na busca e 0 para produto não existente. O resultado do teste deste método é exibido na figura 14.

Figura 13 - Código do método existeProduto()

```
public static int existeProduto(int pCodigo){
try {
//instancia um objeto de conexão Mysql através da classe MyConnection
        Connection con = MyConnection.getInstance();

//query SQL que procura na base de dados o código passado por parâmetro
String sql = "select * from produto where codigo = " + pCodigo;
        PreparedStatement preparedStmt1 = con.prepareStatement(sql);

//se existir um código retorna 1, se houver erro -1, e se não existir produto com este código, 0.
if (preparedStmt1.executeQuery().next())  return 1;
      } catch (Exception e)  return -1;
      return 0; }
```

Fonte: Autoria própria.

Figura 14 - Teste do método existeProduto()

```
71        @Test
72        public void testExisteProduto() {
73            System.out.println("existeProduto");
74            assertEquals(1, Produto.existeProduto(400));
75            assertEquals(0, Produto.existeProduto(515));
76            //assertEquals(-1, Produto.existeProduto(1));
77        }
78    }
```

FoxPDV.model.ProdutoTest > testExisteProduto >

Resultados do Teste

FoxPDV.control.UtilTest | FoxPDV.model.ProdutoTest

100,00 %

O teste foi(foram) aprovado(s).(0,487 s)
FoxPDV.model.ProdutoTest aprovado
testExisteProduto aprovado (0,41 s)

existeProduto

Fonte: Autoria própria.

Na figura 14, o valor 400 é o código de um produto já cadastrado na base de dados, logo o retorno foi 1, conforme esperado; o valor 515 não existe produto correspondente, e o seu retorno foi 0. A linha comentada de número 76 representa a asserção do retorno -1, que representa uma falha na busca do código. Para forçar a exceção e o retorno -1 é necessário parar a execução do servidor do banco de dados. Observa-se na figura 15 que o tempo do teste foi alto, 4,138 segundos, o que indica que o sistema aguardou uma resposta do servidor e não obteve sucesso.

Figura 15 - Teste do método existeProduto() com servidor parado

```
71      @Test
72      public void testExisteProduto() {
73          System.out.println("existeProduto. Teste com o Apache parado.");
74          //assertEquals(1, Produto.existeProduto(400));
75          //assertEquals(0, Produto.existeProduto(515));
76          assertEquals(-1, Produto.existeProduto(1));
77      }
78  }
```

FoxPDV.model.ProdutoTest › testGetProdutosByNome ›

Resultados do Teste

FoxPDV.control.UtilTest | FoxPDV.model.ProdutoTest

100,00 %

O teste foi(foram) aprovado(s).(4,241 s)

FoxPDV.model.ProdutoTest aprovado

testExisteProduto aprovado (4,138 s)

existeProduto. Teste com o Apache parado.

Fonte: Autoria própria.

O método inserirProduto(Produto) da classe modelo Produto, recebe como parâmetro um objeto com os atributos código, nome, valor e estoque, cria uma instancia de conexão com o banco de dados, monta a query SQL com os atributos do objeto passado por parâmetro e tenta executá-la. As validações são feitas através de outros métodos já apresentados acima, devido a isso o novo produto só será inserido no banco se as validações estiverem corretas. Para testar o método inserirProduto() apenas um retorno com um produto verdadeiro é exercitado, conforme apresentado no relatório do JUnit na figura 17.

Figura 16 - Código do método inserirProduto()

```
public static boolean inserirProduto(Produto pProduto){
       Connection con = MyConnection.getInstance();
try {

//monta a query SQL com os valores do objeto passado por parâmetro
         String sql = "insert into produto values (' pProduto.getCodigo() ', ' pProduto.getNome() ',
'pProduto.getValorUnitario() ', ' pProduto.getEstoque() ')";
PreparedStatement preparedStmt = con.prepareStatement(sql);

//tenta executar o comando SQL, se houver erro retorna false, se não, true
preparedStmt.executeUpdate();
       } catch (Exception e) return false;
return true;
   }
```

Fonte: Autoria própria.

Figura 17 - Teste do método inserirProduto()

```
19  public class ProdutoTest {
20      //                       codigo, valor unitário, estoque, nome
21      Produto produto = new Produto(515, 7, 45, "Macarrão instantâneo");
22
23      public ProdutoTest() {
24      }
25
26      @Test
27      public void testinserirProduto() {
28          System.out.println("Teste inserirProduto");
29          assertTrue(Produto.inserirProduto(produto));
30      }
```

```
Resultados do Teste
FoxPDV.model.ProdutoTest
100,00 %
O teste foi(foram) aprovado(s).(0,781 s)
FoxPDV.model.ProdutoTest aprovado
    testinserirProduto aprovado (0,641 s)

Teste inserirProduto
```

Fonte: Autoria própria.

Como o método utiliza um objeto da classe Produto como parâmetro, primeiramente deve-se criar um objeto na classe teste do Junit, assim como apresentado na linha 21 da figura 17. O objeto é criado com parâmetros válidos para um produto que ainda não existe na base dados: código do produto 515, valor unitário R$ 7,00, estoque disponível 45 e nome do produto “Macarrão instantâneo”.

