UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

GILCIMAR DIVINO DE DEUS

Avaliação de Técnicas de Teste para Dispositivos Móveis por Meio de Experimentação

Goiânia
2009

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE DISSERTAÇÃO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Avaliação de Técnicas de Teste para Dispositivos Móveis por Meio de Experimentação

Autor(a): Gilcimar Divino de Deus

Goiânia, 14 de Julho de 2009.

Gilcimar Divino de Deus – Autor

Auri Marcelo Rizzo Vincenzi – Orientador

GILCIMAR DIVINO DE DEUS

Avaliação de Técnicas de Teste para Dispositivos Móveis por Meio de Experimentação

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Computação.

Área de concentração: Engenharia de Software.

Orientador: Prof. Auri Marcelo Rizzo Vincenzi

Goiânia
2009

GILCIMAR DIVINO DE DEUS

Avaliação de Técnicas de Teste para Dispositivos Móveis por Meio de Experimentação

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Computação, aprovada em 14 de Julho de 2009, pela Banca Examinadora constituída pelos professores:

Prof. Auri Marcelo Rizzo Vincenzi
Instituto de Informática – UFG
Presidente da Banca

Prof. Fábio Nogueira de Lucena
Universidade Federal de Goiás – UFG

Prof. Edmundo Sérgio Spoto
Universidade Federal do Vale do São Francisco - UNIVASF

Gilcimar Divino de Deus

Graduou-se em Ciência da Computação na UCG - Universidade Católica de Goiás, nesse período ministrou cursos sobre tecnologias cliente-servidor e palestras sobre Padrões de Projeto para alunos e professores. Durante o Mestrado, na UFG - Universidade Federal de Goiás, desenvolveu artigos à comunidade científica, participou de diversos eventos de engenharia de software e ofereceu à comunidade cursos gratuitos e palestras relacionados ao Teste de Software.

Aos meus pais José e Divina, à minha esposa Savanna, e a minha irmã Gilciléia.

Agradecimentos

Primeiramente à Deus pela minha saúde e por ter colocado mais essa oportunidade em meu caminho. Aos meus pais pela educação, disciplina, orações e amor dedicados, por tantos anos, à minha pessoa. À minha querida esposa, Savanna, pela paciência e apoio nos momentos difíceis. À minha irmã Gilciléia pela atenção e dedicação quando precisei. Aos amigos Mestres e Doutores da UFG, em especial ao Prof. Auri pela confiança em mim depositada. E aos meus familiares, amigos e colaboradores do INF que contribuíram de alguma forma, por mais simples que seja, para a realização deste sonho.

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

José de Alencar

Resumo

de Deus, Gilcimar Divino. **Avaliação de Técnicas de Teste para Dispositivos Móveis por Meio de Experimentação**. Goiânia, 2009. 117p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

A qualidade de software é almejada por todos profissionais da computação. Muitas técnicas, métodos e ferramentas foram propostas para ajudar a alcançar um alto grau de qualidade no software. Devido ao aumento no número de dispositivos móveis no mercado diariamente, uma grande quantidade de software para esses dispositivos está em desenvolvimento. Para realizar os testes nos softwares para dispositivos móveis é importante levar em consideração as características tanto do dispositivo quanto do software visando a obter um produto de qualidade. Um pacote de experimentação que visa avaliar algumas das mais conhecidas técnicas de teste de software aplicadas no contexto de programas para dispositivos móveis é proposto. O presente trabalho apresenta o pacote proposto, bem como os resultados obtidos após as três replicações deste pacote e mostra qual técnica, dentre as avaliadas, é mais adequada para garantir mais qualidade ao produto de software Java ME. As replicações desse pacote foram executadas em ambiente controlado, e apresenta uma base de informações consistentes sobre o uso das técnicas de teste em programas Java ME. Novas replicações podem ser realizadas para coletar mais dados e aumentar a confiança dos resultados obtidos até o momento. O objetivo final, após uma série de replicações é o desenvolvimento de uma estratégia incremental para o teste de produtos de software Java ME. Uma versão inicial dessa estratégia é apresentada neste trabalho e pode ser refinada futuramente. Além disso, os dados obtidos durante os experimentos servem de subsídios para a melhoria constante da ferramenta de teste JaBUTi/ME, visando sempre a maximizar o auxílio e reduzir os custos na condução da atividade de teste.

Palavras–chave

Pacote de Experimento, Técnica Estrutural, Técnica Funcional, Técnica ad hoc, Dispositivos Móveis, Java ME, JaBUTi/ME

Abstract

de Deus, Gilcimar Divino. **The Evaluating of Test Techniques for Mobile Devices with Package of Experimentation**. Goiânia, 2009. 117p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Software quality is aimed at for every person that works with computation. Many techniques, methods and tools have been proposed helping us to get high level software quality. Due to the increase in the number of mobile devices available daily, there is a high number of software products for such devices are under development. To test the software for mobile devices it is important to know the characteristics of both the device and the software itself aiming at to get high quality products. One package of experimentation aiming at evaluating some well known testing techniques applied in mobile software is proposed. This work describe the proposed package, the results obtained after three replications, and highlight which technique is more adequate for grant quality in Java ME software. The replications occur in a controlled environment, producing a consistent information database about the usage of the testing techniques in Java ME programs. New replications can be carried out for collecting more data and increasing the confidence in the results obtained so far. The final objective, after many replications, is to develop one incremental strategy for testing Java ME software. A preliminary version of this strategy is presented in this work and can be further refined. Moreover, the data obtained during the experiment helps improving the JaBUTi/ME testing tool, aiming at to maximize the support and reduce the cost of the testing activity.

Keywords

Package of Experiments, Estructural Technique, Functional Technique, ad hoc Technique, Mobile Devices, Java ME, JaBUTi/ME

Sumário

Lista de Figuras **11**

Lista de Tabelas **12**

1 Introdução **13**
- 1.1 Motivação 15
- 1.2 Objetivos 15
- 1.3 Considerações Finais 16

2 Engenharia de Software **17**
- 2.1 Teste de Software 19
- 2.2 Casos de Teste 21
  - 2.2.1 Fases do Teste de Software 23
    - Teste Unitário 23
    - Teste de Integração 25
    - Teste de Sistema 25
  - 2.2.2 Depuração 26
- 2.3 Considerações Finais 26

3 Técnicas de Teste de Software **28**
- 3.1 Critérios de Teste 28
- 3.2 Técnica *ad hoc* 29
- 3.3 Técnica Funcional 30
  - 3.3.1 Particionamento de Equivalência 31
  - 3.3.2 Análise de Valor Limite 31
- 3.4 Técnica Estrutural 32
  - 3.4.1 Grafo de Fluxo de Controle 34
  - 3.4.2 Critérios de Fluxo de Controle 36
  - 3.4.3 Critérios de Fluxo de Dados 37
  - 3.4.4 Critérios Baseados em Defeitos 38
- 3.5 Ferramentas de Teste 40
  - 3.5.1 JaBUTi 41
  - 3.5.2 JaBUTi/ME 46
- 3.6 Considerações Finais 52

4 Engenharia de Software Experimental **53**
- 4.1 Experimentos na Engenharia de Software 54
- 4.2 Considerações Finais 59

5 Pacote de Experimentação Proposto para a JaBUTi/ME **60**
5.1 Organização do Pacote Proposto 61
5.2 Análise dos Dados 76
5.3 Estratégia de Teste 82
5.4 Considerações Finais 83

6 Conclusão **84**
6.1 Trabalhos Futuros 86

Referências Bibliográficas **88**

A Materiais do Experimento **94**
A.1 Especificações dos Programas 94
A.1.1 Especificação do Programa B.M.I 94
A.1.2 Especificação do Programa AntiPanela 94
A.1.3 Especificação do Programa CarManger 95
A.1.4 Especificação do Programa CodiceFiscale 95
A.2 Guia para Utilização do Experimento 97
A.2.1 Instruções para aplicação da técnica ad hoc 97
A.2.2 Instruções para aplicação da técnica funcional 97
A.2.3 Instruções para aplicação da técnica estrutural 98
A.3 Formulários do Pacote de Experimentação 98
A.3.1 Formulário 1 - Perfil do Participante 98
A.3.2 Formulário 2 - Definição dos Grupos 99
A.3.3 Formulário 3 - Defeitos Encontrados 99
A.3.4 Formulário 4 - Sugestões 99
A.3.5 Formulário 5 - Avaliação do Experimento 100
A.4 Defeitos Inseridos nos Programas 101
A.4.1 Programa CarManager 101
A.4.2 Programa CodiceFiscale 105
A.4.3 Programa AntiPanela 109
A.5 Dados sem a Redução 112

Lista de Figuras

2.1 Fases do Teste de Software. 23

3.1 Relações entre técnica, critério, requisito e casos de teste [23]. 29
3.2 Exemplos de estruturas básicas do GFC. 34
3.3 Exemplo de GFC. 35
3.4 Exemplo de GFC Reduzido. 35
3.5 Exemplo de GFC Reduzido adaptado [62]. 36
3.6 Hierarquia de Inclusão dos Critérios de Fluxos de Dados. 37
3.7 Hierarquia de Critérios de Teste Estrutural. 39
3.8 Operações da Ferramenta JaBUTi. 43
3.9 Grafo Def-Uso e Coloração dos Pesos de um programa *Java*. 44
3.10 Análise de Cobertura dos Critérios Estruturais por meio da JaBUTi. 45
3.11 Ambiente cross plataform. 46

4.1 Processo Experimental Segundo Wholin [66]. 57

5.1 Esquema de monitoramento adotado. 66
5.2 Processo estatístico. 72
5.3 Método estatístico adotado. 73
5.4 Perfil dos Participantes. 77
5.5 Resultado do Questionário. 77
5.6 Avaliação do Programa Antipanela por Técnica. 78
5.7 Avaliação do Programa CarManager por Técnica. 79
5.8 Avaliação do Programa CodiceFiscale por Técnica. 79
5.9 Número de Casos de Teste por Técnica. 80
5.10 Número de Defeitos Encontrados por Técnica. 81

Lista de Tabelas

3.1 Características apresentadas pelas ferramentas de teste OO (Adaptada de [63]). 42

4.1 Comparação de Estratégias de Pesquisa. 55

5.1 Programa Selecionado para o Treinamento e sua Complexidade. 65
5.2 Programas Selecionados para Prática dos Participantes e suas Complexidades. 66
5.3 Taxonomia de Defeitos. 67
5.4 Dados Coletados por Técnica e Critério 69
5.4 Dados Coletados por Técnica e Critério 70
5.5 Hipóteses Estatísticas. 71
5.6 Teste de Normalidade - Shapiro Wilk 74
5.7 Teste de Kruskal-Wallis - Rank Sum Test 74
5.8 Teste de Comparação Múltipla de Kruskal-Wallis 74
5.9 Cobertura Média dos Critérios e Número de Casos de Teste 82
5.10 Avaliação do Curso e Auto-Avaliação 82

A.1 Caracteres Alfabéticos Pares 96
A.2 Caracteres Alfabéticos Ímpares 96
A.3 Resto para o Dígito Verificador 96
A.4 Exemplo de Codice Fiscale 96
A.5 Caracteres Alfabéticos Pares 100
A.6 Defeitos Inseridos no Programa CarManager 102
A.6 Defeitos Inseridos no Programa CarManager 103
A.6 Defeitos Inseridos no Programa CarManager 104
A.7 Defeitos Inseridos no Programa CarManager 106
A.7 Defeitos Inseridos no Programa CarManager 107
A.7 Defeitos Inseridos no Programa CarManager 108
A.8 Defeitos Inseridos no Programa AntiPanela 110
A.8 Defeitos Inseridos no Programa AntiPanela 111
A.9 Dados do Experimento Sem Redução 113
A.9 Dados do Experimento Sem Redução 114
A.9 Dados do Experimento Sem Redução 115
A.9 Dados do Experimento Sem Redução 116

CAPÍTULO 1

Introdução

O mercado mundial está repleto de produtos, desde copos, panelas, automóveis, celulares, equipamentos médicos, dentre outros. Alguns são ou já foram confeccionados artesanalmente, outros são parcial ou totalmente industrializados. Aqueles de origem da indústria normalmente são construídos por automatização, ou seja, existe um software ou no processo de industrialização ou inerente no próprio produto.

A maioria dos consumidores necessita de produtos com alta qualidade, segundo a ISO/IEC 8402 de 1994, qualidade de um produto se define da seguinte forma: “Conjunto de propriedades de um produto ou serviço, que lhe conferem aptidões para satisfazer as necessidades explícitas ou implícitas.” Sendo assim, qualquer produto deve em primeiro lugar satisfazer o consumidor. Para atingir esse objetivo, empresas implantam processos, métodos, pesquisas de satisfação buscando normatizar e controlar todos os passos diretos e indiretos na criação de um produto.

Com o produto de software não é diferente, o chamado “usuário” ou “cliente”, que é o consumidor ou utilizador de um programa computacional, precisa confiar no software que está sendo utilizado. Pelo princípio da definição de qualidade, para ocorrer uma determinada “confiança” no produto, o software deve, primeiramente, satisfazer o seu usuário, para que o mesmo possa acreditar no software nele introduzido, ou no produto final que ele produz (por exemplo, um carro).

O atendimento das necessidades do cliente é o principal foco de um software com qualidade, ou seja, o cliente fica com a sensação de que o software cumpriu o que era desejado. As definições mais aceitas na literatura para qualidade aplicada ao software são: “O grau com que um sistema, componente ou processo cumpre os requisitos especificados e as necessidades ou expectativas do cliente.” [37]

> “Conformidade com os requisitos de desempenho, os requisitos funcionais explicitamente declarados, as normas de desenvolvimento explicitamente documentadas e finalmente as características implícitas esperadas em todo o software desenvolvido de uma forma profissional.” [53]

Algumas empresas buscam diversas alternativas para conseguir alcançar esses objetivos, aplicando normas e modelos de qualidade no processo de construção do soft-

ware e no produto final gerado. Alguns modelos mais conhecidos são o MPS-BR (Melhoria de Processo de Software Brasileiro) e o CMMI *(Capability Maturity Model Integration)*, este último desenvolvido pelo SEI *(Software Engineering Institute)* [58] e a norma ISO/IEC 15504 [5], a ISO 9000 [1] para gerência da qualidade (mais especificamente as normas ISO 9001 [2], ISO 9002 [3] e ISO 9003 [4]).

Uma das exigências mais comuns desses modelos e normas é a atividade de Verificação, Validação e Testes (VV&T). A verificação busca averiguar se o produto está sendo construído corretamente, enquanto a validação assegura que o produto que está sendo produzido está de acordo com os requisitos do usuário. As atividades de V&V podem ser realizadas de forma estática ou dinâmica. Revisão, inspeção, técnicas de leitura, dentre outras, são consideradas atividades estáticas de verificação e validação de um produto. Existem técnicas que apóiam a atividade de testes [23], em que produto considerado para execução dos testes não é a execução do produto final, mas sua especificação ou estrutura. A avaliação dinâmica de um produto de software é conhecida como a atividade de teste. Nessa atividade o objetivo é avaliar o comportamento do programa em tempo de execução, no qual o objetivo principal é provar que o software contém não-conformidades em seus resultados apresentados. Embora em alguns momentos não seja possível executar o próprio programa, a realização de simulações pode tornar-se viável para avaliar componentes.

Este trabalho de mestrado está focado na atividade de teste, cujo produto dinâmico é utilizado como insumo, e o resultado é a avaliação do produto, se está de acordo com os resultados esperados. Para essa avaliação do produto, existem algumas técnicas que auxiliam na descoberta de defeitos inerentes ao produto. Algumas dessas técnicas são: *ad hoc* (sem técnica ou empírica) que é baseado na experiência do testador e na qual é realizada uma sequência de testes até que o testador se dê por satisfeito no teste do software. Outra é a técnica funcional, que trata o software como uma caixa preta, são fornecidas entradas e avaliadas as saídas, sem conhecimento de como a saída foi produzida. Por outro lado, a técnica estrutural tenta ajudar o testador a criar casos de testes que exercitem trechos de código ainda não executados (pode ser utilizada complementando a técnica funcional ou *ad hoc*). E por fim, a técnica baseada em defeitos, que inclui o teste de mutação, que tenta simular um programador cometendo possíveis “enganos de programação” no código do programa, gerando-se variações do programa original. Dessa forma ao executar o teste no original e suas variações (os programas mutantes) o objetivo é eliminar os mutantes equivalentes, mostrando que não há mutantes gerados que resulte na mesma saída do programa original.

1.1 Motivação

O ano de 2008 fechou com 145 milhões de celulares operando no Brasil [6]. Da mesma forma que crescem os números de dispositivos móveis, cresce também os aplicativos construídos para serem implantados nesses milhões de aparelhos lançados no mercado. Entretanto, é extremamente importante garantir a qualidade dos software inerentes a esses dispositivos antes do lançamento destes produtos, pois após o lançamento, o custo de correção de centenas ou milhares de dispositivos pode tornar-se proibitivo. Buscar uma forma de garantir a qualidade dos software existentes para os dispositivos móveis é a grande motivação deste trabalho.

A grande maioria dos celulares disponíveis no mercado utilizam software implementados utilizando a linguagem *Java* [41], devido a essa característica foi selecionada uma ferramenta que apóie os testes em componentes e programas criados nessa linguagem.

Muitas ferramentas apóiam a execução de testes nas diversas técnicas relatadas anteriormente. Uma delas é a JaBUTi *(Java Bytecode Understanding Testing*) [64]. Existem algumas variantes da ferramenta, e nesse trabalho será utilizada a versão para apoiar os critérios de testes estruturais em dispositivos móveis. A JaBUTi/ME *(Java Bytecode Understanting Testing/Micro Edition)* apóia nos testes estruturais dos aplicativos e componentes criados em *Java* e que podem ser implantados em dispositivos móveis [24]. Dentre suas vantagens as mais relevantes são: 1) Não obrigatoriedade do código fonte escrito em *Java* para realização dos testes, utiliza-se o *bytecode* [18], também conhecido como código-objeto que é interpretado pela *JVM (Java Virtual Machine)* [45] que interpreta e executa o *bytecode* do programa *Java*; 2) possibilidade de executar os casos de testes tanto em emuladores instalados em computadores pessoais, quanto nos dispositivos móveis reais; 3) visualização do grafo estrutural do componente com pesos, que mostra quais os nós que deveriam ser executados para que seja coberta uma maior estrutura do código fonte; 4) flexibilidade no envio das informações de cobertura de um dispositivo real para análise posterior na própria JaBUTi/ME; 5) dados estatísticos da análise de cobertura seguindo diversos critério estruturais.

1.2 Objetivos

Um dos problemas da Engenharia de Software é a falta de conhecimento mensurável de qualidade que apóie os seus produtos gerados. Buscando alcançar uma base de conhecimento foi criada a Engenharia de Software Experimental [60], que dentre outros recursos, usa a criação de pacotes de experimentação os quais permitem que uma determinada técnica, metodologia, processo, dentre outros, seja avaliado continuamente, por

meio de replicações, para aumentar a base de conhecimento exercitado, buscando comprovar uma determinada teoria avaliada no experimento. Existem diversas iniciativas na criação de pacotes de experimentação [29, 25, 35], inclusive ferramentas que apóiam o armazenamento e facilita a replicação de pacotes de experimentos [17] [44] [8] [9] [59] [38].

O objetivo deste trabalho é conduzir estudos experimentais visando avaliar três técnicas de teste de software, *ad hoc*, funcional e estrutural, e também avaliar o uso da ferramenta JaBUTi/ME considerando suas características, tanto em emuladores quanto em dispositivos móveis reais. Além disso, usando a definição de pacote de experimentação para apoiar a replicação do experimento conduzido, buscando aumentar a base de conhecimento de testes no cenário de testes Java ME [20], [21].

