CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TESTE DE SOFTWARE: TÉCNICAS DE TESTE ESTRUTURAL E ANÁLISE DE CÓDIGO

MATHEUS ANTONIO DA ASSUNÇÃO

Americana, SP
2015

CENTRO PAULA SOUZA

GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TESTE DE SOFTWARE: TÉCNICAS DE TESTE ESTRUTURAL E ANÁLISE DE CÓDIGO

MATHEUS ANTONIO DA ASSUNÇÃO
matheus.tba@hotmail.com

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do curso superior de tecnologia em Análise e Desenvolvimento de Sistemas da Fatec - Americana, sob orientação do Prof. e Mestre Anderson Luiz Barbosa.

Área: Tecnologia da Informação

Americana, SP
2015

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

Assunção, Matheus Antonio da

A711t Teste de software: técnicas de teste estrutural e análise de código. / Matheus Antonio da Assunção. – Americana: 2015.
44f.

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. Anderson Luiz Barbosa

1. Desenvolvimento de software I. Barbosa, Anderson LuizII. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.3.05

Matheus Antonio da Assunção

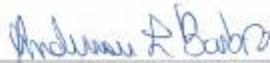
Teste de software:

Técnicas de teste estrutural e análise de código

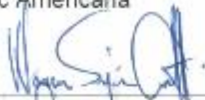
Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Engenharia de software.

Americana, 07 de Dezembro de 2015.

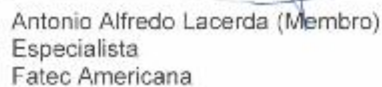
Banca Examinadora:



Anderson Luiz Barbosa (Presidente)
Mestre
Fatec Americana



Wagner Siqueira Cavalcante (Membro)
Mestre
Fatec Americana



Antonio Alfredo Lacerda (Membro)
Especialista
Fatec Americana

AGRADECIMENTOS

Gostaria de demonstrar em algumas palavras minha gratidão ao governo do estado de São Paulo pela oportunidade de estudo (méritos a mim, por ter passado no vestibular) na Faculdade de Tecnologia de Americana. Com toda certeza o conhecimento adquirido aqui irá abrir muitas portas na minha vida profissional.

Agradeço aos meus pais, Jeremias e Elaine, que nunca deixaram de me apoiar, seja qual fosse minha escolha em fases que eu quase desisti de coisas importantes.

Aos meus amigos que tive o prazer de conhecer na faculdade, por quem tenho muito respeito e admiração.

Aos colegas de trabalho que me incentivaram a terminar a monografia.

Aos excelentes professores que estavam sempre dispostos a ensinar, mesmo além do horário definido na grade de matérias.

Aos meus orientadores Alexandre Melo e Anderson Luiz Barbosa, pela paciência de me acompanhar durante os semestres que passei desenvolvendo este trabalho.

Sinto-me lisonjeado e muito agradecido, obrigado!

SUMÁRIO

LISTA DE FIGURAS I

LISTA DE TABELAS II

RESUMO III

ABSTRACT IV

INTRODUÇÃO V

1. FUNDAMENTAÇÃO TEÓRICA: TESTE DE SOFTWARE 1

1.1. Sobre software 1
1.2. Verificação e Validação 2
1.3. Finalidade dos testes de software 3
1.4. Tipos de teste 5
1.4.1. Testes de caixa preta 6
1.4.2. Testes de caixa branca 8
1.4.3. Testes de caixa cinza 10
1.5. Documentação de teste 11
1.6. Norma IEEE 829/1998 12

2. TESTE ESTRUTURAL DE SOFTWARE 14

2.1. Teste de caminho básico 14
2.2. Teste de condição 16
2.3. Teste de fluxo de dados 17
2.4. Teste de Laços 19
2.5. Teste de unidade em programas OO 21
2.6. Teste de integração em programas OO 22
2.7. Teste par-a-par 22

3. ESTUDO DE CASO 24

3.1. Fox PDV (Ponto de caixa) 24

3.2. JUnit 4 27

4. CONSIDERAÇÕES FINAIS..34

REFERÊNCIAS...37

LISTA DE FIGURAS

Figura 1 - Custo e qualidade. 4
Figura 2 - Fases de teste....6
Figura 3 - Teste intramétodo e intermétodo em programação orientada a objeto. 9
Figura 4 - Conceitos de menor unidade em testes estruturais OO....10
Figura 5 - Exemplo de grafo de fluxo. 15
Figura 6 - Função main do programa Identifier. 18
Figura 7 - Grafo Definição-Uso....19
Figura 8 - Laços. 20
Figura 9 - Classe SymbolTable. 21
Figura 10 - Métodos públicos entre classes diferentes. 23
Figura 11 - Menu principal do sistema Fox PDV 27
Figura 12 - Teste do método verificaCodigo() e verificaMoeda()....30
Figura 13 - Teste do método existeProduto() 31
Figura 14 - Teste do método existeProduto() com o servidor Apache parado 32
Figura 15 - Teste do método inserirProduto()....33

LISTA DE TABELAS

Tabela 1 - Custo de investimento. 5
Tabela 2 - Documentos da norma IEEE 829 de 1998. 12
Tabela 3 - Pares para geração do caso de teste par-a-par.......................................23
Tabela 4 - Estrutura principal do sistema Fox PDV...26
Tabela 5 - Principais instruções do JUnit 4. 28

RESUMO

Devido a necessidade de aumentar a qualidade dos sistemas de software, os testes estruturais (realizados diretamente no código do programa), são executados na fase da codificação pelos próprios desenvolvedores. Defeitos que poderiam ser encontrados posteriormente pela equipe de testes são identificados e resolvidos sem que haja perda de tempo e aumento de custo do projeto. Para tratar o tema, a pesquisa bibliográfica é realizada qualitativamente, onde se aborda conceitos de teste de software: a sua importância, suas variações, a formalização através de documentações e normas técnicas e as fases que são aplicadas dentro do projeto. As formas de abordagem dos testes estruturais são apresentadas com exemplos de códigos e grafos, como testes de caminho básico, fluxo de dados, condições, laços de repetição e testes estruturais adaptados à programação orientada a objetos. Para aplicar os conceitos apresentados de forma prática, utiliza-se da ferramenta de desenvolvimento JUnit 4 e da linguagem Java, com trechos de código comentado de um programa real, assim como imagens diretamente do IDE NetBeans (ambiente de desenvolvimento) mostrando o resultado dos testes. A pesquisa teórica e prática tem o objetivo de frisar que testes de software devem ser realizados desde o inicio de um projeto de desenvolvimento, evitando assim problemas na fase de manutenção, onde os custos são multiplicados.

Palavras Chave: Teste de software, Teste estrutural, Caixa branca, JUnit 4, Java, Codificação, Desenvolvimento.