O teste foi aprovado com o comando *assertTrue* do JUnit, o que indica que o retorno booleano do método inserirProduto() foi verdadeiro e o novo produto foi inserido com sucesso na base da dados do sistema.

5. CONSIDERAÇÕES FINAIS

Buscou-se responder a problemática desta pesquisa (como garantir a qualidade dos sistemas computacionais, através de funções em nível de código?) através da apresentação de técnicas de teste de caixa branca e da demonstração do teste na prática fazendo uso de um framework Junit com Java.

A aplicação de testes em um software comercial real apresentados no capítulo 4 serviu para satisfazer o objetivo geral da pesquisa inicialmente proposto. Os objetivos específicos que nortearam este trabalho se fazem presentes na fundamentação teórica realizada no capítulo 2, onde é descrito as fundamentações e os motivos da importância do teste de software; e no terceiro capítulo, no qual se busca aprender diferentes tipos de testes.

A fundamentação teórica mostrou que os testes servem tanto para procurar, identificar e corrigir erros, como para validar se a necessidade do cliente está de acordo com o que foi desenvolvido, baseando-se nos requisitos, na fase de validação e verificação.

Os principais autores referenciados na pesquisa possuem obras completas sobre engenharia de software e seus ensinamentos são amplamente utilizados nos trabalhos. Autores como Roger S. Pressman e Ian Sommerville foram essenciais para o desenvolvimento da temática, teste de software, deste trabalho.

A metodologia utilizada para cumprir os objetivos da pesquisa foi planejada para ser apresentada de forma lógica e estruturada, devido a isso o conhecimento pode ser bem aproveitado para fins didáticos. A organização foi a seguinte: fundamentação teórica, especificação do tema e aplicação prática.

Os testes estruturais junto com o uso da ferramenta JUnit 4, se provaram extremamente necessários na prática de desenvolvimento de software, graças a eles o desenvolvedor pode aumentar a qualidade do seu programa, diminuir as chances

de falhas após a implantação do sistema e consequentemente a satisfação do cliente com o resultado final.

O *framework* utilizado para a aplicação dos testes no código do sistema Fox revelou sua simplicidade e praticidade no manuseio. Devido ao fato dele oferecer suporte para os principais ambientes de desenvolvimento Java, pode ser utilizado por desenvolvedores em diferentes plataformas, conforme suas preferências. É extremamente importante o uso do framework desde o início do desenvolvimento até sua conclusão, e ainda nas possíveis manutenções.

Além do relatório de retornos do JUnit que mostram se os testes foram executados com sucesso, a ferramenta também mostra o tempo exato de processamento do método. Graças a isso o desenvolvedor pode otimizar o código se perceber que algo esteja atrasando a execução do método, fazendo com que o sistema fique mais enxuto e ágil no tempo de resposta já que muitas vezes os comandos são aninhados e executados quase que simultaneamente.

Alguns itens que dariam um estudo complementar seria o aprofundamento de normas e documentação para testes de software e a utilização do PMBOK para gerenciamento dos projetos de testes de software.

REFERÊNCIAS

ALEXANDER, R. T.; OFFUTT, J. Coupling-based Testing of O-O Programs. J.UCS, v. 10, n. 4, 2004. Disponível em: < http://www.jucs.org/jucs_10_4/coupling_based_testing_of/Alexander_R_T.pdf> Acesso em 09/2015.

BLACK, R. The Cost of Software Quality. Disponível em: <http://www.stickyminds.com>, consultado em 04/05/2015.

BRUNELI, MARCOS V. Q. A utilização de uma metodologia de teste no processo da melhoria da qualidade do software. Campinas – SP: Universidade Estadual de Campinas, 2006.

DELAMARO, MÁRCIO E.; MALDONADO, JOSÉ C.; JINO, MARIO.Introdução ao TESTE DE SOFTWARE. Rio de Janeiro: Elsevier, 2007.

DEVMEDIA – Tutoriais, vídeios e cursos de programação. Disponível em <https://www.devmedia.com.br/>, consultado em 01/03/2016.

FRANCHIN, IVAN G.Teste estrutural de integração par-a-par de programas orientados a objetos e a aspectos: critérios e automatização. São Carlos – SP: ICMC-USP, 2007.

HARROLD, MARY JEAN; ROTHERMEL, GREGG, Performing data flow testing on classes. Clemson, SC-USA: Clemson University, 1994.

LEWIS, WILLIAM E. Software testing and continuous quality improvement. 2ª ed. Boca Raton, Florida: CRC Press LLC, 2005.

PAULAFILHO, WILSON P. Engenharia de Software Fundamentos, Métodos e Padrões.3ª ed. Rio de Janeiro – RJ: LTC, 2009.

PRESSMAN, ROGER S. Engenharia de Software. 7ª ed. São Paulo: AMGH Editora ltda, 2011.

RIOS, EMERSON; MOREIRA, TRAYAHÚ. Teste de Software. 2ª ed. Castelo Rio de Janeiro – RJ: Alta Books ltda, 2006.

SOFIST, INTELLIGENT SOFTWARE TESTING. Disponível em: <http://www.sofist.com.br/> acesso em: 16 abril 2016.

SOMMERVILLE, IAN. Engenharia de Software. 8ª Ed. São Paulo: Pearson Addison – Wesley, 2007.