Em síntese os objetivos são:

1. Avaliar a adequação da ferramenta JaBUTi/ME no teste de produtos Java ME;
2. Desenvolver um pacote de experimentação para avaliar critérios de teste para produtos Java ME;
3. Demonstrar o uso do pacote em treinamentos;
4. Analisar os dados obtidos nas replicações do pacote de experimentação; e
5. Definir uma estratégia de teste incremental com base nos dados coletados.

1.3 Considerações Finais

Neste capítulo foram apresentadas os principais objetivos, bem como as motivações do trabalho. A seguir serão apresentados os principais conceitos da engenharia de software relacionados a qualidade de seus produtos, bem como as fases ou etapas de teste de software e a atividade de depuração.

CAPÍTULO 2

Engenharia de Software

A Engenharia de Software é considerada por muitos uma ciência nova. Com cerca de cinquenta anos chega a ser comparada com um “recém-nascido” quando relacionadas outras engenharias como a Engenharia Civil, praticada desde a época das pirâmides do Egito. Apesar de “poucos” anos é possível observar uma evolução extremamente rápida, que na maioria das vezes traz muitos benefícios para a nossa vida em sociedade. Por essa rapidez na evolução e grande aderência na vida dos humanos, ela é reconhecida como um objeto sério de pesquisa, fazendo com que milhares de profissionais trabalhem em paralelo para contribuir para esse grande crescimento.

Diversas áreas da sociedade como o governo, a educação, a indústria, a ciência, a medicina, defesa civil, dentre outros, desfrutam dessa evolução, na maioria das vezes melhorando a qualidade de vida de todos. Por meio do software há uma difusão de conhecimento, uma explosão de informações que consegue chegar a quase todos os lugares do mundo e interligando todos a praticamente tudo.

A forma de construir um software mudou muito nos últimos anos e continua sua evolução. Antes para construir um software era necessário, na maioria das vezes, ser um matemático exímio e que entendesse também de eletrônica. Os profissionais iniciaram com programas simples construídos em linguagem de máquina, ou seja, os *bits* 0 (zero) e 1 (um). Além da complexidade para construir uma simples calculadora, o projetista estava limitado a poucos *kilobytes* de memória e a um baixo poder de processamento. Em comparação com os software que tem-se hoje como, por exemplo, os jogos “quase reais” que rodam em celulares, os bilhões de dados que são processados por segundo em um supercomputador da NASA e outros, é possível notar que quase tudo mudou às vezes ficando mais simples e mais poderoso.

Outras evoluções na computação são observadas como o surpreendente desempenho do hardware, mudanças profundas nas arquiteturas dos computadores, aumento da memória e desempenho significativo no processamento de informações. Essa sofisticação levou a construção de software e equipamentos jamais imaginados antes e extremamente complexos. Embora sofisticação e complexidade possam produzir excelentes resultados em software bem-sucedidos, em alguns casos pode gerar diversas dificuldades para quem

necessite construir projetos de software de alta complexidade. Segundo Pressman [53], na década de 70 apenas 1% das pessoas sabiam definir o significado de “software de computadores”. Já nos dias atuais esse percentual encontra-se bem maior entre os profissionais de informática, áreas afins e o público geral.

Independentemente da época sempre foi necessário um procedimento, método, processo ou algo nesse sentido para que o software de cinquenta anos atrás fosse produzido e para que o projeto mais arrojado tecnologicamente também obtivesse sucesso. No início da computação, talvez o procedimento não fosse documentado e não estivesse muito claro para os engenheiros de software. Dois passos sempre foram fundamentais para a construção de software desde o início da ciência até os dias atuais, o primeiro era construir o software ou parte do mesmo, e posteriormente verificar se este foi corretamente construído.

Hoje a Engenharia de Software dispõe de diversos artefatos dentre eles, processos, metodologias, livro de experiências, artigos, técnicas, ferramentas, dentre outros. Embora não se tenha uma “receita de bolo”, de acordo com a natureza do software a ser construído é possível agregar um conjunto desses recursos a fim de construir um software com qualidade. Já foi alcançado um grande progresso, mas existe muito a ser feito para que a Engenharia de Software tenha uma plenitude em seu nível de maturidade.

A Engenharia de Software possui uma alta taxa de evolução, em alguns casos como os sistemas legados podem ficar sem ter como expandirem ou evoluírem, devido ao projeto não extensível, tecnologia ultrapassada, documentação pobre ou inexistente e modificações não gerenciadas. Quando um sistema desse porte necessita evoluir como, por exemplo, ser acessado por meio da Internet ou dispositivos móveis, pode ser necessário recriar todo o projeto de maneira que seja mais extensível a tecnologias futuras que atendam a tais requisitos. Essa evolução na Engenharia de Software trouxe diversas contribuições para a construção sistematizada de produtos de software. Alguns modelos foram criados para aferir o grau de maturidade dos processos de desenvolvimento de produtos de software. O CMMI *(Capatibility Maturity Model Integration)* é um exemplo de um processo de melhoria que possui os seguintes níveis [58]: Nível 0 – Incompleto ou *ad hoc*; Nível 1 – Executado; Nível 2 – Gerenciado ou Gerido; Nível 3 – Definido; Nível 4 – Quantitativamente Gerenciado; e Nível 5 – Otimizado. Este modelo define um conjunto de boas práticas genéricas e/ou específicas que devem ser utilizadas para um mesmo objetivo. Outro exemplo de modelo é o MPS.BR (Melhoria de Processo de Software Brasileiro), baseado na ISO 12207, foca exclusivamente em processos de software das empresas brasileiras, e possui os seguintes níveis [52]: Nível G – Parcialmente Gerenciado; Nível F – Gerenciado; Nível E – Parcialmente Definido; Nível D – Largamente Definido; Nível C – Definido; Nível B – Gerenciado Quantitativamente; e Nível A – Em

Otimização. O mesmo está em conformidade com o CMMI e com as normas ISO/IEC 12207 e 15504.

Voltando o foco para a qualidade de software, um fato importante é que no nível 2 do CMMI e no nível F do MPS.BR possuem restrições sobre a qualidade do software produzido pelo processo. Isso significa que ambos definem disciplinas que inicialmente se preocupam com a qualidade do produto gerado. Ou seja, testes nos produtos são exigidos a fim de garantir que o produto atenda seus requisitos. No nível 3 do CMMI e E do MBS.BR o conhecimento adquirido sobre a qualidade de software é centralizado e começa a ser difundido pela corporação. A empresa passa a ter a preocupação de realizar verificações para aumentar a chance de encontrar defeitos enquanto o software estiver sendo implementado, busca criar a cultura de testar, relatar experiências de testes e melhorias constantemente.

Assim pode-se observar que a qualidade é um fator de preocupação dos modelos já nos primeiros níveis de avaliação. Uma das formas de garantir a qualidade de um software é tratada em uma subárea da Engenharia de Software denominada teste de software. Nessa subárea existem diversas formas de testar um software, partindo desde o teste de especificação dos requisitos até caminhos percorridos no código fonte. Mais detalhes sobre o teste de software podem ser vistos a seguir no Capítulo 3.

2.1 Teste de Software

De acordo com o IEEE, uma das definições mais aceita pela literatura para teste é: *"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component"* [37]. O teste de software pode ser visto como um tipo específico de teste, cujo componente ou processo é submetido a uma avaliação de acordo com os resultados produzidos em determinado ambiente.

Algumas controvérsias existem na literatura sobre algumas definições e termos utilizados no teste de software, assim é necessário tornar única a linguagem durante todo o trabalho e serão adotadas as definições colocadas por Delamaro et. al. [23] a seguir:

- Defeito: passo, processo ou definição de dados incorretos;
- Engano: ação humana que produz um defeito;
- Erro: durante a execução de um programa é caracterizado por um estado inconsistente ou inesperado; e
- Falha: resultado produzido é diferente do resultado esperado, pode ser ocasionado por um erro.

O teste de software possui algumas limitações e dentre elas, ele não assegura que um software está correto. Soa estranho, mas a atividade de teste apenas diz que não foi possível encontrar defeitos no software em questão. Só é possível assegurar que um programa não possui defeitos quando todas as possibilidades de entrada de dados foram executadas. Comumente isso não é possível, pois a quantidade de entradas pode ser infinita. Por exemplo, considere um programa que apenas informa se um número é par ou ímpar, para garantir sua correção é necessário testar todos os conjuntos de números possíveis, desconsiderando a limitação numérica dos computadores, esse conjunto de entrada é infinito. Ou seja, não é possível garantir que esse programa está correto por meio de testes exaustivos. Ou mesmo considerando a limitação numérica dos computadores esse resulto poderia levar anos, décadas ou séculos para assegurar a correção do programa. Porém a atividade de teste de software traz uma série de artefatos que ajudam a facilitar a busca por defeitos e serão vistos mais adiante no Capítulo 3. Mesmo com limitações a atividade de teste continua sendo muitíssimo importante para a relação de satisfação do cliente com o software. Segundo Pressman [53] o maior testador (e talvez o melhor) é o próprio cliente. Toda vez que um programa é executado ele é testado. A cada execução torna-se mais perigoso percorrer um trecho ainda não testado e revelações de defeitos acontecerem. “Infelizmente, defeitos estarão presentes. E se o engenheiro de software não os achar, o cliente achará!” [53]

O que os engenheiros de software tentam fazer com a realização de testes é antecipar as ações do cliente, ou seja, percorrer caminhos no programa que o cliente percorrerá quando estiver utilizando o software. Alguns engenheiros de software acreditam que o teste bem sucedido é aquele que não encontra defeitos no componente implementado. Myers [49], um dos clássicos escritores sobre teste de software, cita algumas características que podem guiar o objetivo da atividade de teste, são eles: 1) Teste é um processo de execução de um programa com a finalidade de encontrar um erro; 2) Um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto; 3) Um teste bem sucedido é aquele que descobre um erro ainda não descoberto. Ou seja, um bom teste é aquele que encontra um defeito no componente implementado e um ótimo conjunto de testes é aquele que faz a descoberta de defeito com um esforço e tempo menor.

O teste de software sempre foi uma parte crítica da Engenharia de Software, todos sabem de sua importância, mas na prática dificilmente investem o suficiente nessa atividade. Diversos são os fatores que trazem esse cenário e são alguns deles: 1) Testar software não é uma tarefa trivial; 2) Mercado desprovido de profissionais qualificados; 3) Falta de conhecimento em técnicas e critérios de testes; 4) Faltam relatos de experiências com aplicação prática; 5) Não há informações quantitativas suficientes sobre o custo/benefício do teste de software; 6) O software normalmente só é testado depois do

produto pronto; 7) Poucas universidades focam estudos na área de teste de software; 8) Não há direcionamento para otimizar a descoberta de defeitos; 9) Há um preconceito sobre a produtividade do teste relacionado ao projeto como um todo; 10) Falta de ferramentas que apóiem as atividades de teste de software.

A atividade de teste de software, de um modo geral, também possui algumas limitações conhecidas pela teoria da computação pois: 1) não existe um procedimento de teste de propósito geral que possa ser usado para provar a correção de um programa; e de um modo geral: 2) é indecidível se dois programas computam uma mesma função; 3) é indecidível se dois caminhos de programas computam a mesma função; e 4) é indecidível se um dado caminho é executável. Mesmo com essas limitações é possível utilizar a atividade de teste como aferidora da qualidade de um software.

Tendo em vista que o teste exaustivo é impraticável na maioria dos casos, a grande questão que se coloca é: com quais valores o produto em teste deveria ser executado para encontrar a maior parte dos defeitos com um menor tempo? Tais valores correspondem a uma parte do que são chamados os casos de teste.

2.2 Casos de Teste

De acordo com o RUP *(Rational Unified Process)* [36], caso de teste é um conjunto composto por entradas, resultados esperados e condições de execução que são identificados com a finalidade de avaliar um determinado aspecto em um software.

O caso de teste tem a principal função de registrar e comunicar formalmente as condições e detalhes específicos que deverão ser avaliados de um subconjunto de requisitos, por exemplo, um Caso de Uso [40], desempenho, disponibilidade, dentre outras.

As entradas de um caso de teste geralmente são identificadas como dados fornecidos para que o programa inicie ou continue sua execução a fim de avaliar o resultado produzido. O meio mais comum de realizar a entrada de dados para um software é via dispositivos de entrada como teclado, mouse, dentre outros. Existem outras formas de entradas dados como dados oriundos de outros sistemas, dados lidos de arquivos ou banco de dados, estado do sistema quando dados são recebidos, dentre outros.

As saídas de um caso de teste é o resultado do processamento da informação de entrada e que deve ser avaliada se é o resultado esperado ou se aconteceu alguma divergência nos resultados. O meio mais comum de apresentação da saída é apresentada pelo monitor de vídeo, mas outras formas também podem ser utilizadas como dados serem enviados para outro sistema ou outro dispositivo externo, informações gravadas em banco de dados, alterações no estado do sistema e outros. Existem dois tipos de saídas envolvidas no teste de software, as denominadas saídas esperadas são as informações

que o sistema deve apresentar no caso de não haver falha no sistema, ou seja, está de acordo com os requisitos propostos para o mesmo. As saídas produzidas ou obtidas são as informações mostradas pelo sistema após o processamento do caso de teste, podem divergir das esperadas ou não. Caso não estejam de acordo com as saídas esperadas o sistema apresentou uma falha e deve ser corrigido. Para identificar se uma saída esperada está de acordo com a produzida é necessário uma tomada de decisão, um oráculo é utilizado, normalmente essa tarefa é executada pelo testador, embora ela também possa em alguns casos ser automatizada.

Beizer [12] lista cinco tipos de oráculos, 1) Oráculo “Kiddie” - ao executar o programa observe sua saída caso pareça estar correta o programa deve estar correto; 2) Conjunto de teste de regressão - ao executar o programa compare a saída obtida com a saída produzida por uma versão anterior do programa; 3) Validação de dados - ao executar o programa compare a saída obtida com uma saída padrão determinada por uma tabela, formula ou outra definição aceitável de saída válida; 4) Conjunto de teste padrão - ao executar o programa com um conjunto de teste padrão que tenha sido previamente criado e validado, muito comum em compiladores, navegadores Web e processadores SQL; 5) Programa existente - execute o programa a ser testado e um programa semelhante e compare as saídas, semelhante ao teste de regressão.

As condições de execução são informações úteis ao testador para criar o ambiente correto para a execução do teste, sequência de execução e para avaliar se a saída produzida está de acordo com a esperada. Vale ressaltar que a saída esperada pelo sistema e as pré-condições e pós-condições encontram-se dentro das condições de execução. A ordem de execução pode ser de dois tipos: 1) execução independente e 2) execução em cascata. O primeiro estilo cada caso de teste é auto-contido, ou seja, pode ser executado de forma aleatória, pois em cada execução o estado do inicial do programa é recuperado, permitindo assim, a execução de qualquer caso de teste partindo do mesmo ponto do programa. Suas principais desvantagens são a complexidade que envolve cada caso de teste por ser único e de restaurar o ponto inicial do programa. No segundo, a ordem de execução é de fundamental importância para a execução correta do teste, pois a cada execução de um caso de teste o programa permanece em um estado que serve de entrada para o próximo caso de teste, e assim ocorre até a finalização de toda a cadeia de execução de casos de teste projetada. Sua vantagem é que os casos de testes, que compõem a ordem de execução, tendem a ser menores e mais simples. Por outro lado se um caso de teste falhar é necessário recuperar o estado inicial do programa e repetir toda a sequência de execução novamente.

2.2.1 Fases do Teste de Software

Existem diversas formas de se testar um software, uma delas é aguardar que o software seja totalmente construído e só então iniciar a fase de testes, como no Modelo Cascata [40]. A experiência mostrou aos engenheiros de software que essa estratégia não funciona muito bem. Ao iniciarem os testes uma quantidade exorbitante de falhas é revelada e é muito caro corrigir e testar tudo novamente várias vezes se necessário. Uma estratégia muito difundida na literatura é a divisão das atividades de teste em fases como acontece no desenvolvimento de algum software conforme é ilustrado na Figura 2.1. Utiliza-se a estratégia de "dividir para conquistar" e em cada etapa é realizado um tipo de teste focando em determinada característica do software, iniciando nas unidades menores, depois nas integrações dessas unidades e incrementalmente alcançando todo o sistema. As etapas normalmente são distribuídas no processo de desenvolvimento de software em três fases, unidade, integração e de sistema. No teste de unidade o esforço é focado na menor unidade de projeto que o software é composto, pode ser um método ou procedimento, uma classe, um componente ou um módulo. O teste de integração procura por defeitos nas ligações entre os módulos, ou seja, averiguar se os componentes que foram testados através do teste de unidade quando interligados funcionam como deveriam. E finalmente o teste de sistema tem a finalidade de verificar se os elementos do sistema foram todos adequadamente integrados e produzem respostas de acordo com o especificado para o software.

A seguir, cada uma dessas fases é descrita mais detalhadamente.

Figura 2.1: *Fases do Teste de Software.*

Teste Unitário

Também conhecido com o teste de unidade, no teste unitário é definida qual a menor unidade do software que é possível de ser testada. De acordo com o paradigma de programação e linguagem de programação utilizada é possível identificar diferentes unidades mínimas de um software. Por exemplo, para a programação estruturada utilizando

à linguagem de programação PASCAL [19], a menor unidade a ser testada pode ser um procedimento ou uma função. No paradigma orientado a objeto com a linguagem de programação *Java* [45], por exemplo, pode ser utilizado como unidade para esse tipo de teste um método ou até mesmo uma classe. Um método e um procedimento se assemelham em suas estruturas, são invocados e possuem estrutura interna composta por comandos. O foco deste teste é encontrar erros nessa estrutura interna do módulo do software e pode ser realizado em paralelo com diversas unidades do software.

O teste de unidade é recomendado como sendo o primeiro teste a ser realizado no software, pois se um software como um todo é composto de partes (unidades) e essas unidades apresentam defeitos o software não irá funcionar. Por isso o esforço é concentrado nessas unidades menores e só após o teste de unidade segue incrementalmente para as próximas fases. Alguns dos defeitos mais comuns encontrados nos testes de unidade são: 1) precedência aritmética incorreta; 2) operações mescladas; 3) inicialização de variáveis incorreta; 4) imprecisão numérica; 5) comparação de tipos de dados diferentes ou variáveis diferentes; 6) operadores e precedência lógica incorreta; 7) final de ciclo inadequada ou inexistente; e 8) variáveis de laço inadequadamente alteradas.

Um dos testes mais importantes para uma unidade é testar os limites do domínio de entrada. O software frequentemente falha em seus limites, por exemplo, quando o i-ésimo elemento de um vetor é processado, ou quando ocorre a enésima repetição de um laço, ou com valores mínimos e máximos são encontrados. Um bom projeto de caso de teste deve contemplar nos testes de unidade os testes de limites, aumentando as chances de encontrar defeitos enquanto é realizado o teste de unidade, evitando deixar chegar na fase de teste de sistema para encontrá-los.

Quando uma unidade que está sendo testada necessita realizar chamadas a outros módulos do software é possível utilizar os chamados *stubs*. Estes por sua vez, simulam o componente invocado a fim de focar o teste na unidade previamente selecionada.

Pressman [53] define um conjunto de tarefas para o teste unitário são elas: 1) Projetar testes de unidade para cada componente de software; 2) Revisar cada teste unitário para garantir a cobertura adequada; 3) Conduzir o teste unitário; 4) Corrigir os defeitos descobertos; e 5) Reaplicar os testes unitários até que nenhum defeitos seja revelado ou até que uma condição de parada seja satisfeita. Ele também defende a ideia que quanto mais ligações uma unidade possui, mais complexos são seus casos de testes. Ou seja, se uma unidade foi criada com alta coesão (apenas um método) a tendência é que o número de casos de testes seja reduzido e que os defeitos sejam previstos e apareçam com mais facilidade.