ABSTRACT

Due to need to increase the quality of software systems, the structural testing (performed in the program code), are executed in the coding step by developers. Bugs that can found later by testing team are identified and solved without lost of time and increased project cost. About the monograph theme, the bibliographical research is conducted qualitatively, that approach software testing concepts: the importance, variances of test types, the formalization through documentation, technical standards and your project steps. The approach forms of structural testing are introdruced with codes and graph examples, like basis path testing, data flow, conditions, loops and structural testing adapted for object-oriented programming. To apply the concepts in a practical way, is used JUnit 4 framework and the program language Java, with commented code parts on real software system, and images directly from the IDE NetBeans (development environment) showing the test results. The theoretical research and practical research are designed to prove software testing should be performed from the development beginning, avoiding problems in maintenance phase, where project costs are multiplied.

Keywords: Software testing, structural testing, white-box, JUnit 4, Java, Coding, Development.

INTRODUÇÃO

As empresas de tecnologia e desenvolvimento de sistemas, na atualidade, têm se preocupado muito em relação à qualidade do produto. A demanda do mercado e a evolução nos processos empresariais necessitam que os sistemas de software sejam cada vez mais eficientes.

Essa preocupação faz com que haja um crescente investimento de tempo e dinheiro em atividades de testes de software. Este, que procura garantir o bom funcionamento do sistema (conforme os seus requisitos) através de técnicas desenvolvidas especificamente para apontar e revelar defeitos no tratamento de dados do programa.

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 (PRESSMAN, 2011).

A falta da devida atenção aos testes por parte dos desenvolvedores e testadores pode comprometer a qualidade do sistema e causar atrasos no prazo de entrega, aumentando assim os custos do projeto. Segundo Rios e Moreira (2006) informações de mercado dizem que mais de 90% dos sistemas são liberados com graves defeitos. Softwares com problemas de performance e com defeitos na execução são custosos. Os custos para identificar e corrigir esses problemas podem ser de 100 a 1.000 vezes maiores do que se fossem realizados logo após a sua introdução.

Para tanto o estudo se justificou pela necessidade de apresentar métodos de teste estrutural aos acadêmicos interessados em aprimorar seus conhecimentos em teste e desenvolvimento de software. De acordo com SOMMERVILLE (2007) o teste estrutural é uma abordagem para projetar casos de teste na qual os testes são derivados do conhecimento da estrutura e da implementação do software. Essa abordagem é, algumas vezes, chamada de teste ‘caixa-branca’, teste ‘caixa-de-vidro’ ou teste ‘caixa-clara’ para distingui-lo do teste caixa-preta.

A pergunta que se buscou responder foi: como garantir que funções em nível de código sejam validadas pela qualidade de software buscando minimizar o custo de manutenção de defeitos após a entrega do produto?

O objetivo geral consistiu em analisar técnicas de teste estrutural buscando a qualidade segundo a engenharia de software. Os objetivos específicos foram: a) realizar uma pesquisa biográfica sobre testes de software, com o objetivo de selecionar diferentes técnicas de teste. b) desenvolver casos de teste caixa branca para um sistema de controle de venda e estoque real c) aplicar os testes e avaliar os melhores métodos para o problema proposto.

Em sua natureza, o desenvolvimento deste trabalho foi através da metodologia 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. O primeiro conceitua testes de software como requisito básico para a qualidade final do sistema como um todo; o segundo busca apresentar técnicas de teste estrutural aplicando-os e documentando-os em casos de teste para que, no terceiro capítulo, sejam aplicados em um sistema real e avaliados.

O quarto capítulo é destinado às considerações finais com base no estudo apresentado nos primeiros capítulos.

1. FUNDAMENTAÇÃO TEÓRICA: TESTE DE SOFTWARE

Este trabalho tem por objetivo aplicar testes estruturais em um sistema de software a fim de relatar e comparar diferentes técnicas e utilidades para cada situação.

O primeiro capítulo deste estudo conceitua teste de software para uma melhor compreensão do leitor, facilitando o entendimento dos capítulos consequentes.

1.1. SOBRE SOFTWARE

Segundo PRESSMAN (2011, p.32) uma das definições para software é o conjunto de instruções de um programa de computador que, quando executadas, produzem a função e o desempenho desejados. Também pode ser definido como estruturas de dados que possibilitam que os programas manipulem adequadamente a informação.

Muitas pessoas associam o termo software aos programas de computador. Na verdade, essa é uma visão muito restritiva. Software não é apenas um programa, mas também todos os dados de documentação e configuração associados, necessários para que o programa opere corretamente. (SOMMERVILLE, 2007, p. 4)

Um software é criado através de um processo, que segundo o IEEE[1], que PAULA FILHO (2009, p. 89) cita em seu livro é uma “sequencia de passos executados com um determinado objetivo”.

SOMMERVILLE (2007, p. 6) escreve que um processo de software é um conjunto de atividades e resultados associados que produz um produto de software. Existem quatro atividades fundamentais que são comuns a todos os processos de software: (1) Especificação de software, onde os engenheiros e os clientes definem as especificações do que será produzido, assim como suas restrições; o (2) Desenvolvimento de software é o processo no qual o produto é projetado e programado; em seguida são feitas as (3) Validações de software, na qual o

---

[1] *Sigla IEEE: Institute of Electrical and Eletronics Engineers.*

software é verificado para garantir que é o que o cliente deseja; e por fim a (4) Evolução de software, quando o software é modificado ou adaptado para atender às mudanças dos requisitos do cliente e do mercado.

Definitivamente, a construção de software não é uma tarefa simples. Pelo contrário, pode se tornar bastante complexa, dependendo das características e dimensões do sistema a ser criado. Por isso, está sujeita a diversos tipos de problemas que acabam resultando na obtenção de um produto diferente daquele que se esperava (DELAMARO, MALDONADO e JINO, 2007, p. 13).

Para que tais “problemas” não perdurem, ou seja, para serem descobertos antes de o software ser liberado para utilização, existe uma série de atividades, coletivamente chamadas de “Validação, Verificação e Teste”, com finalidade de garantir que tanto o modo pelo qual o software esteja sendo construído quanto o produto em si estejam em conformidade com o especificado (DELAMARO, MALDONADO e JINO, 2007, p. 1).

1.2. VERIFICAÇÃO E VALIDAÇÃO

Segundo SOMMERVILLE (2007, p. 52), o terceiro processo de software é o chamado de Verificação e Validação (V & V). Neste processo destina-se mostrar que um sistema está em conformidade com sua especificação e que atende às expectativas do cliente que está adquirindo o sistema.

PRESSMAN (2011, p. 402) diz que a verificação refere-se ao conjunto de atividades que garante que o software implemente corretamente uma função específica. A validação refere-se a um conjunto diferente de atividades que garante que o software que foi construído é rastreável às exigências do cliente.

Para ilustrar:

Verificação: verificar se o módulo de cadastro de clientes está funcionando corretamente.

Validação: o módulo de cadastro de clientes está de acordo com o que o cliente precisa?

PRESSMAN (2011, p. 402) cita que a verificação e validação engloba uma série de atividades de SQA (*Software Quality Assurance*, em português Garantia de Qualidade de Software) nas quais existem revisões técnicas formais, auditorias de configuração e qualidade, monitoração do desempenho, simulação, estudo de viabilidade, revisão da documentação, revisão de bancos de dados, análise de algoritmos, teste de desenvolvimento, teste de qualificação e teste de instalação.

Os testes de software são entendidos basicamente em dois tipos ao longo do processo de software: SOMMERVILLE (2007, p. 343) **Teste de validação** tem a finalidade de mostrar que o software é o que o cliente deseja; **Teste de defeitos** é destinado a revelar defeitos no sistema em vez de simular o seu uso operacional. O objetivo do teste de defeitos é encontrar inconsistências entre um programa e sua especificação.

Este estudo aborda teste de software como uma atividade contida em testes de defeitos, segundo a engenharia de software.

1.3. FINALIDADE DOS TESTES DE SOFTWARE

Os testes, mais do que meios de detecção e correção de erros, são indicadores da qualidade do produto. Quanto maior o número de defeitos detectados, provavelmente maior o número de defeitos não-detectados. A ocorrência de um número anormal de defeitos em testes indica a necessidade de redesenho dos itens testados (PAULA FILHO, 2009, p. 349).

O autor PRESSMAN (2007, p. 788 apud Glen Myers, 1979) 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;

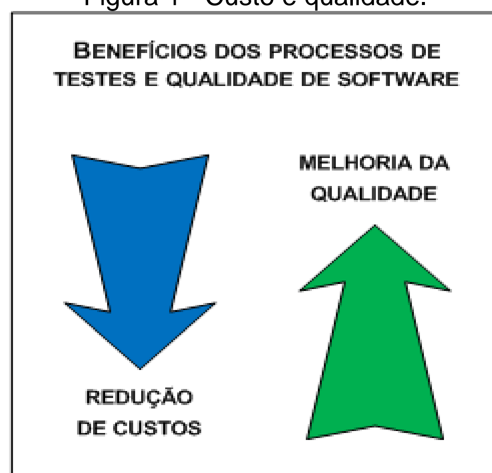
3. Um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

Tais objetivos acima apontam inversamente ao ponto de vista de que testes de software bem-sucedidos são os quais passam sem nenhum ou com poucos erros. A finalidade de um caso de testes é investigar os erros do programa sistematicamente, no menor tempo e esforço possível (PRESSMAN, 2011).

Um objetivo central de toda a metodologia dos testes é maximizar a sua cobertura, ou seja, a quantidade potencial de defeitos que podem ser detectados por meio do teste. Deseja-se conseguir detectar a maior quantidade possível de defeitos que não foram apanhados pelas revisões, dentro dos limites de custo e prazos (PAULA FILHO, 2009, p. 351).

Os testes permitem avaliar se o software desenvolvido atende os requisitos para ele especificados ainda durante o desenvolvimento, reduzindo significativamente os custos de correção após a implantação e durante a manutenção do sistema. Dependendo da aplicação, defeitos encontrados após a implantação podem custar de 10 a 10.000 vezes mais que defeitos encontrados durante o desenvolvimento. A figura 1 ilustra o fato de que a qualidade é inversamente proporcional a redução de custos (em: <http://www.sofist.com.br/solucoes/testes-de-software> acesso em: 16 abril 2015).

Figura 1 - Custo e qualidade.



Fonte: <http://www.sofist.com.br/>

Com relação à redução de custos, Black (2002, p. 3) em seu artigo sobre o custo da qualidade faz um estudo hipotético comparando o valor do investimento de uma empresa desenvolvedora de software utilizando técnicas formais e não formais, conforme consta na tabela 1.

Tabela 1 - Custo de investimento.

	Sem metodologia formal de teste	**Com metodologia formal de teste**
Pessoal, infra-estrutura e ferramentas	$ 70,000	$ 82,500
Erros encontrados e corrigidos durante a fase de desenvolvimento	$ 2,500	$ 2,500
Erros encontrados e corrigidos pela equipe de testes	$ 35,000	$ 50,000
Erros encontrados pelo cliente após a entrega	$ 400,000	$ 250,000
Custo total do investimento	**$ 507,500**	**$ 385,000**
Retorno do investimento	350%	445%

Fonte: <http://www.stickyminds.com/>

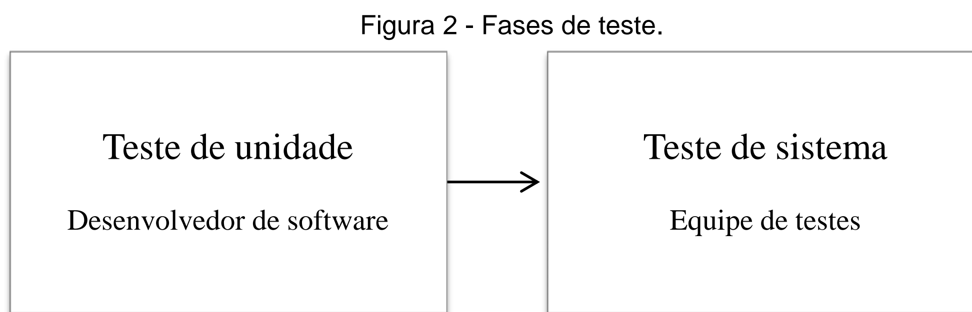
Como apresentado na hipótese acima os custos com a não utilização de uma metodologia formal de testes mostram-se superiores. Este valor fica ainda mais alto quando não se utiliza procedimentos e esforço unicamente dedicados aos testes.

1.4. TIPOS DE TESTE

Segundo SOMMERVILLE (2007), todos os tipos de testes se agrupam em duas modalidades fundamentais, os testes de sistema, onde os requisitos do negócio são colocados a prova no software como um todo e os testes de componentes (ou testes de unidade), no qual são testados componentes distintos separadamente. Estes testes geralmente são feitos pelos próprios desenvolvedores em conjunto com a codificação.

O objetivo do teste de unidade é descobrir defeitos através de testes de partes individuais do programa, que podem ser funções, trechos de códigos, métodos e componentes de software reutilizáveis. No teste de sistema estes componentes são unificados e é feito uma verificação para validar se o sistema atende aos requisitos funcionais, como entrada e saída de dados, registro no banco de dados, telas e interface. Muitas vezes os defeitos que passam despercebidos nos testes de unidade são identificados no teste de sistema. A figura 2 ilustra em etapas e responsabilidades o teste de unidade e teste de sistema.