No teste de unidade a técnica de teste mais utilizada, e até mais adequada é a técnica estrutural que será descrita no Capítulo 3.

Teste de Integração

Em software complexos as unidades sozinhas não fazem sentido. Quando se utiliza a ideia de “dividir para conquistar” é necessário ligar as unidades após o seu desenvolvimento. Sendo que a união de todas as unidades forma o sistema completo. Como recomendado por Pressman [53] não precisa esperar todo o sistema ficar pronto para iniciar os testes, encontrar e corrigir defeitos.

As ligações entre os elementos são construídas por meio das interfaces de comunicação das unidades. Ou seja, regras para invocar outros elementos, para que os mesmos possam funcionar de acordo com o esperado. Após ter testado as unidades, de forma incremental é possível iniciar uma nova fase de testes chamada de teste de integração. O principal objetivo desta fase é verificar se as unidades que funcionam independentemente, quando unidas, não divergem do resultado esperado. As formas mais comuns de se conduzir os testes de integração é de forma descendente ou *top-down* e ascendente ou *button-up* [53].

Os tipos de erros mais comuns nos testes de integração são: 1) Dados inválidos nas chamadas às interfaces do componente; 2) Modificações em variáveis globais utilizadas por várias unidades; 3) Incompatibilidade ou perda de precisão nas passagens de valores; e outros. Por exemplo, quando um módulo A chama outro módulo B e passa uma informação de A para B incorreta, há uma enorme chance do módulo B produzir uma saída incorreta ou não esperada.

No teste de integração a técnica de teste estrutural é menos adequada, devido a complexidade da união dos módulos e a técnica funcional, também conhecida como caixa-preta, trabalha de forma mais efetiva nessa fase de teste. Mais detalhes das técnicas funcional e estrutural serão detalhadas no Capítulo 3.

Teste de Sistema

Após terem sido testadas as unidades, as integrações entre essas unidades e o software não apresentar mais falhas, já é hora de montar um pacote completo com todo o sistema, preparar o ambiente e iniciar o que teste de sistema.

Nesse teste as expectativas definidas nas especificações de requisitos do software deverão ser utilizadas como insumo para o projeto dos casos de teste. Nesse caso, o testador tem um trabalho de investigador com o foco em tentar burlar o sistema, tentar passar pelos pontos inseguros do software. Buscar simular o comportamento dos seus usuários, de simples operadores à *hackers* experientes, com a visão de usuário tentar descobrir as falhas produzidas durante a fase de montagem do sistema.

O teste de sistema é o último teste que antecede a entrega do produto de software a seu cliente. Nesse teste é necessário executar todo o sistema sob o ponto de vista

do usuário final, verificando todas as funcionalidades e procurando encontrar falhas no sistema. Um fator importante é o ambiente montado para a realização dos testes, quanto mais próximo for do ambiente real utilizado pelo cliente, menores serão as chances de encontrar surpresas desagradáveis para o cliente.

Dentre os tipos de testes realizados nessa etapa pode-se citar: 1) teste de recuperação; 2) teste de segurança; 3) teste de estresse; 4) teste de desempenho; 5) teste de usabilidade, dentre outros.

Nessa etapa a técnica de teste mais comumente utilizada é a técnica funcional ou caixa-preta, pois verifica se o sistema produz o resultado esperado e não exige-se, nessa etapa, detalhes da estrutura do software. Á técnica funcional será detalhada no Capítulo 3.

2.2.2 Depuração

A atividade de teste tem a função de demonstrar que um produto de software contém defeitos. Desse modo, sendo a atividade de teste bem sucedida é necessário identificar qual o motivo que levou o produto de software a falhar. Nesse ponto, entra em ação outra atividade denominada Depuração. A fundação da atividade de depuração é rastrear o produto em teste e localizar, precisamente, qual defeito resultou na falha detectada pelos testes. Assim sendo, as atividades de teste e depuração podem ser vistas como complementares.

Assim como os testes, depuração é uma área de pesquisa que está recebendo muita atenção da sociedade acadêmica, vários experimentos, modelos e ferramentas já foram desenvolvidos buscando aperfeiçoar essa atividade na construção de software. Pode-se citar um modelo de depuração chamado Hipótese-Validação [39], técnica de fatiamento de programas [28] e outras. As informações oriundas do teste podem ter um papel importante durante a fase de depuração, fornecendo informações para acelerar o processo de busca e correção dos defeitos [14][67][65].

2.3 Considerações Finais

Neste capítulo foram apresentados conceitos da engenharia de software voltados para a melhoria da qualidade de seus produtos. Uma importante atividade adotada para buscar a garantia da qualidade em artefatos dinâmicos é a atividade de teste. Os casos de testes foram apresentados como forma de aperfeiçoar os valores dos conjuntos de entradas, que normalmente é muito extenso. Foram apresentadas as fases ou etapas do teste de software, unidade, integração e sistema. No teste de unidade as menores partes do programa são testadas, no teste de integração as unidades unidas e testadas em conjunto, e no teste de sistema o software é testado como um todo. Ao final, foi mencionado sobre

a atividade de depuração cuja função é localizar onde se encontra a falha identificada pelos testes. A seguir serão apresentadas as principais técnicas de teste de software, suas características e aplicações.

CAPÍTULO 3

Técnicas de Teste de Software

Somente após a década de 70 que o teste de software começou a ganhar a atenção dos cientistas e da indústria. A demanda por software começava a crescer, assim aumentou a preocupação por qualidade de software. Era necessário focar em pesquisas que colaborassem em como encontrar defeitos, desenvolver técnicas, modelos e então criar software que apoiassem as conclusões das pesquisas.

Os primeiros modelos foram desenvolvidos para realizar teste em Máquinas de Estados Finitos ou autômatos [33] são: Método TT, Método DS, Método UIO e Método W. Mais informações sobre esses métodos e uma tabela com a comparação deles podem ser vistos em [23].

Algumas técnicas também foram propostas para apoiar o teste de software, as mais conhecidas são: 1) Técnica *ad hoc* - Sem critério ou teste aleatório; 2) Técnica Funcional ou caixa-preta - Baseado exclusivamente na especificação de requisitos do programa, não é utilizado nenhum conhecimento da estrutura interna do programa; 3) Técnica Estrutural ou caixa-branca - É focado na estrutura interna do programa, ou seja, em sua implementação; e 4) Técnica Baseada em Defeitos - São baseados em informações históricas sobre defeitos cometidos durante o desenvolvimento de programas. Todas possuem características positivas e negativas, e serão analisadas a seguir.

3.1 Critérios de Teste

As técnicas de teste são compostas por critérios de teste. Os critérios sistematizam como os requisitos de teste devem ser produzidos a partir de uma fonte de informação disponível que pode ser uma especificação de requisitos, o código fonte do programa, o histórico de defeitos, dentre outros. A importância dos requisitos de teste se resume em servir de subsídio para geração dos casos de teste e avaliar a qualidade de um conjunto de casos de teste existente. Um esquema de representação é ilustrado na Figura 3.1.

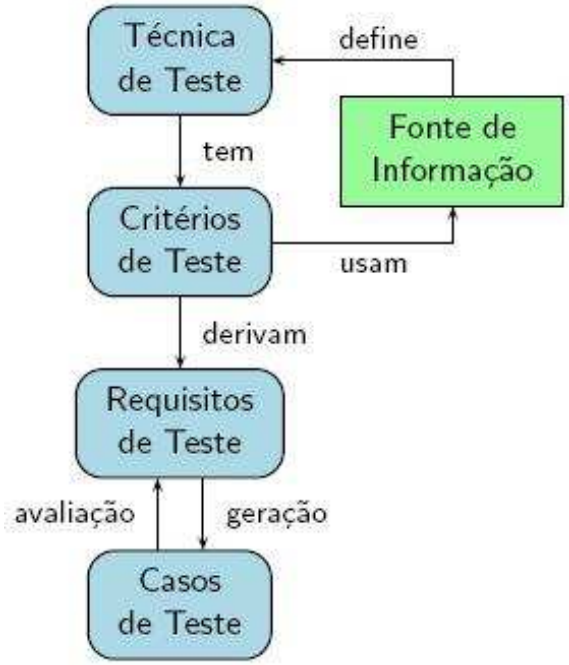

Figura 3.1: *Relações entre técnica, critério, requisito e casos de teste [23].*

3.2 Técnica *ad hoc*

Também conhecida como teste aleatório ou sem critério especificado, talvez seja a mais utilizada pelos profissionais engenheiros de software que não focaram seus estudos em outras técnicas de teste. Essa técnica não é estabelecida em determinado ponto do programa, quem diz quais aspectos observar é o próprio testador. Se ele quiser abrir o código para avaliá-lo ele o faz, caso queira encarar o software como uma caixa-preta e avaliar apenas os resultados com a especificação do software, tudo bem. Também é possível mesclar as estratégias que julgar necessárias para revelar a maior quantidade de defeitos possíveis. Ou seja, vale tudo para revelar os defeitos.

Como vantagens é possível dizer que pouco esforço é gasto para treinamento sobre essa técnica talvez por ser mais simples de aprender, não há ferramentas complexas. Como desvantagem é possível citar a dependência do conhecimento do profissional que realiza o teste, a qualidade do teste também fica totalmente dependente da criatividade do profissional em revelar defeitos, não há indicação ao testador do momento de concluir os testes, ele pára quando achar que já testou o suficiente para confiar no software testado. Isso é ruim, pois ele pode levar uma hora para fazer isso ou muitos anos. Todos quando começam a programar e não conhecem outras técnicas partem desta, e observam que não é produtivo e nem livra o software totalmente dos defeitos. Muitas empresas também adotam apenas essa técnica como única para garantir a qualidade do produto de software.

Por outro lado, sabe-se que a intuição do testador é de grande importância e ela pode ser explorada livremente com essa técnica de teste.

3.3 Técnica Funcional

A partir da especificação são gerados diversos artefatos incluindo o próprio software. A técnica funcional é voltada para a criação de casos de testes a partir de informações contidas nos requisitos do software em produção.

Conhecida também como teste caixa-preta, essa técnica baseia-se na especificação do software. As únicas informações que se conhece do sistema são: 1) sua especificação; 2)as entradas de dados ao sistema; e 3) as saídas esperadas. Por meio dessa técnica são criados a partir dos requisitos, os conhecidos casos de testes vide na Seção 2.2. Basicamente contém as entradas de informações do sistema e as saídas esperadas que o sistema produza. A tarefa do testador é conduzir o sistema como uma caixa-preta, da qual não se conhece o funcionamento interno, e avaliar se o que é produzido corresponde ao esperado.

Os seguintes passos podem ser utilizados para realizar o teste de um software utilizando a técnica funcional: 1) Analisar a especificação de requisitos; 2) Identificar a partir da especificação entradas válidas e inválidas para o sistema; 3) Determinar as saídas esperadas para cada conjunto de entrada; 4) Produzir e documentar o caso de teste; 5) Executar o caso de teste conforme projetado; 6) Capturar as saídas produzidas pelo software e comparar com as saídas esperadas; e 7) Documentar a comparação entre o resultado esperado e o produzido.

Vale ressaltar que esse tipo de teste pode ser aplicado em qualquer fase do desenvolvimento de software, ou seja, pode ser utilizado no teste de unidade, integração ou sistema. E que quando o cliente receber o software ele provavelmente irá utilizar essa técnica para avaliar e emitir o aceite ao produto.

Como característica do teste funcional é possível citar que todo o projeto de caso de teste depende de uma especificação de requisitos feita com qualidade. Caso isso não aconteça é muito provável que o conjunto de casos de teste não tenha qualidade e que o software não revele a maioria de seus defeitos ou apresente supostos defeitos que não o são. Também não é possível garantir que partes essenciais ou críticas do programa foram executadas, ou seja, se a função complexa que deveria ser executada foi utilizada pelo software para computar a saída esperada. Sua maior eficácia é para determinar se uma funcionalidade do software não está presente, ou seja, está na especificação de requisitos e não foi construída no software. Em alguns casos também pode identificar funcionalidades presentes no software que não foram descritas em sua especificação.

Outro fator importante de avaliar é a quantidade de possibilidades de entrada que um software pode ter, ou seja, infinitas combinações de entrada de dados podem produzir infinitas saídas. Buscando aperfeiçoar o conjunto de casos de testes produzidos para serem executados no programa, a técnica funcional dispõe de alguns critérios que nos permitem reduzir a quantidade de casos de testes produzidos, ou seja, reduzindo a quantidade de combinações. Os critérios mais conhecidos para o teste funcional são o Particionamento de Equivalência e a Análise de Valor Limite. Outros menos conhecidos são: a tabela de decisão, o teste de todos pares, o teste de transição de estado, o teste de análise de domínio e o teste de caso de uso.

Especificamente sobre os critérios de particionamento de equivalência e análise de valor limite será dada maior atenção, pois ambos fazem parte do escopo desse trabalho.

3.3.1 Particionamento de Equivalência

Um conjunto de valores, chamado de domínio, forma as possíveis entradas para realizar determinado teste funcional. O particionamento de equivalência visa dividir o domínio de tal forma que não seja necessário executar o teste funcional com todos os valores pertencentes a este domínio. Definindo assim um caso de teste que descobre classes de defeitos com um menor número de casos de teste. Segundo Pressman [53] as seguintes diretrizes devem ser utilizadas para definir um caso de teste: 1) Se uma condição de entrada do programa especifica um intervalo, uma classe de equivalência válida e duas inválidas devem ser definidas; 2) Se uma condição de entrada do programa exige um valor específico, uma classe de equivalência válida e duas inválidas devem ser definidas; 3) Se uma condição de entrada do programa especifica o membro de um conjunto, uma classe de equivalência válida e uma inválida devem ser definidas; e 4) Se uma condição de entrada do programa é booleana, uma classe de equivalência válida e uma inválida devem ser definidas. Após serem projetados e construídos os casos de teste obedecendo ao critério de particionamento de equivalência eles devem ser executados da mesma forma que foi descrito no teste funcional padrão. Também pode ser utilizado o critério de particionamento para as saídas ou seja, as saídas analisadas são apenas as que fazem parte do conjunto de equivalência definido e que está de acordo com o conjunto de entrada.

3.3.2 Análise de Valor Limite

Em junho de 1996 o CNES *(Centre National D´Etudes Spatiales)* e a ESA *(European Space Agency)* lançaram o foguete *Ariane 5*. Embora parecesse tudo em perfeitas condições para o lançamento, segundo consta nos laudos, uma falha de *overflow*

o causou a autodestruição do foguete após 37 segundos da decolagem, ocasionando um prejuízo de aproximadamente 500 milhões de dólares [51].

Essa catástrofe poderia ter sido evitada caso os engenheiros do software que controlavam o Ariane 5 tivessem executado o teste de Análise de Valor Limite. Esse critério de teste busca para determinada entrada avaliar os valores que estão nas fronteiras superiores, inferiores o próprio limite do software. Por exemplo, se um determinado procedimento possui um intervalo de entrada válido, variando entre 0 e 1000, utilizando esse critério deveriam ser criados casos de testes que executassem o software com os valores 999, 1000 e 1001. Ou seja, uma unidade exatamente abaixo, o próprio valor e uma unidade exatamente acima do valor de fronteira. De acordo com a característica do software é possível determinar a precisão que pode ser utilizadas para o teste.

Linkman et al. definem o teste funcional sistemático como a união dos dois critérios funcionais vistos anteriormente, ou seja, a junção do Particionamento de Equivalência com a Análise de Valor Limite [56].

3.4 Técnica Estrutural

O software é composto de algoritmos, esses são compostos de comandos como laços de repetição, condições, declaração e inicialização de variáveis, computação de valores, dentre outros. Um engenheiro de software projetou um algoritmo que não é utilizado pelo software, ou um trecho de outro algoritmo não consegue ser executado, se um desses exemplos ou algo semelhante acontece, dificilmente o cliente que apenas executa o software fica sabendo disso. Aliás, ele só descobre casos como este quando um membro de sua equipe inicia a manutenção no software. Isso acontece quando não é utilizado o teste estrutural no software.

O teste estrutural ou caixa-branca define seus requisitos de teste com base na informação codificada no software. Ele foca exatamente em verificar a estrutura dos comandos. Quando um comando não é executado pelo menos uma vez não é possível garantir que ele está livre de defeito.

A cobertura de comandos é um artifício muito importante para os engenheiros de software, pois por meio dela é possível garantir qual o percentual do software escrito foi exercitado, pelo menos uma vez, pelos casos de teste. Hoje talvez não seja um artifício muito utilizado pelos desenvolvedores de software, mas se fosse bem utilizado poderia revelar defeitos que o cliente provavelmente irá encontrar. Se a cobertura de um programa atingir 100% dos comandos com certeza o cliente irá ficar mais tranqüilo em relação à utilização do software do que se ele adquirisse um software que obteve 50% de cobertura apenas de comandos exercitados.

Uma seqüência de comandos é conhecida como caminho, em um algoritmo existem diversos caminhos. Para garantir que um programa está correto todos os seus caminhos deveriam ser executados. Porém os caminhos podem ser muito extensos ou infinitos, isso faz com que percorrer todos os caminhos seja uma tarefa, em geral, inviável. Além dos caminhos existem outras estruturas que podem ser verificadas por meio do teste caixa-branca, os critérios do teste estrutural cuidam de todas essas estruturas.

Outra característica do teste estrutural é que o mesmo, em geral, não revela defeitos sensíveis a dados, por exemplo, dada a computação da expressão $y = x * 2$ (multiplicação), se o valor de x for igual a 2, y terá como resultado o valor 4, e esse valor era o esperado para validar o teste. Embora pareça estar correto, houve um engano na escrita do código e a expressão deveria ser $y = x^2$ (potenciação), o valor produzido continuaria sendo 4, ou seja, o valor esperado. No entanto, o teste estrutural nem sempre consegue revelar esse defeito. Os seguintes passos podem ser utilizados para a realização do teste caixa-branca: 1) Analisar a implementação (código) do produto de software; 2) Escolher os caminhos da implementação; 3) Selecionar entradas específicas para exercitar os caminhos escolhidos; 4) Determinar a saída esperada pelo software; 5) Documentar os casos de testes; 6) Executar o caso de teste; 7) Comparar o resultado obtido da execução com o esperado no caso de teste; e 8) Gerar um relatório documentando o resultado do caso de teste.

Uma característica muito importante do teste estrutural é que ele permite garantir que trechos essenciais do programa testado foram exercitados e isso é possível de ser analisado por meio da cobertura de código. Além de auxiliar na detecção de erros de lógica de programação.

O teste estrutural pode ser dividido em dois tipos: 1) Teste estrutural baseado em fluxo de controle; e 2) Teste estrutural baseado em fluxo de dados. O primeiro foca nos comandos e na estrutura estática do programa, o segundo trata da estrutura baseada nos dados que as variáveis assumem.

Alguns dos critérios mais conhecidos do teste estrutural baseado em fluxo de controle são: 1) Critério *Todos-Nós*; 2) Critério *Todas-Arestas*; e 3) Critério *Todos-Caminhos*. O teste estrutural baseado em fluxo de dados, em geral, é dividido nos critérios: 1) Critério *Todas-Definições*; 2) Critério *Todos-Usos*; e 3) Critério *Todos-Potenciais-Usos*. Alguns deles serão detalhados logo a seguir.

O teste estrutural possui as limitações provenientes do teste de software e mais duas que seguem: 1) se um programa não implementa algumas funções, não existirá um caminho que corresponda aquela função, assim, nenhum requisito de teste estrutural é extraído e, e consequentemente, nenhum dado de teste será requerido para exercitá-lo; e 2) o programa pode apresentar, coincidentemente, um resultado correto para um dado particular de entrada, satisfazendo o requisito de teste e não revelando a presença de

um defeito, entretanto, se escolhido outro dado de entrada, o resultado obtido poderia expor o defeito. Mesmo com suas limitações o teste estrutural pode ser utilizado como complemento a outras técnicas, ajudando a identificar caminhos ainda não percorridos que possivelmente contenham defeitos.