Figura 2 - Fases de teste.



Fonte: SOMMERVILLE (2007)

Para generalizar, testes de sistema também podem ser chamados de testes de caixa-preta e testes de unidade referidos como testes de caixa-branca.

1.4.1. 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. Ele permite que o engenheiro de software crie conjuntos de entrada de dados para que os requisitos funcionais como, por exemplo, o cadastramento de dados validado por caracteres específicos seja planejado em um caso de teste. 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**, no qual o sistema é colocado em carga máxima para verificar seu comportamento e tratamento de erros, de forma que não cause inconsistência de dados ou travamento total do sistema; **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 testado 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. Os testes neste caso tem por objetivo comparar os resultados de ambos, normalmente feitos através de ferramentas automatizadas; as **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.

1.4.2. Testes de caixa branca

"O teste de caixa branca é uma filosofia que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste" (PRESSMAN, 2011, p. 431).

Os métodos de caixa branca (*White-box*) permitem que o engenheiro de software desenvolva casos de teste que, segundo PRESSMAN (2011, p. 431) asseguram que os **caminhos independentes** dentro de um determinado módulo de código de software 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.

Entretanto, segundo DELAMARO, MALDONADO e JINO (2007) o teste baseado nestes critérios mostra-se pouco eficaz devido ao fato de que os programas, mesmo pequenos, podem possuir um número de caminhos lógicos infinitos ou extremamente grandes em ocorrência das estruturas de iteração entre classes, componentes ou métodos.

Por outro lado, o autor contrapõe essa desvantagem alegando que a técnica estrutural é vista como complementar às demais técnicas de teste, pois revela classes de defeitos diferentes e indetectáveis por outros padrões de teste. Os métodos de caixa branca podem ser incorporados na fase de testes funcionais e principalmente no decorrer no desenvolvimento da codificação do software.

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

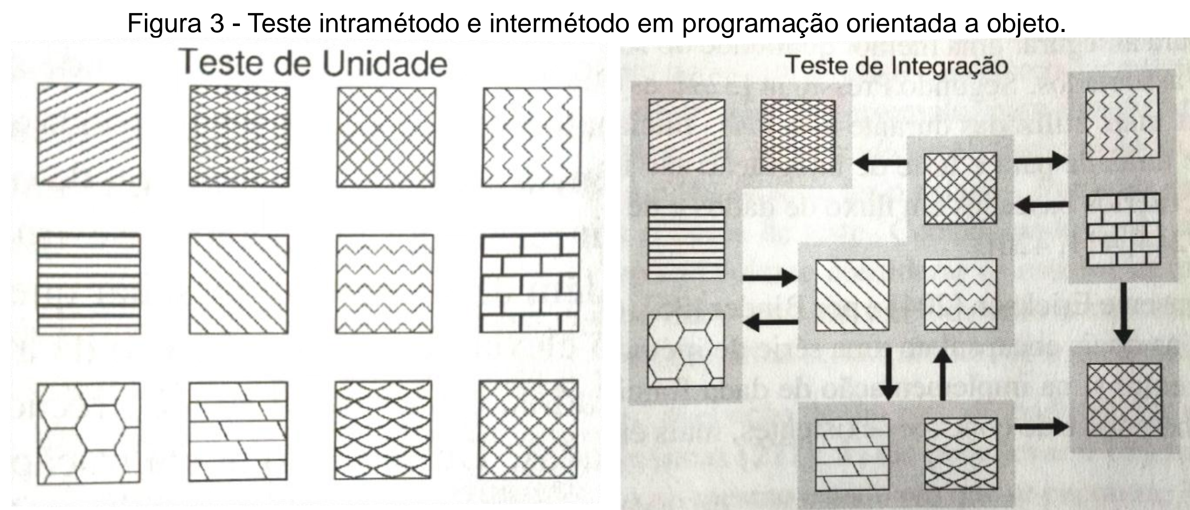
1.4.2.1. Caixa branca em POO

O teste estrutural orientado a objetos pode ser dividido em duas fases: teste de unidade e teste de integração (FRANCHIN, 2007, p. 34).

O teste de unidade em programas procedimentais, também é referido como teste **intrametodo** em OO, trata-se de exercitar a menor unidade de um software: os métodos individualmente. No teste **intermétodo**, equivalente ao teste de integração, são realizadas interações entre métodos diretamente ou indiretamente dentro de uma mesma classe. A finalidade é encontrar chamadas inválidas que causam inconsistência no objeto. Assim como o teste **intraclasse**, a diferença é que neste lida-se chamadas de métodos públicos dentro de uma classe. O teste **interclasse** se ocupa em verificar o funcionamento da relação de polimorfismo, herança e acoplamento dinâmico entre classes. Uma das técnicas para o teste interclasse é a **par-a-par**.

A figura 3 exemplifica unidade e integração representando em quadros os métodos e as classes nas sombras.

Figura 3 - Teste intramétodo e intermétodo em programação orientada a objeto.



Fonte: DELAMARO, MALDONADO e JINO (2007, p. 130)

Diferentes autores utilizam conceitos distintos 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 4 ilustra a diferença entre estes dois pontos de vista.

Figura 4 - Conceitos de menor unidade em testes estruturais OO.

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)

1.4.3. Testes de caixa cinza

Os testes de caixa preta concentram-se na funcionalidade do programa, ou seja, na correta execução e tratamento de dados de entrada e saída. Os de caixa branca focam os caminhos lógicos na estrutura de controle, como instruções, chamadas de funções, parâmetros, laços de repetição, etc. Já os testes de caixa cinza (*Gray-box*) é uma combinação das duas abordagens e é feito pela equipe de testes na fase dos testes funcionais.

O testador deve estudar os requisitos do sistema e conversar com o desenvolvedor para aprender sobre a estrutura interna do sistema. Esta conversa é um compromisso entre os dois e serve para esclarecer funcionalidades ambíguas, tratamento de dados nas diversas telas ou módulos do programa e o armazenamento de informações nas “entrelinhas”. LEWIS (2005) ainda cita dois exemplos práticos nos quais o teste de caixa cinza se mostra útil:

1. Pode ocorrer de existir uma funcionalidade que seja reutilizada em diferentes aplicações do mesmo sistema, neste caso, se o testador se comunica com o desenvolvedor para entender sobre a arquitetura e o design interno do programa, o teste pode ser executado apenas uma vez, fazendo com que muito trabalho repetido seja poupado.

2. Caso exista um comando composto por vários parâmetros, fica inviável testar todas as combinações possíveis. Para 6 parâmetros, um testador teria que desenvolver combinações de 6 formas possíveis (matematicamente na notação 6!), que seria 720 entradas distintas. O problema é agravado se existir parâmetros opcionais O método caixa cinza mostra que através da parceria entre o desenvolvedor e o testador, estes problemas são resolvidos analisando o algoritmo da aplicação. Para este exemplo, se cada um dos seis campos é independente, apenas 6 testes são requeridos para cada parâmetro.