3.4.1 Grafo de Fluxo de Controle

O GFC (Grafo de Fluxo de Controle) ou Grafo do Programa é uma notação visual simples para a representação do fluxo de comandos de um procedimento ou programa. Ele é muito útil para abstrair a estrutura de controle interna da unidade em teste e é uma das representações mais utilizadas pelos critérios estruturais, para identificar rapidamente os caminhos de execução de uma implementação de um produto em teste. Um caminho é uma seqüência de execução de comandos que se inicia em um ponto de entrada e termina em um ponto de saída de uma unidade do programa em teste.

Para cada comando do programa existe uma estrutura visual compatível. Os comandos fazem parte do programa e conseqüentemente do GFC, exemplos de comandos na linguagem *Java* são *if* (Condição), *do* (Laço), *while* (Laço), *for* (Laço), *switch* (Condição Múltipla), dentre outros. Cada estrutura visual que representa um comando pode ser composta por duas partes: 1) Nós - representam uma ou mais instruções executadas em seqüência e 2) Arestas ou Arcos - representam os fluxos e as ligações entre dois nós. Na Figura 3.2 são demonstrados exemplos da representação usada pelo GFC.

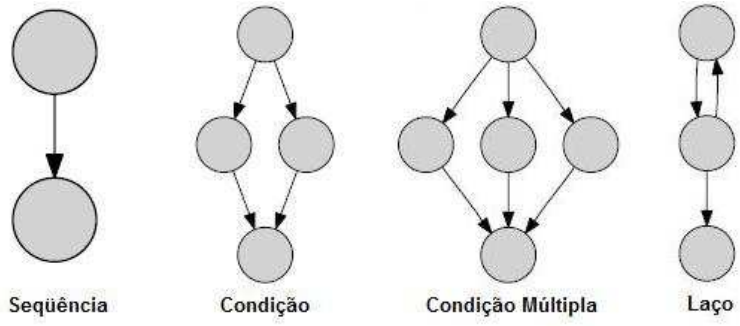

Figura 3.2: *Exemplos de estruturas básicas do GFC.*

Um GFC de um programa é composto pela combinação de estruturas básicas ilustradas na Figura 3.2. Cada instrução do programa representa um nó do grafo e suas ligações são representadas pelas arestas. A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco. A representação de um programa P como um GFC é dada pelo grafo orientado $G = (N, E, s)$, sendo N é o conjunto de nós, E é o conjunto de arestas e s é o nó inicial de entrada. O nó de saída do programa faz parte do conjunto dos nós representado por N.

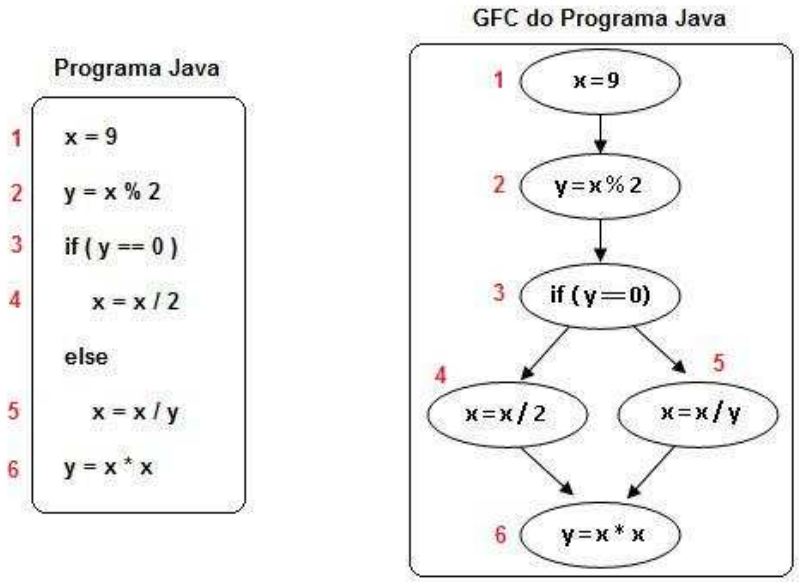

Figura 3.3: *Exemplo de GFC.*

Alguns comandos são obrigatoriamente executados em seqüencia e não há como dividir sua execução, como por exemplo, as instruções que correspondem as linhas 1 e 2 do quadro Programa *Java* da Figura 3.3 sempre serão executadas seqüencialmente, dessa forma é possível fazer uma representação reduzida do GFC demonstrado na Figura 3.3, esse novo grafo reduzido é ilustrado na Figura 3.4, no qual é possível observar quais linhas seqüenciais se transformaram em uma única representação de um nó.

Figura 3.4: *Exemplo de GFC Reduzido.*

As linhas 1 e 2 da Figura 3.3 se transformaram em um único nó do grafo reduzido ilustrado na Figura 3.4. Esse é um exemplo simples de redução, em programas reais a redução de nós é bastante significativa. Outro exemplo de redução de GFC é apresentado na Figura 3.5, cujo grafo *(e)* é a representação reduzida do grafo do método apresentado em *(d)*. Ainda na Figura 3.5 é possível observar o código fonte de um método do programa

(a), no item *(b)* o *bytecode* do respectivo método e em *(c)* a tabela que mapeia as linhas do código fonte original para os respectivos marcadores de linhas no *bytecode*.

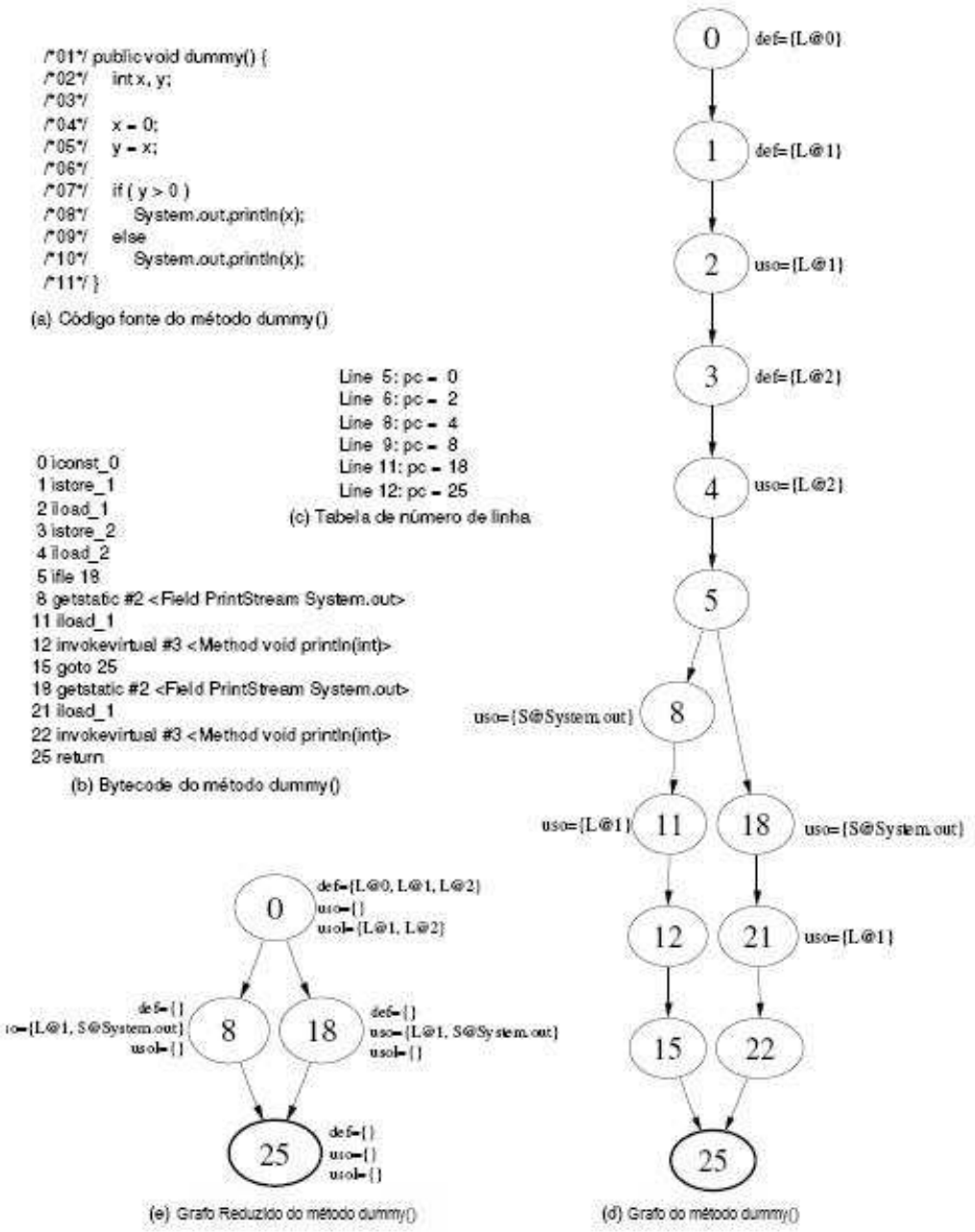

Figura 3.5: *Exemplo de GFC Reduzido adaptado [62].*

3.4.2 Critérios de Fluxo de Controle

Os blocos de controle ou comandos são utilizados pelos critérios que fazem parte do fluxo de controle. Esses critérios levam em consideração a sequência de execução dos comandos de um determinado programa. Os mais conhecidos são [32] [53]: 1) *Todos-Nós* - exige que seja executado pelo menos uma vez cada vértice do GFC, ou seja, cada

comando existente no software seja executado pelo menos uma vez; 2) *Todas-Arestas* - exige que cada desvio de execução (aresta do GFC) seja utilizada pela execução do programa pelo menos uma vez, ou seja, obrigando a execução de todas condições que geram mudanças de fluxos no programa pelo menos uma vez; e 3) *Todos-Caminhos* - exige que todos os caminhos do programa sejam executados pelo menos uma vez.

Os critérios estão definidos na ordem de dificuldade de satisfação, ou seja, é mais fácil satisfazer o critério *Todos-Nós* do que o critério *Todas-Arestas*, e mais difícil satisfazer o critério *Todos-Caminhos*. Através da teoria dos grafos é possível provar que, se o critério *Todas-Arestas* for satisfeito o critério *Todos-Nós* também será satisfeito, a recíproca não é verdadeira. Ou seja, se existem casos de teste que passam por todas as arestas de um grafo, consequentemente irá passar por todos os nós. Por outro lado, se existem casos de testes que percorrem todos os nós de um grafo, não necessariamente passam por todas arestas. O mesmo raciocínio é utilizado para satisfazer o critério *Todos-Caminhos*, que se satisfeito também satisfaz os critérios *Todos-Nós* e *Todas-Arestas*. Esse relacionamento entre os critérios chamados de hierarquia de inclusão dos critérios de fluxo de controle, ou seja, o critério *Todos-Caminhos* incluí o critério *Todas-Arestas* que por sua vez, inclui o critério *Todos-Nós*, demonstrado na Figura 3.6.

Figura 3.6: *Hierarquia de Inclusão dos Critérios de Fluxos de Dados.*

3.4.3 Critérios de Fluxo de Dados

Mesmo para alguns programas pequenos, o teste baseado no fluxo de controle não é suficiente para revelar a presença de defeitos simples e triviais, embora esse teste possa garantir que a estrutura do programa foi executada. Dessa forma foi necessário buscar outra maneira de auxiliar na descoberta de defeitos utilizando o teste caixa-branca, surgiram os critérios baseados em fluxo de dados. Ural [26] defende a ideia de que os critérios baseados em fluxo de dados são mais adequados para revelar certos tipos de defeitos, como os defeitos computacionais (computação de valores no programa).

A fonte de informação utilizada pelo critério de fluxo de dados como requisito de teste é o fluxo de informações, ou seja, iterações que envolvam as definições de variáveis e suas referências no programa. Assim os casos de testes são projetados a partir das associações entre a definição de uma variável e seus possíveis usos subsequentes. Rapps

e Weyuker propuseram na década de 80 alguns conceitos e definições que apóiam o teste estrutural baseado no fluxo de dados. O Grafo Def-Uso foi um deles, ele é uma extensão do conhecido GFC. Nele são acrescentadas informações sobre o fluxo de dados do programa, criando assim, associações entre pontos do programa por meio das definições de variáveis e as referências ou usos dessas variáveis.

O Grafo Def-Uso é criado a partir do GFC associando a cada nó *i* os conjuntos c-uso(i) = variáveis com c-uso global no nó *i* e def(i) = variáveis com definições globais no nó *i*, e a cada arco (i, j) o conjunto p-uso(i, j) = variáveis com p-usos no arco (i, j). Dois conjuntos foram definidos: dcu(x, i) = nós j, tal que $x \in$ c-uso(j) e existe um caminho livre de definição com respeito a (c.r.a.) x do nó i para o nó j e dpu(x, i) = arcos (j, k), tal que $x \in$ p-uso(j, k) e existe um caminho livre de definição c.r.a. x do nó i para o arco (j, k).

Partindo dessas definições, Rapps e Weyuker propuseram os seguintes critérios: 1) *Todas-Definições* - requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um c-uso ou p-uso; 2) *Todos-Usos* - requer que todas as associações entre uma definição de uma variável e seus subsequentes usos (c-usos ou p-usos) sejam exercitados por pelo menos um caminho livre de definição, ou seja, um caminho em que a variável não é redefinida; e 3) *Todos-Du-Caminhos* - requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos sejam exercitados por todos os caminhos livres de definição e livres de laço que cubram essa associação.

Segundo os estudos de Maldonado [43] nos anos 90, definiu o critério denominado Potenciais-Usos, que requer que as associações independentemente da ocorrência explicita de uma referência (uso) a uma definição de variável. Ou seja, se um uso dessa definição pode existir (há um caminho livre de definição até um certo nó ou arco, isto é, um potencial-uso), a potencial-associação entre a definição e o potencial-uso é caracterizada e eventualmente requerida. Mais detalhes sobre os critérios estruturais e alguns critérios não citados nesse trabalho, como *Todos-Potenciais-Usos/Du* e *Todos-Potenciais-Usos-Du-Caminhos* podem ser vistos em [23]. A hierarquia entre os critérios de fluxo de controle e de fluxo de dados pode ser demonstrada na Figura 3.7.

3.4.4 Critérios Baseados em Defeitos

As técnicas funcional e estrutural buscam criar casos de testes que melhor representem as principais características do domínio de entrada do programa a ser testado. Por outro lado os critérios baseados em defeitos são utilizados defeitos típicos da implementação do software para derivar os requisitos de teste.

Um dos critérios mais conhecidos dessa categoria é o teste de mutação que

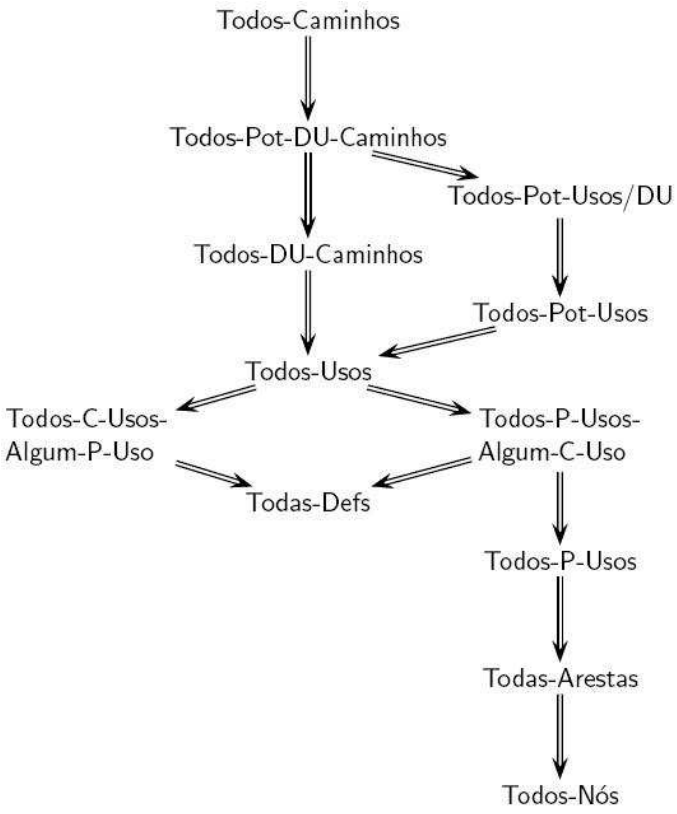

Figura 3.7: *Hierarquia de Critérios de Teste Estrutural.*

é bastante estudado por pesquisadores de todo o mundo devido sua efetividade para encontrar defeitos. Em [23], Delamaro relata que eventos internacionais foram propostos exclusivamente para tratar do assunto, um deles aconteceu em San Jose, nos EUA em 2000.

Essa nova abordagem para encontrar defeitos em software, baseia-se na "hipótese do programador competente" que produz um software correto ou muito próximo do correto. Partindo dessa hipótese é possível caracterizar uma série de defeitos comuns que acontecem na criação de um software. Os mais comuns são, a troca de operadores lógicos, aritméticos, dentre outros.

Dessa forma são criados os chamados "mutantes" do programa, ou seja, programas baseados no original que possuem "alterações" em seu conteúdo. Por exemplo, um trecho do programa compara com sinal de $>$ (maior) duas variáveis, um mutante é criado trocando o operador $>$ (maior) por $<$ (menor) e assim sucessivamente até completar as combinações mais prováveis de defeitos (identificados por meio de observações, heurísticas, ou estudos experimentais).

Após os mutantes serem criados, casos de testes são projetados a fim de testar o programa e seus mutantes. Assim, dados de entrada exercitam os programas e por meio de seus resultados é avaliado se o mutante continua "vivo", ou seja, pode ser executado um novo caso de teste ou se ele está "morto", ou seja, não é compatível com

o original. Essa comparação é realizada pois, os programas que apresentam mesma saída para determinado caso de teste indicam que há uma possibilidade do programa original conter erros, diferentemente quando um programa mutante apresenta saída diferente da mostrada pelo programa original. Por meio desses dados, com o total de mutantes que continuam vivos e dos mutantes mortos possibilita-se calcular o *escore* de mutação através da fórmula: $ms(P,T) = \frac{DM(P,T)}{M(P)-EM(P)}$, sendo: $DM(P,T)$ é o número de mutantes mortos pelo conjunto de casos de teste T; $M(P)$ é o total de mutantes gerados a partir do programa P; e $EM(P)$ é o número de mutantes gerados que são equivalentes a P. Caso o *escore* de mutação $ms(P,T)$ for $1,0$ ou muito próximo de $1,0$, então pode-se encerrar o teste e considerar T um bom conjunto de caso de teste para P.

É possível notar que o teste de mutação é muito trabalhoso e para fazê-lo com eficiência e maior rapidez é necessária uma ferramenta que apóie pelo menos a geração de mutantes e a execução dos casos de teste. Algumas ferramentas foram propostas como, por exemplo, Proteum [22] e Mothra [16]. Mais detalhes sobre algumas ferramentas e também sobre o teste de mutação podem ser vistos em [23].

3.5 Ferramentas de Teste

O ser humano desde sua existência sempre desempenhou atividades, algumas racionais, outras específicas e outras repetíveis. Porém sua característica de raciocinar trouxe uma grande vantagem que veio à tona na Revolução Industrial com o auxílio de máquinas que permitiam ao homem ter maior produtividade, um menor tempo e em geral com menor custo.

Com o software, não é diferente, diversas ferramentas são criadas a cada dia buscando auxiliar nas mais diversas atividades de construção de software. Dentre as mais comuns pode-se citar os compiladores, os interpretadores, as ferramentas de ambiente de desenvolvimento, servidores de aplicação, dentre outros.

O teste de software também passou por determinado tempo sem nenhuma ferramenta que auxiliasse o engenheiro de software a construí-lo e testá-lo. Pressman relata que essa dificuldade em criar ferramentas de apoio ao teste de software só começou a ser priorizada após os anos 70 e iniciaram-se os esforços para aprimorar a descoberta de erros no desenvolvimento de software. “Durante muitos anos, nossas únicas defesas contra erros de programação eram o projeto cuidadoso e a inteligência própria do programador.” [53].

A partir dessa data com o avanço tecnológico e a prospecção de potencialidade dos computadores e de seus software, iniciou o lançamento de diversas ferramentas que a apoiavam o teste de software. Como a tecnologia não para de crescer, a cada dia surgem

novas linguagem, arquiteturas, conceitos e a atividade de teste sempre busca acompanhar essas melhorias na computação.

Muitas ferramentas surgiram, cada uma com seu propósito, linguagem, arquitetura ou critério especifico. Como por exemplo, desde 1989 a ferramenta *POKE-TOOL* [13] apóia alguns critérios de teste estrutural (vide Seção 3.4) no teste de unidade em programas escritos na linguagem C [57]. Outra ferramenta disponibilizada em 1990, voltada para o teste baseado em mutantes (vide Seção 3.4.4), foi a Proteum [22] também para programas em C.

Para testes na linguagem *Java* pode-se contar com algumas ferramentas de apoio como o *JUnit Framework* [11] que além do código aberto vem sendo amplamente utilizado para escrever e executar conjuntos de testes unitários. Outra ferramenta bastante interessante que é um dos focos deste estudo, também voltada para *Java*, é a JaBUTi *(Java Bytecode Understanting Testing)* [64] que apóia o teste estrutural e como principal característica não necessita do código-fonte, informações mais detalhadas podem ser conferidas na Seção 3.5.1.

Para tecnologias mais de ponta como as empregadas na maioria dos dispositivos móveis (celulares, *PDAs* e outros) também existem ferramentas que apóiam o teste nesses dispositivos, uma delas é uma nova versão da JaBUTi adaptada exclusivamente para testes com essas características, que é a JaBUTi/ME *(Java Bytecode Understanting Testing /Micro Edition)* [24].

Existem centenas de ferramentas disponíveis que apóiam a atividade de teste de software, uma comparação entre algumas delas, voltadas para o teste de programas OO, e seus recursos mais importantes podem ser conferidos na Tabela 3.1, extraída de [63].

3.5.1 JaBUTi

Algumas atividades da computação, se executadas manualmente podem ser caras e sujeitas a erros humanos. As ferramentas de apoio ao teste de software visam reduzir o risco de erros humanos nessas atividades e proporcionam maior produtividade ao teste de software. Além disso, são de fundamental importância para a realização de estudos comparativos entre os critérios de teste, para a transferência tecnológica e a atividade de treinamento, ensino e capacitação. Por outro lado novas tecnologias surgem a todo momento, e podem representar desafios para a área de teste de software.

A JaBUTi *(Java Bytecode Understanting Testing)* [64] é um ambiente completo para o entendimento e para o teste de programas e componentes desenvolvidos em *Java* [41]. Voltada para o teste estrutural, apóia diversos critérios estruturais para análise de cobertura, possui também um conjunto de métricas estáticas de complexidade de

Tabela 3.1: *Características apresentadas pelas ferramentas de teste OO (Adaptada de [63]).*

Ferramentas de teste de software	**Teste de unidade**	**Teste de integração**	**Teste de sistema**	**Critérios funcionais**	**Critérios de fluxo de controle**	**Critérios de fluxo de dados**	**Critérios baseados em mutação**	**Linguagem suportada**	**Exigência de código fonte**	**Atividade de depuração**	**Teste de regressão**
PiSCES	✓				✓			Java	✓		
SunTest	✓		✓		✓			Java	✓		
xSuds Toolsuite	✓		✓		✓			C/C++	✓	✓	✓
JProbe Developer Suite	✓		✓		✓			Java	✓	✓	
Panorama C/C++	✓				✓			C/C++	✓		
Panorama for Java	✓				✓			Java	✓		
TCAT/Java	✓				✓			Java	✓		
JCover	✓				✓			Java	✓		
Parasoft CodeWizard								C/C++	✓		
Parasoft C++ Test	✓		✓	✓	✓			C/C++	✓		✓
Parasoft Insure++								C/C++	✓	✓	
ProLint								C/C++	✓		
Rational PureCoverage	✓		✓		✓			C++/Java	✓		
Rational Purify			✓					C/C++	✓	✓	
JUnit	✓			✓				Java			
CTB	✓		✓	✓				Java/C/C++			
Parasoft JTest	✓		✓	✓	✓			Java	✓		✓
Glass JAR Toolkit	✓		✓	✓	✓			Java			
Object Mutation Engine		✓					✓	Java	✓		
Ferramenta de [15]		✓					✓	Java	✓		
Ferramenta de [42]		✓					✓	Java	✓		
Mutation Testing System		✓					✓	Java	✓		
JaBUTi	✓				✓	✓		Java		✓	

programas e implementa algumas heurísticas de particionamento de programas que visam auxiliar a localização de defeitos.

Dentre os critérios estruturais de cobertura dos programas *Java*, a ferramenta JaBUTi dá suporte aos seguintes: critérios de fluxo de controle: *Todos-Nós* e *Todas-Arestas*; critérios de fluxo de dados: *Todos-Usos* e *Todos-Potenciais-Usos*. Devido ao fluxo de execução de programas *Java* terem a característica de execução de fluxo normal ou de fluxo de exceção, os critérios apoiados pela JaBUTi também tiveram que ser adaptados para essa linguagem, tornando-se os seguintes: *Todos-Nós*$_{ei}$, *Todos-Nós*$_{ed}$, *Todas-Arestas*$_{ei}$, *Todas-Arestas*$_{ed}$, *Todos-Usos*$_{ei}$, *Todos-Usos*$_{ed}$, *Todos-Potenciais-Usos*$_{ei}$ e *Todos-Potenciais-Usos*$_{ed}$. Do qual o sufixo *ei (exception-independence)* trata do fluxo

de execução normal e o sufixo *ed (exception-dependence)* faz a análise baseada no fluxo de execução quando ocorre uma exceção nos programas *Java*.

Para obter a cobertura a ferramenta faz a instrumentação do programa a ser testado. A instrumentação é a inclusão de trechos de código no programa original em pontos específicos que geram as informações do *trace* ou rastro de execução para serem analisadas posteriormente, como pode ser ilustrado na Figura 3.8. A ferramenta JaBUTi é implementada na linguagem *Java* e pode ser executada em qualquer computador que possui a *Java Virtual Machine*. Assim a ferramenta desfruta da portabilidade que *Java* fornece, podendo ser executada em qualquer sistema operacional como Linux, Windows, Unix e outros.

A Figura 3.8 mostra a estrutura da arquitetura de funcionamento da JaBUTi e mostra que, por meio do programa *Java* que se deseja testar, a JaBUTi faz uma análise estática da estrutura do programa criando o grafo Def-Uso, após a criação do grafo Def-Uso os requisitos de teste são gerados com base nos critérios apoiados pela mesma. Após isso, é feita a instrumentação do programa *Java* que permite a geração do arquivo *trace* durante sua execução. Ao executar o programa instrumentado é armazenado no arquivo de *trace* as informações pelas quais o programa passou durante a execução e essas informações são analisadas juntamente com os requisitos de teste, assim verificando quais requisitos foram satisfeitos e mostrando as estatísticas de cobertura de acordo com o critério escolhido. A Figura 3.9 mostra um exemplo de grafo Def-Uso e representação de cobertura.

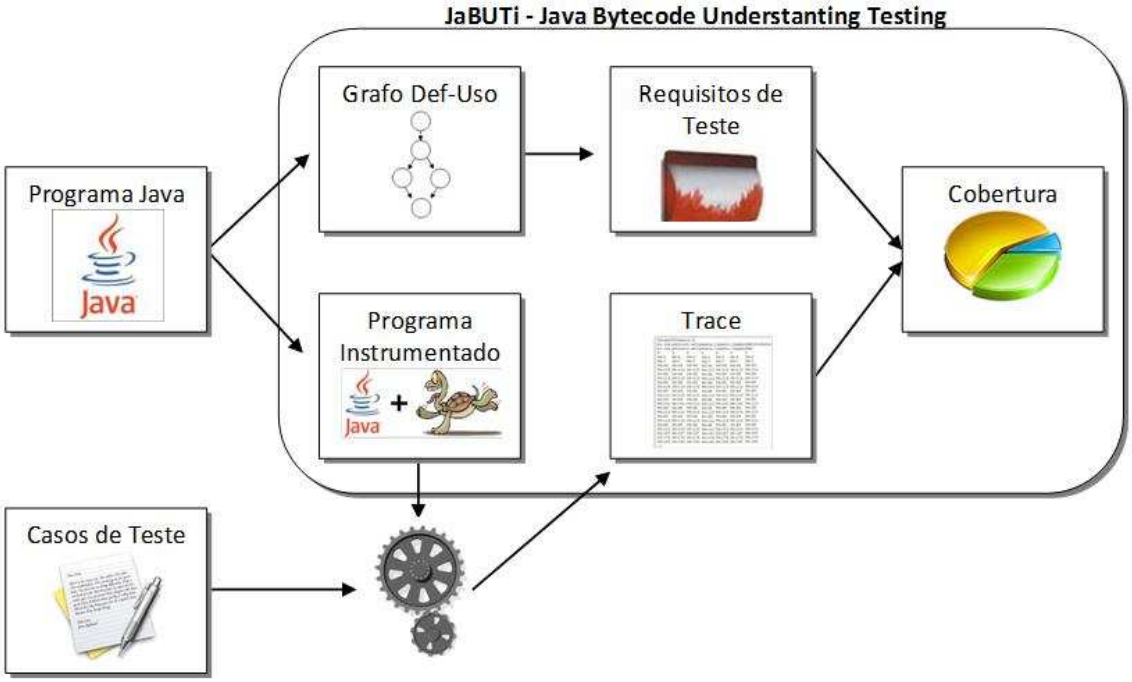

Figura 3.8: *Operações da Ferramenta JaBUTi.*

Além da portabilidade, a JaBUTi possui outra característica marcante, a ferra-

menta realiza a instrumentação do programa e cria o grafo Def-Uso sem a necessidade do código fonte, ela utiliza o próprio código objeto *(bytecode)* da JVM para inserir as informações da instrumentação. Caso o programa possua o código fonte, fica mais fácil para o testador pois ela fornece a opção de visualizar a cobertura dos comandos no próprio código fonte, mas caso não esteja disponível ela disponibiliza as mesmas informações via código objeto.

Outra característica bastante interessante da ferramenta é a atribuição de pesos, utilizando o conceito de dominadores e super-bloco [28]. Ela atribui diferentes pesos aos requisitos de teste indicando qual o requisito de teste que, caso coberto, levará a um aumento na cobertura de forma mais rápida. Ou seja, se um caso de teste conseguir cobrir um dos requisitos marcados com maior peso, vários outros requisitos de menor peso também serão cobertos. Informalmente, os pesos correspondem ao número de requisitos de teste que serão cobertos quando um requisito de teste particular é coberto. Na ferramenta a barra de cores define os pesos que vai do branco (peso 0) ao vermelho (peso 7), vide um exemplo de cores e pesos apresentados na Figura 3.9, em escalas de cinza nessa imagem. Informações mais detalhadas podem ser obtidas no manual de usuário da ferramenta [61].

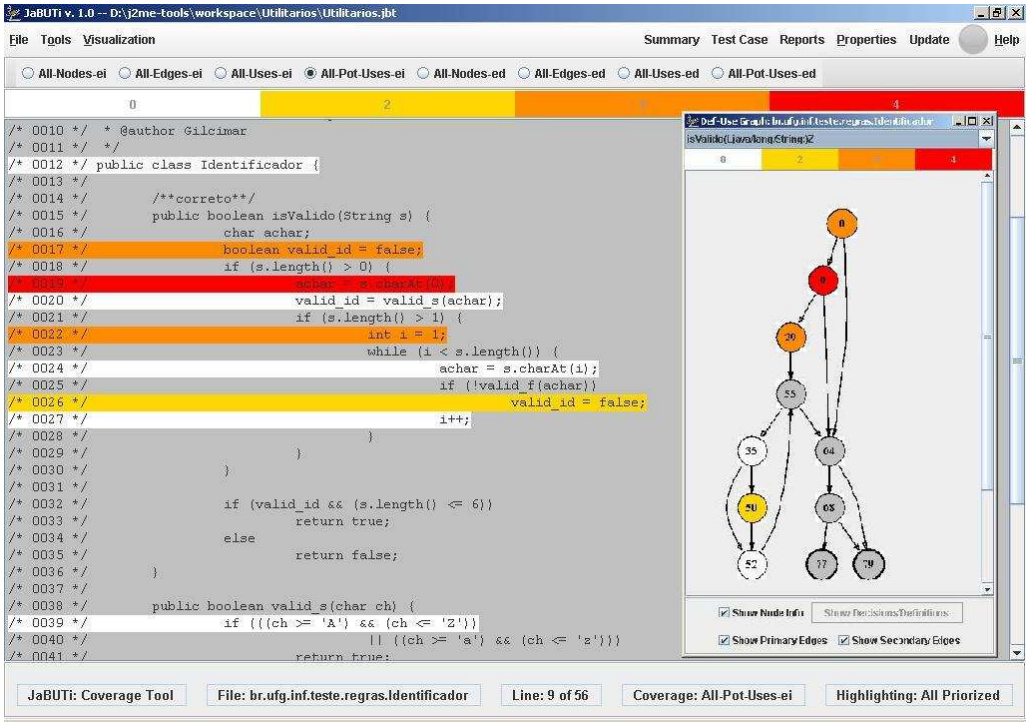

Figura 3.9: *Grafo Def-Uso e Coloração dos Pesos de um programa Java.*

Após a execução dos casos de teste é possível iniciar a análise de cobertura de código do programa. Ao ser realizada essa operação é apresentada uma tela como a demonstrada na Figura 3.10, com o percentual de cobertura por critério. No exemplo

da Figura 3.10, o critério *Todos-Nós*$_{ei}$ alcançou 65% de cobertura, enquanto o critério *Todas-Arestas*$_{ed}$ alcançou 57%. Dessa forma é possível observar que ainda faltam 35% dos comandos para serem cobertos. Ou seja, a ferramenta está informando ao testador que é necessário criar mais casos de testes para alcançar uma melhor cobertura.

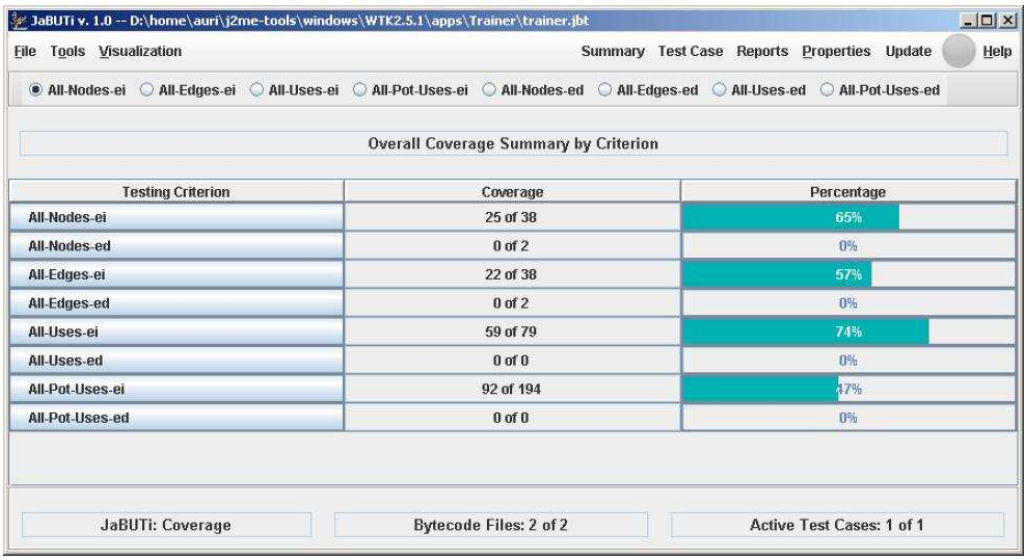

Figura 3.10: *Análise de Cobertura dos Critérios Estruturais por meio da JaBUTi.*

Um fator que merece destaque quando se fala em teste estrutural é a ocorrência de requisitos não executáveis. De acordo com a implementação do software, alguns caminhos ou trechos de caminhos não podem ser executados, isso ocorre devido às condições que o programa foi implementado. Se um comando realiza um teste condicional conforme apresentado no algoritmo 3.1, claramente o *COMANDO 02* nunca será executado. Esse é um exemplo simples com constantes, mas isso pode ocorrer com variáveis e é comum haver esse tipo de requisito não executável quando as condições aninham e tornam-se mais complexas. Mas a JaBUTi é preparada para casos como esses, ela permite que marcar requisitos como não executáveis de modo que os mesmos sejam desconsiderados na análise de cobertura. Caso esse requisito não seja marcado o testador nunca será capaz de conseguir 100% de cobertura no referido critério.

Algoritmo 3.1: Requisito não executável

$a \leftarrow 1$.
$b \leftarrow 2$.
se $(a < b)$ **então**
  COMANDO 01
senão
  COMANDO 02
fim

A JaBUTi também possui recursos para importação de casos de testes criados pelo JUnit Framework [11], sendo possível aproveitar todos os métodos criados com o *framework* e executá-los como casos de testes para que a ferramenta possa realizar a avaliação da cobertura dos casos de teste. Muitos outros recursos fazem parte da ferramenta JaBUTi dentre eles estão os relatórios que a ferramenta emite e as análises métricas que a ferramenta realiza ajudando a compreender a complexidade do software que está sendo testado.

3.5.2 JaBUTi/ME

Como mencionado no Capítulo 2, a tecnologia evolui muito rápido, e as ferramentas também devem se aprimorar para continuarem atendendo aos requisitos do mercado. Com a ferramenta JaBUTi não é diferente, com a popularidade dos dispositivos móveis e dos aplicativos a eles incorporados utilizando *Java ME (Java Micro Edition)* [47], a JaBUTi necessitou evoluir seus recursos. E se adaptou em uma nova versão denominada JaBUTi/ME *(Java Bytecode Understanding and Testing/Micro Edition)* que traz todos os recursos da JaBUTi original só que fazendo análise de código para programas de dispositivos móveis, como celulares e *PDAs*.

Devido às limitações dos dispositivos móveis relacionados ao poder de processamento e armazenamento, na maioria das vezes, não é viável a instalação da JaBUTi no próprio dispositivo. Sendo assim a proposta da JaBUTi/ME é criar um ambiente *"cross-plataform"* que mescla a execução do software instrumentado no próprio dispositivo móvel e analisa a captação de dados no ambiente *desktop* utilizando a tecnologia cliente/servidor, como pode ser demonstrada na Figura 3.11. Desta forma é utilizada uma conexão de rede, com ou sem fio, para envio de informações sobre a execução dos testes para o servidor responsável pela coleta de dados de teste. Caso o dispositivo móvel não esteja disponível a JaBUTi/ME também permite o uso de emuladores em ambiente *desktop* para realização dos testes do software Java ME.