1.5. DOCUMENTAÇÃO DE TESTE

SOMMERVILLE (2007, p. 344) diz que um planejamento de teste cuidadoso serve para, além de abranger o máximo de cobertura em testes e controlar custos com o processo de V&V, também minimizar as chances de ocorrer manutenções posteriores a implantação, onde os custos de reparo são muito mais elevados.

Quanto mais crítico o sistema que está sendo desenvolvido, maior o esforço que deve ser alocado no processo de teste. Este esforço pode ser ajustado dependendo do tipo do sistema, do cronograma de desenvolvimento e da habilidade da equipe ou organização que realizará os testes.

O planejamento de teste tem por finalidade estabelecer padrões para serem seguidos de forma organizada. Também ajudam os gerentes e engenheiros de teste a estabelecer cronogramas de tempo, recursos necessários (mão de obra, equipe, hardware, sistemas, material), projeção dos testes e apoio aos testadores na execução dos mesmos. Para sistemas pequenos, um plano de testes menos formal pode ser usado mais eficientemente.

Planos de testes são documentos que se alteram durante o desenvolvimento. Fatores como o cronograma, atrasos, imprevistos, mudanças na equipe e ordem de tarefas são pontos cruciais que devem ser acompanhados e revisados na documentação periodicamente ou sempre que ocorrerem.

Quanto antes se iniciar as atividades de teste, menos problemas com defeitos o software tende a ter no final do projeto. Uma metodologia de testes é essencial para auxiliar a gerência no planejamento, projeto e realização dos testes, evitando assim um erro muito comum nos projetos de desenvolvimento, que é só começar o planejamento de testes após o término da fase de codificação do produto (BRUNELI, 2006, p. 24).

1.6. NORMA IEEE 829/1998

A norma IEEE 829 de 1998 descreve uma série de documentos que orientam na criação da documentação do teste de software, desde o seu planejamento até a validação dos resultados (BRUNELI, 2006, p. 21).

Ela é dividida em três partes, fase de planejamento, fase de especificação e relato dos resultados; e subdividida em oito documentos, possibilitando que a equipe de testes adapte conforme a sua necessidade e complexidade do projeto, resumindo itens, agrupando documentos e utilizando o método menos ou mais formalizado (BRUNELI, 2006, p. 22).

A nome e a descrição dos documentos de cada uma das três etapas mencionadas pela norma é apresentada na tabela 2.

Tabela 2 - Documentos da norma IEEE 829 de 1998.

	Nome do documento	**Conteúdo do documento**
Fase de planejamento	Plano de teste	Identifica as funcionalidades que serão testadas no sistema, um cronograma com as datas, pessoas envolvidas e risco
Fase de especificação	Projeto de teste	Define os critérios de aprovação e especifica as funcionalidades descritas no Plano de teste, assim como identifica os casos e procedimentos de teste
	Caso de teste	Descreve os dados de entrada, resultados esperados, ações e as condições para a execução dos testes

	Procedimento de teste	Especifica a ordem e etapas para a execução dos casos de teste
Fase de relato	Diário de teste	Relatório das ações de teste cronológicas que são relevantes para o projeto
	Incidente de teste	Relatório de eventos que ocorrem no projeto de testes e que precisam ser revisados posteriormente
	Resumo de teste	Provê avaliações do resultado dos testes em um relatório resumido baseado nas funcionalidades do sistema.
	Encaminhamento de item de teste	Caso as equipes de desenvolvimento e testes forem diferentes, relata o encaminhamento dos itens de teste

Fonte: BRUNELI (2006)

2. TESTE ESTRUTURAL DE SOFTWARE

Neste capítulo o estudo apresenta as técnicas utilizadas para testes estruturais de software, ou caixa branca. Técnicas de programação procedimental e orientada a objeto serão estudadas e selecionadas para a aplicação em um estudo de caso no próximo capítulo.

Segundo PAULA FILHO (2009) o teste de unidade é um tipo de teste executado pelos desenvolvedores na fase de codificação. O objetivo é exercitar uma unidade de código, ou seja, um conjunto de métodos, uma classe ou cluster (um grupo de classes destinadas a um fim em comum). Em média 70% dos defeitos do programa que seriam encontrados pelo usuário são identificados se os testes forem bem planejados e aplicados.

Algumas variações de teste estrutural serão discutidas neste capítulo, PRESSMAN (2011) se refere a elas como testes de estrutura de controle, são elas:

1. Teste de caminho básico;
2. Teste de condição;
3. Teste de fluxo de dados;
4. Teste de laços (loops);
5. Teste de unidade em programas orientados a objetos;
6. Teste de integração em programas orientados a objetos;
7. Teste Par-a-Par.

2.1. TESTE DE CAMINHO BÁSICO

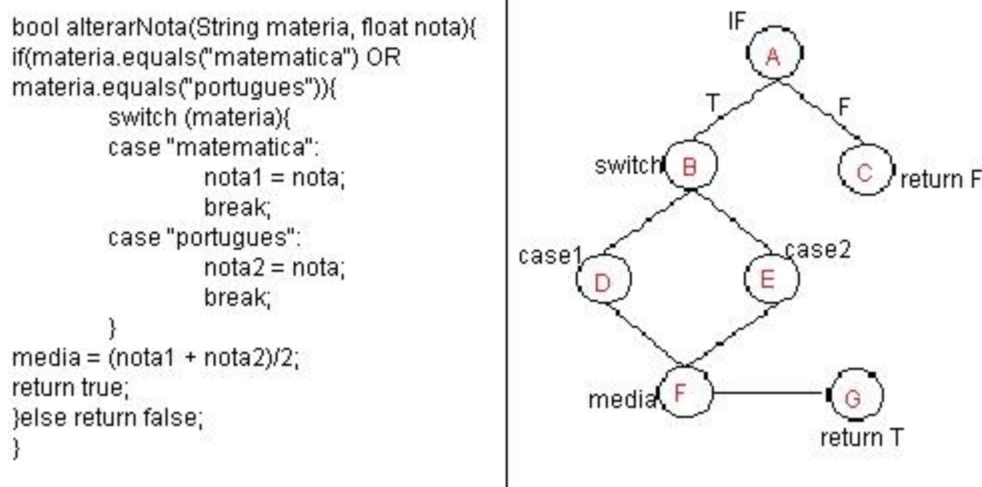
Segundo SOMMERVILE (2007), esta técnica de teste estrutural, proposta inicialmente por Tom McCabe, consiste em exercitar todos os caminhos de uma determinada função ou método pelo menos uma vez, de forma que todas as possibilidades lógicas sejam testadas. Além disso, as declarações condicionais são testadas como verdadeiro e falso.

Entretanto, não é possível testar todas as combinações possíveis em todos os caminhos lógicos do programa. Em programas com loops, pode existir um número

infinito de caminhos. Por este motivo, o teste de caminho básico pode ser feito de 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 que são seqüenciados logicamente pelos ramos. As áreas delimitadas pelos ramos e nós são denominadas regiões, conforme ilustrado na figura 5.

Figura 5 - Exemplo de 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, para que seja desenvolvido um caso de teste com o número máximo de entradas que possam exercitar todos os caminhos independentes pelo menos uma vez.

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.

$$C = 2 + 1 \quad => \quad C = 3 \text{ caminhos independentes}$$

2.2. TESTE DE CONDIÇÃO

A finalidade deste tipo de teste é detectar erros nas condições lógicas do código. Desenham-se casos de testes que compreendam o máximo de combinações de valores de entrada possíveis. Se o número de combinações for muito grande, projeta-se a pilha de testes com os valores de entrada mais relevantes para exercitar a condição (PAULA FILHO, 2009, p. 426).

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 seja planejado 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 *apud* TAI, K. C., 1989), 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 utiliza-se 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.

2.3. TESTE DE 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 6, 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 6 - Função main do programa Identifier.

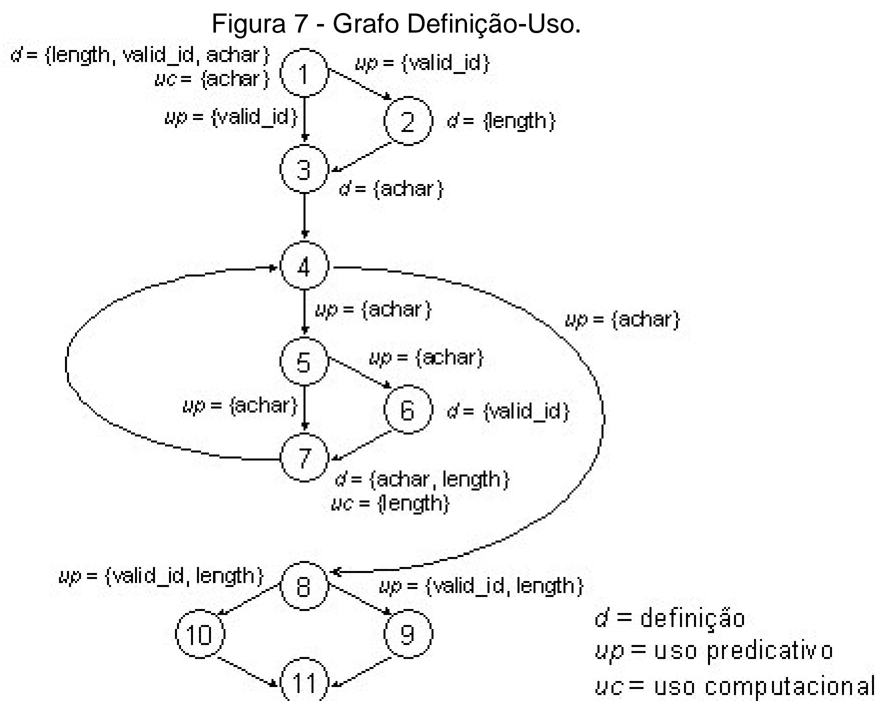
```
*01*  {
*01*    char achar;
*01*    int length, valid_id;
*01*    length = 0;
*01*    printf("Identificador:");
*01*    achar = fgetc(stdin);
*01*    valid_id = valid_s(achar);
*01*    if(valid_id)
*02*            length = 1;
*03*    achar = fgetc(stdin);
*04*    while(achar != 'n');
*05*    {
*05*            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 6 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 7. 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 7 - Grafo Definição-Uso.



Fonte: DELAMARO, MALDONADO e JINO (2007, p. 61)

2.4. TESTE DE 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 8 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

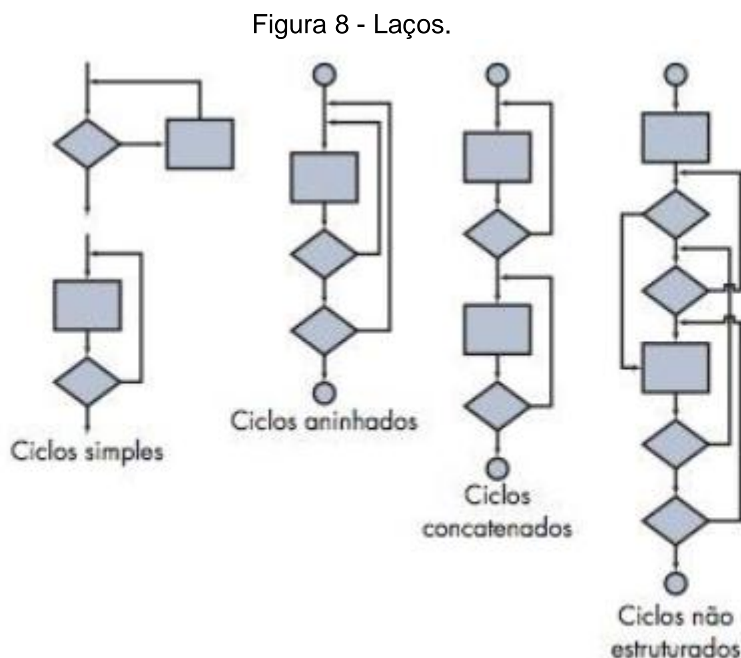
do laço. Para simplificar, PRESSMAN (2011, p. 439 *apud* Beizer, B., 1990) sugere uma abordagem que diminui a quantidade de testes. A técnica consiste em começar exercitando os laços mais internos da estrutura com os mesmos testes sugeridos para os *laços simples*, fixando os laços externos com valores 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 8 representa graficamente os tipos de laços:

Figura 8 - Laços.



Fonte: PRESSMAN (2011, p.438)

2.5. TESTE DE 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 *apud* Harrold e Rothermel, 1994), este critério denomina-se teste **intramétodo**.

Figura 9 - 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 9, 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.

2.6. TESTE DE INTEGRAÇÃO EM PROGRAMAS OO

Depois do teste de unidade, FRANCHIN (2007, p. 53 *apud* Harrold e Rothermel, 1994) 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 9) 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.

2.7. TESTE 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 10 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 10 - Métodos públicos entre classes diferentes.

```
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 10, os pares de métodos são:

Tabela 3 - Pares para geração 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.

3. ESTUDO DE CASO

Este capítulo descreve e executa os testes estruturais em trechos chaves de códigos de um software Java. Buscou-se exercitar somente as principais unidades do programa para que a proposta da monografia seja atingida sem estender desnecessariamente o seu tamanho.