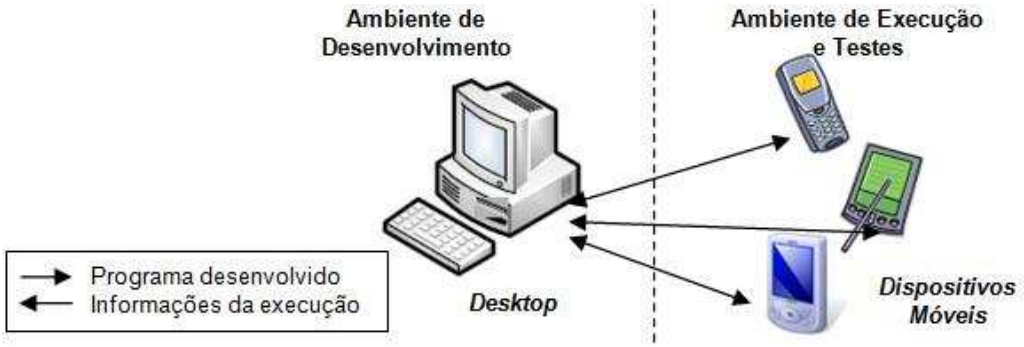

Figura 3.11: *Ambiente cross plataform.*

Por exemplo, alguns dispositivos móveis possuem conexão a rede sem fio, caso essa opção esteja disponível é possível enviar as informações para o servidor por meio de uma conexão *Wi-Fi* ou *Bluetooth*, esse tipo de conexão é mais utilizada quando não há custos na utilização dos serviços de rede. Outro exemplo, um celular que não possui conexão a rede *Wi-Fi* ou *Bluetooth* pode utilizar a opção de envio das informações via rede particular com custo de ligação ou envio de dados. Quando uma opção assim torna-se a única disponível, a JaBUTi/ME possui opções na instrumentação que só abre a conexão com o servidor quando for necessário, otimizando a quantidade de conexões tornando mais barato a utilização da rede particular.

Assim como na JaBUTi, a JaBUTi/ME também utiliza a instrumentação como forma de capturar as informações de execução do programa no dispositivo móvel ou emulador. A diferença é que ao invés de gravar diretamente em arquivo (como faz a JaBUTi), ela faz uma conexão via *socket* e então realiza a transferência das informações do arquivo *trace* para o servidor. Isso ocorre devido à limitação de armazenamento de informações no dispositivo móvel. A JaBUTi/ME possui um conjunto de classes responsável pela comunicação entre o cliente Java ME e o servidor, no momento da instrumentação essas classes são incorporadas ao aplicativo Java ME juntamente com as informações adicionadas no bytecode do programa. As informações são utilizadas para criação das informações do arquivo *trace* e as classes para comunicação com o servidor.

No momento da instrumentação a ferramenta disponibiliza algumas opções para a equipe de teste decidir como será a estratégia de captura de informações. Os fatores que podem ser analisados são: 1) o tipo de meio de comunicação; 2) a quantidade de memória disponível no dispositivo; e 3) a existência ou não de um meio para armazenamento temporário no dispositivo; essas opções dão maior flexibilidade e permite que a ferramenta seja utilizada em uma maior quantidade de dispositivos móveis.

A JaBUTi/ME oferece recursos de interface gráfica para a realização dos passos de instrumentação, esses recursos são mais indicados para o ensino e aprendizagem da ferramenta. No caso de experimentos é recomendado o uso de interface de texto, também conhecido como linha de comando, pois suas opções podem ser mais detalhadas e utilizadas de acordo com a necessidade. A seguir serão apresentados os passos para a instrumentação de um dos programas utilizados no experimento descrito no Capítulo 5.

Dentre os programas escolhidos para fazer parte deste trabalho, um será utilizado para exemplificar como foi realizada a instrumentação por meio da JaBUTi/ME, permitindo que o software seja executado em um dispositivo móvel ou emulador e os dados de sua execução sejam capturados e analisados pela própria ferramenta instalada em um servidor de testes.

Dois passos são necessários para preparar o aplicativo para a execução dos testes, o primeiro é chamado de instrumentação e o segundo de pré-verificação.

Para realizar a instrumentação de um programa, ou seja, prepará-lo para capturar as informações de cobertura, é necessário a execução de um comando da JaBUTi/ME que gera e prepara o pacote a ser executado no dispositivo móvel ou emulador.

As opções estão disponíveis na ferramenta para a instrumentação são: 1) endereço e porta do servidor de teste; 2) identificador do programa a ser testado - o servidor é capaz de gerenciar várias conexões e várias sessões de teste ao mesmo tempo e esse identificador é utilizado para localizar de qual software testado é a informação recebida pelo servidor; 3) quantidade de memória mínima disponível - esse parâmetro diz ao dispositivo quando deve ser criada uma conexão e enviado ao servidor as informações armazenadas temporariamente no dispositivo, logo na sequência é liberado espaço do arquivo gerado e repete-se o processo até finalizar os testes; e 4) manter ou não a conexão - o comportamento padrão é manter uma única conexão durante todo o período de execução dos testes, mas através deste parâmetro é possível informar ao cliente qual momento para a criação da conexão. Os itens 3 e 4 em conjunto podem diminuir custos quando a utilização da rede para envio das informações forem pagas.

O comando para realização da instrumentação é:

```
java -cp <classpath> br.jabuti.device.ProberInstrum
        <opções> <classe base>
```

A classe base é a classe inicial conhecido como *MIDlet* do programa que se deseja testar. As demais opções podem ser analisadas abaixo (os itens entre colchetes são opcionais), mais informações podem ser vistas no manual da ferramenta [61]:

- `[-d]` indica o diretório no qual se encontra o projeto;
- `-p` nome do arquivo de projeto, criado pela JaBUTi/ME;
- `-o` nome do arquivo de saída do comando, nos quais estarão as classes instrumentadas do programa;
- `-name` nome dado ao programa em teste, identifica as informações enviadas ao servidor de teste;
- `-h` endereço do servidor de testes, que é o destino das informações coletadas no dispositivo móvel;
- `[-temp]` especifica o depósito temporário dos dados para envio ao servidor de testes;
- `[-mem]` especifica um limite de memória que deve ser respeitado para armazenamento dos dados, quando a quantidade de memória do dispositivo fica abaixo desse valor, os dados são enviados ou armazenados em arquivo temporário, se não informado é assumido que não existe limite e os dados são enviados assim que a execução finaliza; e

- `[-nokeepalive]` indica se a conexão entre o dispositivo e o servidor de teste não deve ser mantida durante toda a execução do programa. Dessa forma, a cada execução os dados são enviados ao servidor de teste. Esta opção permite economizar a utilização da rede de comunicação que no caso dos dispositivos móveis pode ser cara.

O comando para a realização da pré-verificação é o seguinte e as opções para execução desse comando podem ser (os itens entre colchetes são opcionais):

```
java -cp <classpath> br.jabuti.device.Preverify
        <opções>
```

- `-source` indica o arquivo .jar no qual se encontram os originais do programa a ser implantado no dispositivo móvel.
- `-instr` arquivo .jar criado no passo de instrumentação das classes do programa;
- `[-o]` indica o nome do arquivo .jar de saída, se não informado é utilizado o mesmo do original;
- `[-jad]` especifica o nome do arquivo de descrição do programa, com as informações sobre o pacote gerado; e
- `-WTK` caminho do Wireless Tool Kit que contém o comando para a execução da pré-verificação propriamente dita.

Um exemplo dos comandos vistos anteriormente podem ser analisados abaixo:

```
java -cp ../../jabuti/Jabuti-bin.zip;bin br.jabuti.device.ProberInstrum
        -name bmi -p bmi.jbt -o inst/bmi-tmp.jar  -h 198.162.10.5:2000
        uk.co.marcoratto.j2me.bmi.BmiMIDlet
```

Neste comando foram informados a localização da ferramenta JaBUTi/ME no caminho do sistema operacional, posteriormente o nome e localização da classe que realiza a instrumentação do aplicativo `br.jabuti.device.ProberInstrum`. A opção `-name` é utilizada para a nomeação do programa, `-p` para indicar o nome do projeto a ser criado pela JaBUTi/ME (extensão .jbt), com a opção `-o` é possível indicar o local que será armazenado o resultado da instrumentação. A opção `-h` informa a localização de rede ou endereço *IP (Internet Protocol)* que receberá as informações enviadas pelos dispositivos, e finalmente a classe que inicia o programa juntamente com sua localização. Por meio desse comando o arquivo `bmi-tmp.jar` será criado após a instrumentação e enviará as informações coletadas para o *host* indicado na opção `-h`, que será o servidor de teste com a JaBUTi instalada.

Um software Java ME necessita de mais um passo para tornar-se executável em um dispositivo móvel. Essa fase chama-se de pré-verificação, nela são verificados

os *bytecodes* produzidos, otimizado o tamanho da aplicação e unido tudo em um único arquivo que será implantado no dispositivo móvel.

Após a instrumentação é gerado um pacote contendo todos os arquivos necessários para a execução do aplicativo no dispositivo móvel ou emulador. Antes de executá-lo no dispositivo móvel é necessário executar a chamada pré-verificação com a qual as classes são validadas e compactadas as informações do bytecode para a execução no dispositivo móvel. Depois de recebidas as informações da execução dos casos de testes dos programas testados a JaBUTi/ME retoma seu papel original, analisando as coberturas e disponibilizando informações nos relatórios como apresentado na Figura 3.10.

Para esse passo, a JaBUTi/ME também oferece uma interface texto para realização dessa atividade, vide o comando abaixo:

```
java -cp ../../../jabuti/Jabuti-bin.zip br.jabuti.device.Preverify
     -source ../deployed/bmi.jar -instr bmi-tmp.jar -o bmi.jar
     -jad bmi.jad -WTK D:/j2me-tools/windows/WTK2.5.1
```

Neste comando é informado primeiramente a localização da JaBUTi/ME e posteriormente o nome da classe responsável pela atividade de pré-verificação `br.jabuti.device.Preverify`, a opção `-source` indica a localização dos fontes do programa (este comando é opcional). Na opção `-instr` é informado o arquivo resultante da instrumentação do programa, a opção `-o` indica o nome do arquivo criado para implantação no dispositivo móvel, a opção `-jad` indica a localização do arquivo de descrição do software móvel (necessário para todos aplicativos móveis) e por último a opção `-WTK` indica a localização do *Wireless Tool Kit* responsável pela real pré-verificação do aplicativo.

Um outro passo necessário é a preparação do servidor de teste, esse passo é responsável por iniciar o software que estará aguardando as conexões dos programas instrumentados localizados nos dispositivos móveis ou emuladores para envio das informações de execução dos mesmos.

Para iniciar o servidor de teste é necessário executar o seguinte comando oferecido pela JaBUTi/ME:

```
java -cp <classpath \jabutime> br.jabuti.device.ProberServer
         <port> <filename>
```

O argumento `<port>` indica qual é a porta que o servidor de testes ficará preparada e responsável pelo recebimento das informações enviadas pelos dispositivos móveis ou emuladores. O segundo é o nome e caminho do arquivo de configuração que diz quais são as aplicações que poderão enviar informações ao servidor de testes. Um exemplo desse comando segue abaixo:

```
java -cp ../../jabuti/Jabuti-bin.zip br.jabuti.device.ProberServer 2000
         D:/j2me-tools/workspace/bmi/ServerProjects.txt
```

Neste comando a porta `2000` será utilizada para receber as informações, é possível observar que no momento da instrumentação dos arquivos do exemplo, a porta do servidor utilizada também foi a `2000`. Um exemplo do conteúdo do arquivo com as aplicações pode ser visualizado a seguir:

```
bmi
D:/j2me-tools/workspace/bmi/bmi.trc
```

O primeiro item é o nome ou identificador do programa no servidor de teste e a segunda informação é a localização do arquivo *trace* que será utilizado pela JaBUTi/ME para análise de cobertura dos dados recebidos pelo servidor.

Após esses passos já é possível implantar o software no dispositivo móvel e o mesmo irá enviar as informações de execução para o servidor de teste indicado no passo de instrumentação do programa.

As opções de armazenamento e envio de informações ao servidor são: 1) Sem estabelecer o *threshold* de memória, o cliente abre uma conexão com o servidor, armazena e envia-os ao final da execução do programa, posteriormente fecha a conexão; 2) Com a opção para não manter a conexão aberta, no inicio da execução do aplicativo uma conexão é aberta com o servidor para efeito de teste de comunicação, e somente ao final da execução é criada a conexão para envio das informações e fechada a conexão com o servidor; 3) Também é possível indicar a quantidade de memória disponível para armazenamento dos dados de *trace* e quando o tamanho for alcançado, uma conexão com o servidor é criada e enviados os dados para análise; 4) Pode-se também utilizar um arquivo temporário no próprio dispositivo em que os dados sempre serão enviados para esse arquivo *trace* e ao término da execução são enviados para o servidor; ou 5) Apresentar os dados na saída padrão do dispositivo sem envio ao servidor.

A decisão de qual estratégia seguir e qual o endereço do servidor de testes, para que as informações coletadas sejam enviadas, deve ser tomada no momento da instrumentação. Um ponto de destaque é que para alterar o endereço do servidor ou a estratégia deve-se instrumentar as classes do programa novamente.

Dessa forma a JaBUTi/ME mostra ser uma ferramenta que disponibiliza um conjunto de opções para apoiar o teste estrutural de software Java ME. Nesse trabalho a JaBUTi/ME será utilizada como objeto de estudo na análise de três técnicas de teste de software e seus recursos também serão avaliados pelos autores e participantes desse trabalho.

Os exemplos ilustrados nessa seção foram retirados da preparação real de um dos programas, utilizado para treinamento, que ajuda compor o pacote de experimentação proposto por esse trabalho.

3.6 Considerações Finais

Neste capítulo foram apresentados algumas técnicas de teste que compõem o estudo proposto. Dentre elas estão a técnica *ad hoc*, funcional e estrutural. Critérios de teste que pertencem a cada técnica foram apresentados, dos quais podem se destacar a análise de valor limite e particionamento de equivalência da técnica funcional, e *Todos-Nós* e *Todas-Arestas* da técnica estrutural. Algumas ferramentas que tratam do teste de software foram mostradas e uma delas, a JaBUTi/ME, foi apresentada como uma ferramenta de apoio ao teste estrutural. Dentre as principais características da JaBUTi/ME se destacam a possibilidade de testar o *software* em dispositivo real ou emulado e sua independência do código fonte do programa, além de permitir diversas estratégias para envio das informações de execução do software para análise dos dados de teste. Por fim, foi apresentado exemplos de preparação de um *software* para realização de testes proposto nesse trabalho. No próximo capítulo será abordado o assunto das características da engenharia de software experimental e apresentado como definir um experimento nessa área.

CAPÍTULO 4

Engenharia de Software Experimental

A ciência gira em torno de experimentos, e esses são realizados desde há milhares de anos. Devido sua racionalidade, o homem, a partir de uma ideia busca comprová-la fazendo experiências. Pode-se citar como exemplo um dos primeiros fatos experimentais descrito na história que foi realizada por Galileu Galilei [50]. O filósofo Aristóteles (384 a.C. a 322 a.C.) é considerado um dos maiores pensadores da história, por ser tão respeitado ele dizia que corpos com diferentes massas chegariam ao chão em tempos diferentes se abandonados em queda vertical. Essa teoria foi sustentada até 1642 quando Galileu Galilei, que defendia a ideia de “ver para crer”, e realizou um experimento simples em público para comprovar sua ideia diferente e seus estudos, ele subiu na famosa Torre de Pisa na Itália, e soltou de lá duas pedras de tamanhos diferentes ao mesmo tempo (uma com o dobro do peso da outra), e comprovou a todos que independente de sua massa os objetos chegam ao mesmo tempo ao chão. Essa experiência causou grande impacto nos estudiosos e nas autoridades da época. Galileu Galilei com este experimento pode também aperfeiçoar seus estudos sobre a famosa Lei do Movimento Uniformemente Acelerado. E posteriormente Isaac Newton (1643 - 1727) e depois Albert Einstein (1879 - 1955) utilizaram os resultados de seus experimentos para criar a Teoria Mecânica do Cosmo [30].

Os experimentos são de fundamental importância para a ciência, e por meio deles é possível comprovar fatos que são observados e até criar teorias. É importante notar que os experimentos, em alguns casos, não possuem contribuição científica tão relevante, mas por meio de experiências registradas por experimentação é possível criar bases de informações que podem ser úteis a outros pesquisadores, como aconteceu com Newton e Einstein, já que futuramente com mais estudos, podem modificar totalmente a forma como os fatos são vistos e analisados.

No Capítulo 2 foram discutidas algumas características de dois conhecidos modelos relacionados à engenharia de software, o CMMI e o MPS.BR que atuam na melhoria em processo de software. Ambos os processos são utilizados por pessoas para a construção de muitas atividades. Uma forma de avaliar os processos de melhoria e as atividades desenvolvidas por pessoas são os estudos experimentais. A experimentação

fornece uma sistemática, disciplinada, mensurável e controlada forma de avaliar uma atividade realizada por pessoas.

4.1 Experimentos na Engenharia de Software

Novos métodos, técnicas, linguagens e ferramentas são sugeridas e lançadas ao mercado a todo momento. É muito importante a avaliação dessas novas “invenções”, seus propósitos e comparação com as existentes. Os experimentos na engenharia de software devem ser utilizados com esse propósito, ou seja, deve-se utilizar os métodos e estratégias disponíveis para pesquisar e usufruir das qualidades de toda nova tecnologia oferecida, ou refutar outras. De acordo com Basili [7] quatro são os métodos de pesquisa na engenharia de software: 1) Método Científico - O mundo é observado e um modelo é construído baseado na observação; 2) Método Construtivo - Soluções atuais são estudadas e propostas mudanças a elas, e então avaliadas; 3) Método Empírico - Um modelo é proposto e avaliado por meio de estudos empíricos (estudos de casos ou experimentos); e 4) Método Analítico - Uma teoria formal é proposta e comparada com observações empíricas.

Para investigações na engenharia de software os métodos analítico, construtivo e científico são inapropriados. O mais adequado é o método empírico, pois além de ser usado nas mais diferentes áreas como ciências sociais, psicologia, leis da natureza e física, é voltado para o comportamento humano, ou em atividades desempenhadas por pessoas. Ou seja, é o único que consegue avaliar as pesquisas que se baseiam na criatividade humana, como é o caso do desenvolvimento de software. Há duas abordagens para os estudos empíricos: 1) Pesquisa qualitativa - que concentra os estudos em objetos em seu ambiente natural; e 2) Pesquisa quantitativa - que concentra em quantificar relações ou comparar dois ou mais grupos. É possível que uma mesma pesquisa avalie qualitativamente e quantitativamente seus objetivos, embora cada tipo responda a determinado tipo de questão. Por exemplo, enquanto a pesquisa quantitativa investiga o quanto determinado método aumenta a quantidade de defeitos encontrados em um produto, a visão qualitativa foca nas diferenças de tipos de inspeção de defeitos.

De acordo com o propósito a ser avaliado no experimento, Robson [54] define três tipos de estratégias de investigação: 1) Inspeção - é feita quando o uso de determinada técnica ou ferramenta já foi aplicada ou introduzida, pode ser vista como uma amostra de determinada situação; 2) Estudo de Caso - investiga uma simples entidade ou fenômeno por um tempo específico, a partir da qual os pesquisadores coletam informações detalhadas do produto analisado; e 3) Experimento - são executados em ambiente controlado e é necessário manipular diretamente, precisamente e sistematicamente o comportamento

do produto, muito utilizados para comparar resultados. Runeson e Höst [55] propõem um guia para condução de pesquisas por meio de estudo de caso.

Os três tipos de estratégias podem ser comparados na Tabela 4.1. Detalhando as informações da Tabela 4.1, o Controle de Execução descreve o quão o pesquisador pode controlar a pesquisa, por exemplo, o experimento é o único que possui suas características de controle. O Controle de Medição especifica o grau de medição que o pesquisador possui ao adotar sua estratégia. De acordo com o tipo de estratégia adotado o custo difere, como o experimento necessita de laboratório e de controle seu custo é mais elevado do que uma investigação populacional. Um importante aspecto a se considerar é a facilidade de replicação, cujo objetivo é mostrar que o resultado do experimento é valido para uma amostra cada vez maior.