Para a execução dos testes será usado o ambiente de desenvolvimento NetBeans IDE 8.0.2 e o plug-in integrado do framework de testes de unidade JUnit versão 4.

3.1. FOX PDV (PONTO DE CAIXA)

O programa que é estudado foi desenvolvido na linguagem Java, da empresa Sun Microsystems, comprada pela Oracle em 2009. A linguagem é compilada para uma codificação *bytecode*, que executada por uma máquina virtual Java JVM (Java virtual machine), tornando-a perfeita para a adaptação em diferentes plataformas e sistemas operacionais. Seu web site principal é: <http://www.oracle.com/technetwork/java/> (CAELUM, acesso em 29/09/2015).

Basicamente, o sistema é um software ponto de caixa que conta com um controle de estoque, um módulo de vendas por código de barras, impressão de cupom não fiscal e relatórios de movimentações. Os seguintes requisitos funcionais foram retirados do próprio contrato de venda do sistema Fox PDV:

1. CONTROLE DE ACESSO

- Haverá um controle de acesso baseado em usuário e senha. Para utilizar as funcionalidades do software o usuário deverá estar cadastrado na base de dados com as informações: nome, email e telefone.

2. CONTROLE DE PROCESSOS

- O software deverá armazenar em um banco de dados MySql um registro de cada venda realizada incluindo baixa em estoque com os seguintes dados:

data exata da venda, total pago, o nome do vendedor e a listagem dos produtos vendidos. Com relação ao produto: nome, preço por kg/g e quantidade.

- O software deverá organizar as informações citadas acima junto com as informações da empresa para impressão em máquina de cupom NÃO FISCAL. Sendo de responsabilidade do CONTRATANTE a posse e manutenção da impressora.

- O software deverá estar apto para coletar todas as informações contidas no código de barras gerado pela balança. Sendo a balança e o aparelho leitor de código de barras de total responsabilidade de posse e manutenção do CONTRATANTE.

- O software deverá fornecer a opção de backup das informações armazenadas a qualquer momento que solicitado. Sendo de responsabilidade do CONTRATANTE o armazenamento dos arquivos de backup com extensão .sql.

- O software deverá apresentar um formulário para o cadastro de produtos e as suas informações, citadas no primeiro tópico do item 2 deste mesmo anexo. Sendo de responsabilidade do CONTRATANTE a integridade das informações inseridas.

3. RELATÓRIO

- O software deverá disponibilizar a emissão do relatório em tela com opção para impressão em folha tamanho A4.

- As opções de filtros para a geração do relatório são: produtos mais vendidos em um determinado período de tempo; horários com mais vendas em um determinado período de tempo; relação dos produtos em estoque; relação de todos os produtos, estoque e valor.

Utilizamos o banco de dados gratuito MySQL com um servidor Apache local. O banco conta com 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 Venda.id.

O motivo pelo qual o desenvolvemos foi aprimorar nossas habilidades junto a disciplina de programação Java na Fatec Americana, em 2012. Juntos, eu e meu até hoje amigo Felipe Almeida dos Santos, pessoa por quem guardo grande consideração, desenvolvemos este software sob medida para uma empresa que vende produtos naturais no varejo no centro da nossa cidade. Até a presente data o software vem sendo aprimorado e expandido para outras empresas.

Na tabela 4 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 11.

Tabela 4 - Estrutura principal do sistema Fox PDV

Model	Controller	View
Produto	Util	JFrameProduto
Venda		JFrameVendas
VendaProduto		JFrameRelatorio

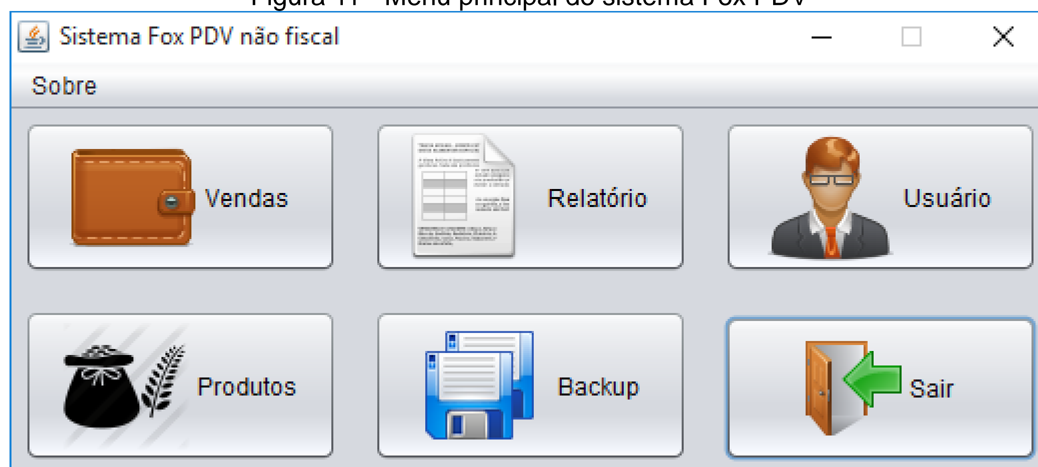
Fonte: Autoria própria.

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 11 - Menu principal do sistema Fox PDV



Fonte: Autoria própria.

3.2. JUNIT 4

O JUnit é um framework de teste estrutural em Java que possibilita de forma simples a execução de casos de teste comparando valores de entrada com valores esperados em um determinado método. O modo como o JUnit trabalha é automático. Ao executá-lo, um relatório é exibido com o resultado dos testes. Se alguma saída inválida, exceção ou erro não esperado ocorrer, o JUnit mostra a origem do erro (DELAMARO, MALDONADO e JINO, 2007, p. 171).

No IDE NetBeans 8, na sua forma mais completa de instalação, o JUnit já está integrado, porém é possível facilmente adicionar o plug-in do framework em qualquer ambiente de desenvolvimento, como o eclipse, por exemplo. Atualmente ele está na versão 4.

A forma de funcionamento é simples. Para cada classe que será testada, adiciona-se uma classe de teste JUnit. Como prática de bom desenvolvimento, aconselha-se criar um pacote apenas para as classes de teste. O JUnit cria automaticamente 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 01/10/2015).

Na tabela 5 são descritos alguns dos comandos que são utilizados nas classes de testes.