Tabela 4.1: *Comparação de Estratégias de Pesquisa.*

Fator	Inspeção	Estudo de Caso	Experimento
Controle de Execução	Não	Não	Sim
Controle de Medição	Não	Sim	Sim
Custo de Pesquisa	Baixo	Médio	Alto
Facilidade de Replicação	Alto	Baixo	Alto

A replicação de experimentos envolve repetir a pesquisa sob condições idênticas, mas com outra população. Se os dados capturados nas replicações são os mesmos ou parecidos para o universo das amostras, isso mostra estatisticamente que, provavelmente, também deve acontecer o mesmo para toda a população. Segundo Wohlin [66] experimentos são adequados para investigar, principalmente, os seguintes aspectos:

- Confirmação de teorias (teste de teorias);
- Confirmação de sabedoria convencional (teste de conceitos de pessoas);
- Explorar relações (teste de comportamento esperado);
- Avaliar a eficácia de modelos (teste de modelos, técnicas e outros); e
- Avaliação de medidas (teste de qual medida é suposta para algo).

A realização de experimentos não é a tarefa mais difícil das estratégias de pesquisa, dentre suas exigências pode-se destacar a necessidade de preparar, conduzir e analisar o experimento. Porém algumas vantagens são oferecidas como o controle de todo ambiente de execução do experimento (participantes, objetos de estudo, instrumentação e outros). Outras vantagens incluem o apoio da análise estatística utilizando o teste de hipóteses e grande facilidade de replicação. Como foi abordada anteriormente (no início desse capítulo - que descreve as ideias dos seres humanos), diversas ideias surgem a partir de pensamentos humanos. Esse é o ponto inicial de um experimento, ter uma ideia de uma causa ou efeito de relacionamento. Posteriormente são formuladas hipóteses baseadas na

ideia e no fato a ser pesquisado, ou seja, quais são as relações entre eles. O principal objetivo de um experimento é mostrar a avaliação ou relacionamento de suas hipóteses, buscando conclusões válidas.

Durante um experimento são avaliados dois tipos de variáveis, as chamadas variáveis independentes, ou seja, as variáveis que são manipuladas e controladas durante o experimento. E as variáveis dependentes ou variáveis de resposta, que são as variáveis que sofrem mudanças pelas variáveis independentes. As variáveis independentes podem ser, por exemplo, métodos, ferramentas, pessoas, ambiente, dentre outros. As variáveis dependentes por sua vez podem ser, por exemplo, uma medida de produtividade, tempo, quantidade, dentre outros.

O processo experimental, segundo Wohlin [66] é conhecido por iniciar com uma ideia de experimento e caracteriza-se por iniciar a partir da Definição do experimento, passando pelo Planejamento, Operação, Análise e Interpretação até a Apresentação e Empacotamento do experimento, conforme ilustrado na Figura 4.1. Algumas partes do processo são compostas por passos, sendo que o Planejamento do Experimento é composto por sete passos, a Operação do Experimento e a Análise e Interpretação possuem cada uma três passos. Para o sucesso do experimento é necessário que cada passo seja concluído para iniciar o seu subsequente. A primeira atividade do processo de experimentação é a Definição dos termos do experimento, seus objetivos e metas. As hipóteses são iniciadas, mas não são definidas nesse passo. O objetivo é formulado para resolver o problema, e as seguintes perguntas devem ter respostas ao final dessa atividade, “o que será estudado?”, “qual é o propósito?” “qual o foco de qualidade?” “qual é a perspectiva?” “como o estudo será conduzido?”. Segundo Wohlin [66] a definição pode ser resumida obedecendo à seguinte estrutura:

- Analisar <objeto de pesquisa>
- Com o propósito de <propósito>
- Com respeito à <enfoque>
- Do ponto de vista da <perspectiva>
- No contexto de <contexto>

O Planejamento é a segunda atividade e trabalha no projeto do experimento, descreve quais medidas serão coletadas e analisadas na próxima etapa (análise e interpretação). Também são definidos os detalhes do experimento, por exemplo, pessoas e ambiente. Se o local do experimento será executado na indústria ou na academia. No segundo passo dessa atividade, as hipóteses são feitas formalmente e são compostas por hipóteses nulas e hipóteses alternativas. Logo depois, no terceiro passo são planejadas as atividades para determinar as variáveis independentes e dependentes, analisando escalas de medida, método que será aplicado e por quem, para que se possa realizar a análise estatística futuramente. No próximo passo, são definidos qual o tipo de seleção dos participantes, se

será aleatório ou definido, ou seja, como representar uma população por meio da amostra e qual o número de indivíduos recomendado para ter uma maior confiança nos resultados.

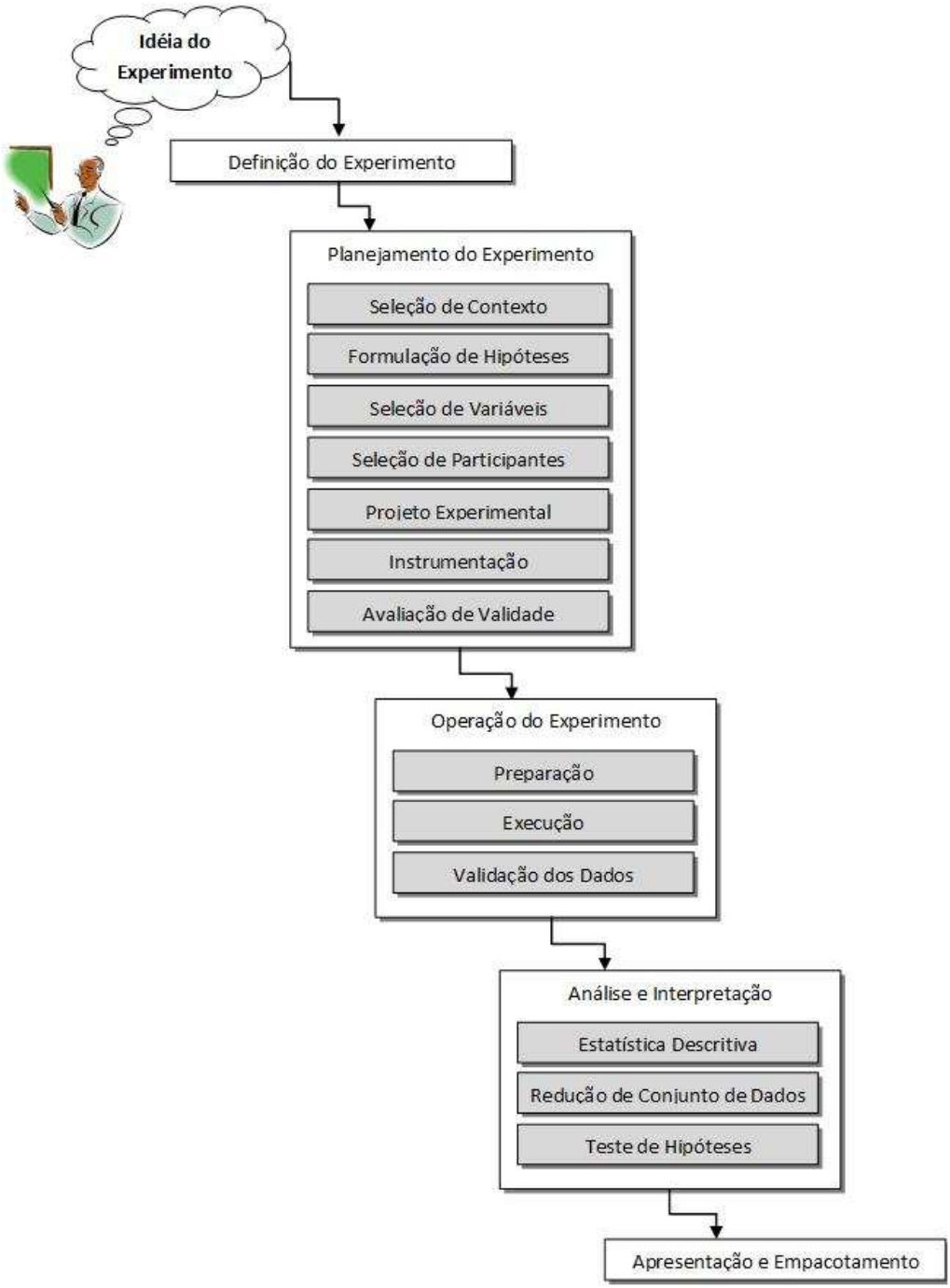

Figura 4.1: *Processo Experimental Segundo Wholin [66].*

No projeto do experimento são considerados fatores como as formas de distribuição das pessoas como, por exemplo, blocagem, balanceamento, aleatoriedade e os objetos durante a realização do experimento. No passo subsequente, a instrumentação é realizada e inclui identificar as diretrizes necessárias aos participantes para se guiarem durante a re-

alização do experimento, em alguns casos são criados *checklists*. O último passo é avaliar se os resultados esperados são confiáveis e se o controle do experimento foi efetivo.

Na operação antes do experimento ser executado ele passa por uma etapa de preparação, em que são passadas informações aos participantes e preparados os materiais, formulários e ferramentas para a execução. Posteriormente o experimento é executado ou conduzido, e os participantes produzem os dados que serão coletados. O último passo dessa etapa é a validação da coleta dos dados criados no passo da execução.

Depois de coletados e validados os dados é iniciada a etapa de análise e interpretação, o primeiro passo é procurar entender os dados de acordo com a estatística descritiva, produzindo a visualização dos dados e descrevendo as estatísticas que ajudarão no entendimento e interpretação das informações coletadas. O próximo passo é, se necessário e de acordo com a análise estatística, remover da amostra os dados que não são úteis ou não foram capturados corretamente e poderão influenciar nos resultados. Depois de removidos já é possível realizar o último passo, chamado de teste das hipóteses, que é um teste estatístico que dá confiabilidade aos resultados obtidos, podendo refutar a hipótese nula ou aceitá-la.

A última etapa do processo de experimentação é a apresentação e empacotamento, na qual os resultados obtidos são documentados e preparados para publicação. Eles podem ser publicados em artigos de revistas, pacotes de laboratório ou base de conhecimento.

Após o término de todas as etapas do processo de experimentação, é possível criar replicações do experimento. As replicações segundo Fusaro [31] significa uma reprodução, da forma mais fiel possível do experimento original, por outros pesquisadores em ambientes e condições diferentes. Assim, quando duas replicações apresentam resultados semelhantes elas fortalecem a hipótese estudada, aumentando a credibilidade do estudo experimental. Caso contrário é criado um aprendizado sobre os experimentos e podem ser criadas novas hipóteses que focam em explicar as diferenças encontradas nas replicações.

Basili [10] relaciona três tipos diferentes de replicações: 1) quando não há alteração nas hipóteses, as replicações reproduzem de maneira mais fiel possível o experimento a fim de aumentar a credibilidade do experimento, ou quando ocorrem mudanças na forma como o experimento é conduzido visando melhorar aspectos internos do experimento; 2) quando há alteração nas hipóteses são destacados três tipos de mudanças: a) ao alterar as variáveis dependentes (forma como está sendo medida uma variável) pode melhorar nos resultados; b) ao alterar as variáveis independentes varia-se o processo e examinam os resultados; e c) altera as variáveis de contexto buscando identificar os fatores ambientais que afetam o resultado; e 3) quanto se alteram diversos

fatores (processo, produto e contexto) e busca identificar se apresenta o mesmo resultado da teoria em condições diferentes.

Diversos trabalhos encontram-se na literatura dentre eles se destaca o experimento de Hetzel [34] que fez a comparação entre três técnicas (leitura de código, teste de especificação e teste misto). Myers [48] também realizou experimentos para estudar melhor a aplicação das técnicas de teste funcional, estrutural e o método de inspeção. Kamsties e Lott [27] e Basili [9] também realizaram experimentos com a técnica funcional, estrutural e leitura de código. Linkman et al. [56] estudou o teste funcional, o teste de mutantes e o teste aleatório. Que se tenha conhecimento até os dias atuais nenhum trabalho, que não seja do autor deste, foi publicado contemplando exclusivamente as três técnicas de teste *ad hoc*, funcional e estrutural (foco deste trabalho).

Este trabalho é voltado para a criação de um pacote de experimentação que visa avaliar essas três técnicas de teste. Além de avaliar o que a ferramenta JaBUTi/ME consegue proporcionar para garantir mais qualidade em software para dispositivos móveis. E mais três replicações do mesmo pacote foram realizadas a fim de fortalecer as hipóteses propostas no experimento.

4.2 Considerações Finais

Neste capítulo foram apresentados os conceitos mais importantes da engenharia de software experimental e quais são as possíveis estratégias de pesquisa para realização de estudos científicos. Dentre eles foram destacados a investigação, o estudo de caso e o experimento. Voltados para a execução de experimentos, pois é o foco do trabalho atual, foram citadas as partes que compõem um experimento, partindo de seus objetivos até a análise estatística e validação de hipóteses. Também foi apresentado um processo para a realização de experimentos na engenharia de software. No Capítulo 5 é proposto um pacote de experimentação para a avaliação de técnicas de software voltados para os dispositivos móveis.

CAPÍTULO 5

Pacote de Experimentação Proposto para a JaBUTi/ME

A tecnologia evolui muito rápido, a cada dia está presente em mais e mais produtos. O software está fazendo parte de quase todos os produtos lançados ao mercado. Apesar de a tecnologia contribuir para muitos problemas, ela ainda por enquanto, não consegue resolver outros.

Dentre os produtos lançados ao mercado incluem-se os dispositivos móveis, popularmente conhecidos como celulares, *PDAs*, *Handhelds*, *Palmtops* e outros. Neles se concentram uma grande quantidade de software diversos, desde sistema operacional específico para cada dispositivo até jogos e pequenos aplicativos. Esses software como qualquer outro podem estar propensos a possuírem defeitos.

Existem diversas técnicas de teste de software, dentre elas pode-se destacar *ad hoc*, funcional e estrutural, descritas detalhadamente no Capítulo 3. Algumas questões são bastante comuns, como por exemplo, qual seria a técnica de teste mais aconselhável ou eficaz para encontrar defeitos nos software de dispositivos utilizados cotidianamente? Existe alguma ferramenta que ajude na descoberta de defeitos? Como garantir que o software móvel contém somente o que lhe foi proposto? Essas e algumas outras questões fazem parte da proposta deste pacote de experimentação que compõe este trabalho e será descrito daqui a diante.

Das estratégias empíricas apresentadas na Seção 4.1, a que mais se adéqua as características do estudo é o experimento. As justificativas para tal escolha foram às seguintes características não abordadas pelas outras estratégias empíricas (inspeção e estudo de caso): 1) execução em ambiente controlado e rigoroso; 2) necessidade de análise estatística para validar os estudos; 3) utilização de unidades de medida para avaliação das técnicas; 4) comparação de dois ou mais métodos mostrando suas significâncias estatísticas; e 5) Facilidade de replicação. O pacote de experimentação foi proposto com a intenção de possibilitar sua replicação, ou seja, sua execução por diversas vezes a fim de fortalecer ou refutar as hipóteses e conclusões apresentadas nesse trabalho. Dois pontos de vista são fundamentais para a aplicação desse experimento, o primeiro é o ponto de

vista do replicador cujo foco foi deixar claras as regras para que qualquer pessoa (com conhecimento nas técnicas aplicadas) pudesse executar uma replicação do experimento. O segundo é o ponto de vista do participante, cujo foco foi criar uma metodologia que fosse objetiva e que o participante, além de contribuir para a pesquisa, pudesse aprender e aplicar esse conhecimento em sua vida profissional ou acadêmica.

O objetivo final dos estudos é contribuir para o desenvolvimento de uma estratégia de teste incremental (com o apoio de uma ferramenta de teste) que possa ser empregada para garantir a qualidade de produtos de software e sistemas de informação que utilizem dispositivos móveis. Considera-se que com a crescente demanda por produtos de software para tais dispositivos, os resultados obtidos com esses estudos podem contribuir de maneira significativa na avaliação das técnicas de testes e melhoria da qualidade dos produtos finais Java ME utilizados por toda população mundial. De acordo com a estrutura proposta por Wohlin [66] de estudos experimentais, visto na Seção 4.1, seguem abaixo as etapas do processo experimental e logo em sequência os resultados do experimento realizado.

5.1 Organização do Pacote Proposto

O pacote de experimentação proposto para a análise das técnicas de teste de software Java ME para dispositivos móveis está organizado como definido por Wholin [66] e suas etapas são descritas abaixo:

- **Definição do Experimento**

Analisar as técnicas de teste *ad hoc*, funcional e estrutural, avaliando também a ferramenta utilizada a JaBUTi/ME.

Com o propósito de avaliar e comparar os critérios de teste e criar uma estratégia de método de teste incremental para software Java ME de dispositivos móveis.

Com respeito aos fatores que contribuem para a qualidade do software como, a cobertura de comandos, a quantidade de casos de testes e o número de defeitos encontrados. Do ponto de vista, principalmente dos testadores e desenvolvedores de software, mas se aplica também a gerentes e profissionais diversos da área de TI.

No contexto de alunos de graduação e pós-graduação, e profissionais relacionados a construção de software.

- **Seleção de Contexto**

O contexto escolhido para o experimento foi no problema real em garantir maior qualidade nos software Java ME executados por dispositivos móveis, é específico na área

de teste de software e serão utilizados desenvolvedores de software em geral da academia ou da indústria em laboratório da UFG (Universidade Federal de Goiás) com tempo e ferramentas controladas. A ferramenta utilizada para realizar as medidas de comparação entre as técnicas será a JaBUTi/ME.

- **Formulação de Hipóteses**

  As hipóteses do experimento são:
  Hipóteses nulas ou H_0:

H_{01} - A técnica estrutural com a ferramenta JaBUTi/ME revelou a mesma quantidade de defeitos que a técnica *ad hoc* ou que a técnica funcional;
H_{02} - A técnica estrutural com a ferramenta JaBUTi/ME revelou percentual equivalente de cobertura da técnica *ad hoc* ou funcional;
H_{03} - A técnica estrutural com a ferramenta JaBUTi/ME não contribuiu na criação de novos casos de teste.

  Hipóteses alternativas ou H_1 ou H_A:

H_{11} - A técnica estrutural com a ferramenta JaBUTi/ME revelou quantidade de defeitos superior que a técnica *ad hoc* e/ou que a técnica funcional;
H_{12} - A técnica estrutural com a ferramenta JaBUTi/ME revelou maior percentual de cobertura da técnica estrutural em relação à *ad hoc* e/ou funcional;
H_{13} - A técnica estrutural com a ferramenta JaBUTi/ME contribuiu na criação de novos casos de teste, não identificados antes pelas técnicas *ad hoc* e/ou funcional.

- **Seleção de Variáveis**

As variáveis do experimento são divididas em dois tipos, independentes e dependentes e são:

Variáveis independentes

1. Técnica *ad hoc*;
2. Técnica funcional;
3. Técnica estrutural;
4. Complexidade dos programas;
5. Programas selecionados.

Variáveis dependentes

1. Quantidade de defeitos revelados;
2. Percentual de cobertura;
3. Quantidade de casos de testes criados.

- **Seleção dos Participantes**

Os participantes foram selecionados por ordem de inscrição no curso até preencherem todas as vagas oferecidas no laboratório de realização dos experimentos. Esses participantes representam uma amostra dos diversos profissionais relacionados a construção de software que buscam garantir mais qualidade em seus produtos. Um pré-requisito foi necessário para que não houvesse maiores dificuldades para a aplicação das técnicas no tempo determinado, o pré-requisito para participar do experimento era que o indivíduo deveria ter conhecimentos básicos da linguagem de programação *Java*. Nenhum conhecimento em teste de software foi requisitado. Os participantes do experimento deverão ser divididos em grupos para aplicação das técnicas em cada um dos programas. A técnica *fatorial-fracional randomizado* [29] será utilizada para distribuição dos programas nos grupos, ou seja, cada participante de um grupo irá aplicar uma técnica em cada programa, sendo que ao final do experimento todos os participantes aplicaram todas as técnicas em todos os programas.