Tabela 5 - Principais 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 PDV, 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.

```
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.");
    }
```

Para exercitar o método do botão Inserir com 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 métodos verificaCodigo(string) e verificaMoeda(string) da classe Util verificam se o valor passado por parâmetro não é negativo, conforme apresentado a seguir.

```
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;
}
```

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

esta situação 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 - Teste do método 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,

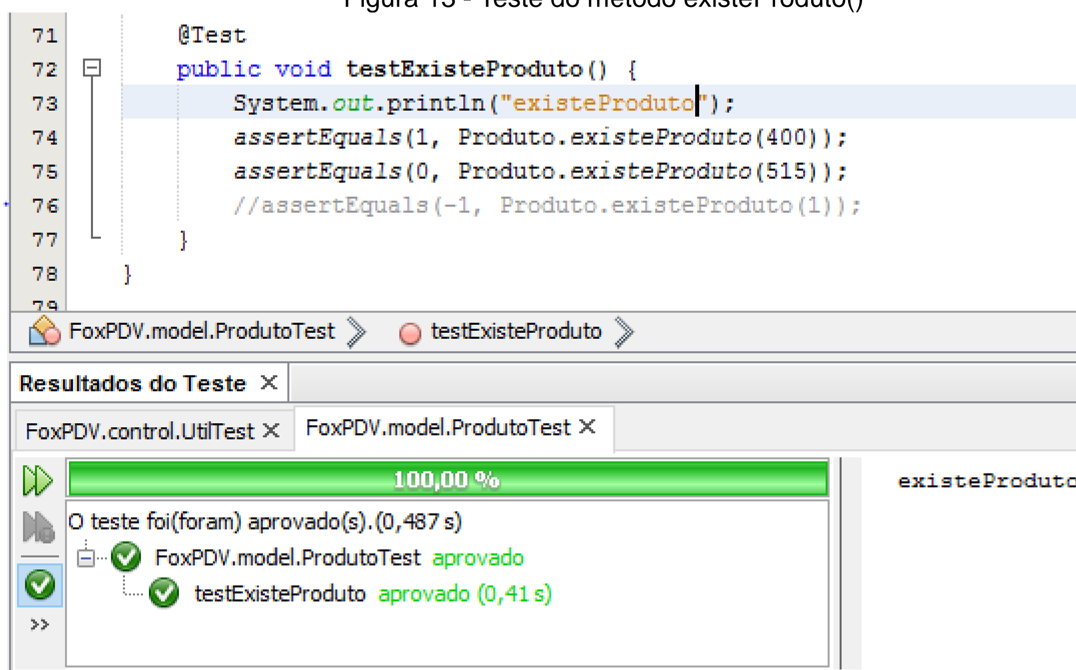
sendo o retorno 1 para produto existente, -1 para erro na busca e 0 para produto não existente. O resultado do teste deste método é exibido na figura 13.

```
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;  }
```

Figura 13 - Teste do método existeProduto()



Fonte: Autoria própria.

Na figura 13, 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 14 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 14 - Teste do método existeProduto() com o servidor do banco de dados 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 15.

```
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;
  }
```

Figura 15 - 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 15. 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.

4. CONSIDERAÇÕES FINAIS

Buscou-se responder a problemática desta pesquisa (como garantir que estruturas complexas de laços e condições a nível de código sejam validadas pela qualidade de software?) através da apresentação de técnicas de teste estrutural de software e da demonstração do teste na prática fazendo uso de um framework Java.

Os métodos de teste estrutural apresentados no capítulo 2 serviram para satisfazer o objetivo geral da pesquisa inicialmente proposto. Baseando-se na pesquisa, conclui-se que muitos erros no software podem ser identificados pelos próprios desenvolvedores graças aos testes estruturais, evitando assim que estes mesmos defeitos sejam encontrados nos testes funcionais realizados pela equipe de testes do projeto.

Os objetivos específicos que nortearam este trabalho se fazem presentes na fundamentação teórica realizada no capítulo 1, onde é descrito as fundamentações e os motivos da importância do teste de software; e no terceiro capítulo, no qual busca-se aplicar de maneira prática como são realizados os testes diretamente no código do programa.

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. A pesquisa bibliográfica realizada nesta etapa se fez efetiva para demonstrar também, que a extensão da área de testes é grande, já que é dividido em três modalidades de teste: caixa branca, ou estrutural, e caixa preta, ou funcional e caixa cinza. Devido a essa abrangência mostrou-se que existem normas que regulamentam especificamente este campo da engenharia de software e que tudo deve ser devidamente documentado.

Para um completo projeto de testes dentro de um projeto de sistema, as três abordagens devem ser utilizadas. O teste de caixa branca realizado na fase de desenvolvimento pelo desenvolvedor valida e corrige erros no nível mais baixo do programa e garante que as estruturas internas do software estejam alinhadas com

as chamadas entre métodos. O teste de caixa preta realizado pela equipe de testes após a conclusão da fase de codificação serve para validar se os requisitos funcionais do sistema estão de acordo com o que foi desenvolvido. A terceira abordagem de testes, caixa cinza, é realizada entre as outras duas, para melhorar a eficiência dos testes de sistema e garantir a perfeita comunicação entre os desenvolvedores e os testadores quanto às estruturas de código e desenho do sistema.

Os principais autores referenciados na pesquisa possuem obras completas sobre engenharia de software e seus ensinamentos são amplamente utilizados nos trabalhos de graduação do curso de Análise e Desenvolvimento de Sistemas na Fatec. Autores como Roger S. Pressman e Ian Sommerville foram essenciais para o desenvolvimento da temática teste de software.

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.

O âmbito inicial era estudar técnicas de teste estrutural a fim de adquirir um conhecimento importante para a área de desenvolvimento de software. Após a realização da pesquisa, o campo dos testes estruturais mostrou-se extenso e com grandes estudos já realizados. Para cada estrutura complexa de código, há uma orientação de como exercitá-lo em busca de erros. Devido a essa vastidão de conteúdo, procurou-se neste estudo apresentar as formas mais conhecidas e básicas no que se refere aos testes de código.

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 PDV 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 a este seria o aprofundamento da norma IEEE 829/1998, desde a abordagem da metodologia de testes à especificação detalhada dos oito documentos citados na fundamentação teórica desta pesquisa. Um projeto devidamente documentado facilita a compreensão das ações que já foram ou que serão tomadas no decorrer do projeto.

A análise detalhada de estruturas de código como laços e condições, assim como a definição da sua complexidade ciclomática e os caminhos de testes são possibilidades de estudos que podem ser aprofundados para sistemas procedimentais. Técnicas de teste em programas orientados a objetos baseados ou não nas apresentadas neste trabalho poderiam ser melhores demonstradas em uma nova oportunidade, pois atualmente é a estrutura de programação dominante no mercado.

REFERÊNCIAS

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **Citação**: NBR-10520/ago - 2002. Rio de Janeiro: ABNT, 2002.

_______. NBR-6023/ago. 2002. Rio de Janeiro: ABNT, 2002.

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.

CAELUM – Ensino e Inovação. Disponível em <https://www.caelum.com.br/>, consultado em 29/09/2015.

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 01/10/2015.

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.

PAULA FILHO, 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 2015.

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