Como os participantes aplicavam as técnicas em um mesmo momento, porém em programas distintos, foi solicitado aos mesmos que não revelassem defeitos ou dicas para encontrá-los em programas que ainda não haviam sido testados por determinado individuo. Essa validação interna deve ser verificada pelos pesquisadores do experimento, e caso observe algum fator que indique a transferência dessas informações deve ser anotados os envolvidos e retirado-os da amostra.

- **Projeto Experimental**

Um curso de teste de software proposto para a comunidade acadêmica e industrial foi disponibilizado para que o experimento pudesse ser realizado. Esse curso foi organizado em três etapas, 1) Coletar dados do conhecimento atual em teste de software dos participantes (técnica *ad hoc*); 2) Treinamento da técnica funcional e aplicação da técnica coletando os dados da aplicação dessa técnica; e 3) Treinamento da técnica estrutural com a utilização da ferramenta JaBUTi/ME e também coletados os dados da aplicação da técnica. O objetivo do experimento proposto é comparar eficácia das três técnicas no ambiente de desenvolvedores de software.

Informações sobre o perfil do participante e uma avaliação do curso/experimento serão obtidas por meio de formulários de resposta, dos quais os participantes irão preenchê-los, um no início do experimento e outro ao final do mesmo. As informações sobre defeitos e casos de teste serão obtidos também por meio de formulários. O participante deve preencher o formulário de caso de teste a cada caso de teste criado e executado. As informações sobre cobertura serão analisadas pela ferramenta JaBUTi/ME, porém os participantes só tomarão conhecimento da mesma quando receberem o treinamento sobre a

técnica estrutural. Portanto a JaBUTi/ME será utilizada em dois momentos, um para avaliar a cobertura dos programas independente da técnica aplicada e no treinamento prático sobre a técnica estrutural.

Como a JaBUTi/ME necessita de um servidor de testes para receber as informações de cobertura dos programas, cada máquina do laboratório possui instalado um servidor e os arquivos *trace* serão enviados para esse servidor, posteriormente essas informações serão compactadas e enviadas para um servidor único no qual o instrutor validará o recebimento do mesmo concluindo a aplicação de uma técnica.

- **Instrumentação**

Para a preparação do pacote de experimentação muito trabalho foi realizado como: preparação de formulários, captura dos programas a serem utilizados no experimento, inserção de enganos nos programas (que culminam em defeitos), preparação dos programas para captura de suas coberturas, preparação do ambiente para execução dos programas, guias para os participantes, instruções e especificações dos programas, dentre outros. Os formulários desenvolvidos, bem como todos os resultados coletados estão disponíveis em `http://www.inf.ufg.br/~auri/curso`.

Primeiramente foram preparadas as especificações dos programas, ou seja, uma especificação dos requisitos de cada software foi elaborada para ser entregue aos participantes para o entendimento dos programas. As especificações dos programas podem ser visualizadas no Apêndice A.1.

Os formulários para captação do perfil do participante foram criados com o intuito de verificar analogias do perfil e experiência do participante com os resultados obtidos no experimento, por meio deste formulário é possível obter informações como experiência acadêmica e industrial, experiência em teste de software, experiência na linguagem *Java*, quais das três técnicas de teste já eram conhecidas, se o participante já participou de outros treinamentos com as técnicas deste experimento, e outros. Este formulário também pode ser visto no Apêndice A.3.1. O formulário para avaliação do curso e sugestões é aplicado ao final do experimento, e suas principais funções é captar por meio de uma auto-avaliação se o conhecimento adquirido durante o experimento foi válido para a vida acadêmica ou industrial do participante e para que o mesmo possa conceder sugestões de melhoria para futuras replicações do experimento. Estes dois formulários também podem ser visualizados no Apêndice A.3.4 e A.3.5 respectivamente.

Um dos principais formulários do experimento é o que registra as informações dos casos de teste projetados e executados pelos participantes de acordo com a técnica praticada, e pode ser encontrado no Apêndice A.3.3. Nesse formulário informações básicas de qualquer caso de teste como, entradas, saída produzida e saída esperada são registradas. Outras informações também foram solicitadas para serem preenchidas para

realização de análise que são indicativos de erro e hora da execução do caso de teste. Por meio dessas informações o participante indica se o que foi produzido pelo programa é um erro ou não e quanto tempo foi gasto para projetar e executar os casos de teste. Para auxiliar os participantes devido ao número de passos e a sequência de execução necessária para a realização do experimento, um roteiro foi criado com o objetivo de guiar os participantes durante o processo de experimentação, um para cada técnica a ser aplicada. No Apêndice A.2 é possível encontrar este guia.

Para os instrutores também foram preparadas apresentações *(slides)* com o conteúdo a ser ministrado, teoria geral de testes antes da aplicação da técnica *ad hoc*, teoria da técnica funcional e outro sobre a teoria da técnica estrutural. Cada uma dessas técnicas foi exemplificada com o intuito de treinamento aos participantes do experimento com o programa *P0* da Tabela 5.1.

Para a seleção dos programas do experimento, vinte programas que obedecem à arquitetura Java ME foram pré-selecionados dos repositórios `http://www.sourceforge.net/` e `http://code.google.com/`. Destes apenas quatro foram selecionados para participar do experimento, os critérios para seleção dos programas foram, a existência de código fonte e a complexidade dos programas, sendo escolhidos os de maior complexidade.

A existência de código fonte era necessária para que os defeitos fossem inseridos nos programas, já que o foco do pacote de experimentação inclui em encontrar defeitos para comparar as três técnicas. A complexidade ciclomática também foi determinante na escolha dos programas, visto que muitas das aplicações Java ME disponíveis são extremamente simples e sem complexidade, assim seria difícil inserir enganos no software devido a pouca quantidade de código e de sua complexidade. Um programa será utilizado para treinamento dos participantes nas técnicas funcional e estrutural, os demais programas selecionados serão utilizados pelos participantes para praticarem os conceitos das técnicas durante o experimento, e um resumo de suas funcionalidades podem ser analisados na Tabela 5.1 e Tabela 5.2.

Tabela 5.1: *Programa Selecionado para o Treinamento e sua Complexidade.*

Id.	Nome	Complexidade	Descrição
P0	B.M.I.	2,32	Calcula o Índice de Massa Corporal de um indivíduo e o classifica de acordo com as faixas definidas.

Com base em critérios baseados em mutação uma média de dez erros foram inseridos nos programas escolhidos que fazem parte do experimento. Os defeitos inseridos foram de inicialização de variáveis, cálculos, fluxo de controle, interface e estrutura de dados, baseados na Tabela 5.3. Após a inserção dos defeitos, os programas foram compilados e instrumentados utilizando os recursos oferecidos pela JaBUTi/ME, viabilizando o

rastreamento da execução dos casos de teste e, posteriormente, a análise de cobertura dos mesmos em relação aos critérios apoiados pela ferramenta. Os defeitos inseridos em cada programa podem ser conferidos na Seção A.4 no Apêndice A.

Tabela 5.2: *Programas Selecionados para Prática dos Participantes e suas Complexidades.*

Id.	Nome	Complexidade	Descrição
P1	AntiPanela	3,87	Faz o cadastro de jogadores e realiza o sorteio das equipes com base no número de jogadores evitando a formação de "panelas".
P2	CarManager	5,52	Controla e gerencia os gastos de combustível de um veículo automotor.
P3	CódiceFiscale	6,17	Verifica a validade ou gera o "Codice Fiscale" italiano, semelhante ao CPF brasileiro.

O experimento será realizado em três períodos programados sendo que para cada técnica de teste será ministrada uma hora de treinamento e, posteriormente, os participantes terão uma hora e meia para a aplicação prática da mesma em cada um dos programas selecionados.

O laboratório utilizado possui 30 máquinas *desktop* com Sistema Operacional Linux, *Java* 6.0, Eclipse, Wireless Took Kit 2.5, EclipseME e a ferramenta JaBUTi/ME instalados.

Para viabilizar a coleta de dados de execução de todos os programas, estes foram instrumentados previamente utilizando-se a ferramenta JaBUTi/ME. Desse modo, mesmo quando são utilizados critérios *ad hoc* ou funcional para a geração dos conjuntos de teste, a execução dos testes é "monitorada" e a cobertura dos mesmos pode ser posteriormente avaliada em relação aos critérios estruturais implementados pela JaBUTi/ME. Vale ressaltar que a mesma ferramenta foi utilizada para avaliação das três técnicas utilizadas. Na Figura 5.1 é ilustrado esse processo de execução das aplicações instrumentadas e como a coleta das informações de cobertura é realizada. Além disso, a quantidade de defeitos identificados pelos casos de teste criados a partir de cada critério também será coletada por meio de formulários que preenchidos pelos participantes.

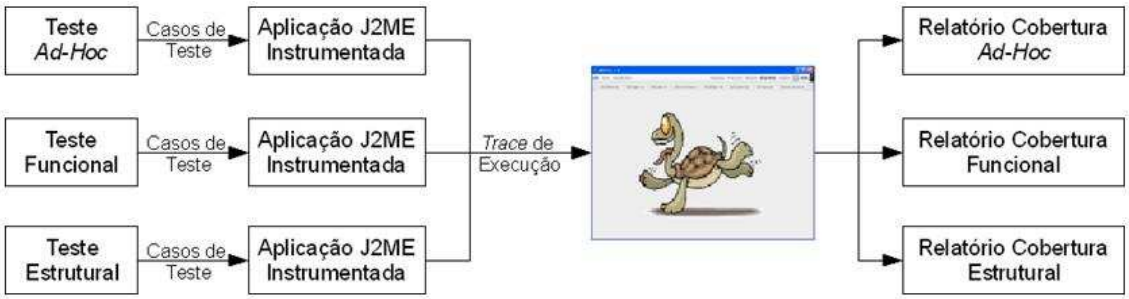

Figura 5.1: *Esquema de monitoramento adotado.*

A mesma classificação de Basili & Selby de 1987 [9] foi utilizada para classificar os defeitos inseridos nos objetos do experimento.

Tabela 5.3: *Taxonomia de Defeitos.*

Classe de Defeito	**Causa do Defeito/Exemplo**
Omission	Causados por falta de segmento de código fonte;
Commission	Causados por um segmento de código fonte incorreto;
Tipo de Defeito	**Causa do Defeito/Exemplo**
Initialization	Causados pela inicialização incorreta de variáveis ou estrutura de dados;
Computation	Causados pela falta ou presença de uma expressão incorreta; Por exemplo, operador de incremento na ordem errada;
Control	Causados pela execução de um caminho errado; Por exemplo, uso incorreto de IF-THEN-ELSE;
Interface	Causados pela suposição de um módulo ao invocar outro módulo; Por exemplo, supõe que a passagem do objeto não altera seu estado, porém altera;
Data	Causados pelo uso incorreto de uma estrutura de dados; Por exemplo, índice inválido de um vetor;
Cosmetic	Não causam um resultado inválido do programa; Por exemplo, um erro de ortografia;

- **Avaliação da Qualidade**

Para a validade interna do experimento a quantidade de participantes e a grande quantidade de testes executados garantem bons dados para esse quesito. Concentrando na validade externa, como o perfil dos participantes é baseado em profissionais acadêmicos ou industriais que possuem conhecimento na linguagem de programação *Java*, o experimento tem grandes chances de apresentar os mesmos resultados quando o experimento for aplicado a engenheiros de software em geral com conhecimentos em *Java*. A maior preocupação é com a validação da conclusão, nos quais os participantes deverão seguir exatamente os passos fornecidos nos guias e entregarão uma grande quantidade de informações, algumas em papel por meio dos formulários e outros por meio magnético em formato de arquivos. É possível haver falhas no preenchimento, na execução e no envio das informações para validação, assim dados inconsistentes poderão existir e deverão ser descartados do conjunto amostral. Sobre a validade de construção as unidades de medidas adotadas para avaliação das técnicas são mais que suficientes para avaliação e validação das três técnicas do experimento.

- **Preparação**

Os participantes não ficam cientes dos aspectos pelo qual é intencionado o estudo. Para eles é apenas um curso de teste de software que está sendo ministrado.

Porém são informados dos passos que devem seguir durante o estudo em cada aplicação de técnica. Apenas ao final do experimento é informado aos participantes que os programas que eles executaram estavam coletando informações sobre os testes realizados e que as três técnicas que eles aplicaram serão comparadas. Os nomes dos participantes não são relevantes para a análise dos dados.

O conjunto de formulários, os programas, suas especificações, juntamente com o guia para execução dos testes devem ser previamente preparados, organizados e distribuídos aos participantes do experimento. O ambiente do laboratório também deve ser uniforme e com os software necessários para a execução do experimento.

É necessário passar para os participantes a importância de seguir as instruções fornecidas e mostrando a importância de relatar a realidade dos fatos que acontecem durante a execução do experimento. Deve ser solicitado aos participantes para não conversarem entre si e não trocarem informações sobre defeitos encontrados nos programas.

- **Execução**

O pacote de experimentação descrito foi replicado por três vezes. Na primeira replicação foi executado em dois dias com intervalo semanal com trinta participantes. A segunda e a terceira replicações foram executadas em três dias consecutivos cada uma. Todas replicações tiveram um total de doze horas de duração, sendo apenas a primeira dividida em duas de seis horas, e a segunda e terceira com três horas cada dia. Todos os envolvidos na construção de software e com conhecimentos na linguagem *Java*. O laboratório para a realização do experimento fica no Instituto de Informática da Universidade Federal de Goiás. Qualquer desvio de execução do experimento deve ser documentado e todos os procedimentos devem obedecer ao tempo estimado.

- **Validação dos Dados**

Para validar as informações coletadas no experimento, um computador foi designado para receber os formulários preenchidos e os dados de cobertura. Ao final da execução dos testes de cada técnica os participantes enviaram as informações para validação dos organizadores do experimento. Caso fosse detectada alguma divergência de envio os participantes eram notificados a repetir o procedimento.

Embora a validação fosse executada não havia tempo hábil para validar o conteúdo enviado, ou seja, a validação realizada nesta etapa era de recebimento dos arquivos, pois uma verificação mais detalhada necessitava de mais tempo para ser executada. Os dados perdidos são definidos no item *Redução* mais adiante.

- **Estatística Descritiva**

Para análise estatística deste experimento foram utilizadas as escalas do tipo ordinal e intervalar. Nesses dois tipos de análise a mediana e a média, respectivamente, foram utilizadas como medidas de tendência central para comparações entre as técnicas de teste de software analisadas. Os dados capturados para análise podem ser encontrados na Tabela 5.4. Essa tabela agrupa os dados resultados das três replicações do pacote de experimentação.

Tabela 5.4: *Dados Coletados por Técnica e Critério.*

N.	*ad hoc*			Funcional			Estrutural		
Participante	T-Nos	Casos Teste	Erros Reais	T-Nos	Casos Teste	Erros Reais	T-Nos	Casos Teste	Erros Reais
1	66	10	4	91	4	2	92	12	3
2	86	9	6	75	3	2	83	19	4
3	61	4	1	91	14	5	96	8	0
4	89	20	4	71	19	3	95	20	5
5	78	12	8	63	9	4	73	13	3
6	83	35	7	79	8	4	94	12	7
7	56	11	5	44	11	4	45	6	5
8	92	13	6	72	15	2	88	18	5
9	71	13	0	61	7	3	94	16	1
10	55	0	0	72	6	2	92	10	2
11	69	13	5	61	6	1	98	14	3
12	45	12	5	78	12	3	97	12	2
13	21	1	0	34	7	0	73	5	0
14	81	14	3	22	15	3	62	0	0
15	88	5	1	41	12	3	66	15	1
16	78	4	0	82	18	3	65	11	2
17	50	2	1	43	16	2	48	11	2
18	51	6	3	63	8	2	72	8	2
19	73	8	2	70	13	3	67	11	0
20	51	4	1	48	6	3	70	9	0
21	67	8	3	68	5	3	72	15	3
22	66	4	2	51	9	3	63	0	0
23	62	0	0	67	6	5	54	9	1
24	43	14	0	32	10	0	68	8	0

Tabela 5.4: *Dados Coletados por Técnica e Critério.*

N.	*ad hoc*			Funcional			Estrutural		
Participante	T-Nos	Casos Teste	Erros Reais	T-Nos	Casos Teste	Erros Reais	T-Nos	Casos Teste	Erros Reais
25	68	4	1	45	16	3	69	11	2
26	57	0	0	28	6	0	75	13	0
27	72	16	2	71	1	0	70	12	0
28	62	12	5	74	11	7	73	22	2
29	66	5	0	69	1	0	64	2	1
30	60	12	2	67	3	0	44	12	2
31	18	4	0	70	12	2	71	28	2
32	45	8	2	45	20	0	78	9	1
33	54	9	1	71	10	0	76	11	3
34	52	5	2	42	15	1	18	6	3
35	51	5	2	38	12	0	63	10	4
36	18	11	2	60	9	2	73	13	3
37	54	5	2	60	9	2	76	24	0
38	46	6	1	47	7	2	76	10	4
39	46	18	1	52	4	3	76	10	4
40	73	6	3	65	8	2	19	10	2
41	55	9	2	18	5	2	78	10	5
42	61	27	2	59	7	1	74	0	0
43	44	7	2	57	20	2	76	8	4
44	18	16	4	66	4	1			
45	41	3	0	25	1	0			
46	18	1	0	74	12	4			
47	59	10	3	50	8	1			
48	18	6	0	68	12	1			
49	47	5	2	69	6	2			
50	29	1	1	18	6	1			
51	50	3	2	42	2				
52	18	1	0						

O processo estatístico adotado para a análise estatística ilustrado na Figura 5.2. Inicialmente é necessário agrupar os dados para acordarem com as definições das hipóteses. No próximo passo é feita a avaliação da distribuição da amostra coletada, ou seja,

descobrir se a mesma possui distribuição normal ou não-normal. Caso a distribuição seja normal é necessário utilizar a estatística paramétrica [46], e caso contrário a estatística não-paramétrica [46]. Independente da forma da distribuição é preciso descobrir se há diferença entre as três técnicas do experimento. Caso exista diferença entre as três técnicas é necessário saber qual a diferença entre a comparação das combinações duas a duas das três técnicas avaliadas, caso contrário as técnicas são ditas como equivalentes. No caso de diferença entre as técnicas é possível os valores apresentados pela estatística e concluir quais hipóteses do experimento serão aceitas ou rejeitadas. Vale ressaltar que esses passos são necessários para decidir quais métodos estatísticos devem ser utilizados de acordo com as características dos dados amostrais.

Durante a análise estatística do processo mencionado anteriormente, para emitir cada conclusão sobre cada um dos passos, novas hipóteses (não relacionadas ao experimento, mas relacionadas com a análise estatística) devem surgir e são apresentadas na Tabela 5.5. Todas essas hipóteses fazem parte da análise estatística e são importantes para a conclusão final proposta pelas hipóteses do experimento que busca mostrar quais técnicas possuem melhores resultados nos critérios de cobertura dos nós, quantidade de casos de teste e número de defeitos revelados.

Tabela 5.5: *Hipóteses Estatísticas.*

Hipótese		Teste
H_0	a distribuição da amostra é normal	Teste de Normalidade
H_1	a distribuição da amostra não é normal	Teste de Normalidade
H_2	há diferença entre as três técnicas	Teste de Diferença
H_3	não há diferença entre as três técnicas	Teste de Diferença
H_4	há diferença significativa entre o par comparado	Teste de Comparação
H_5	não há diferença significativa entre o par comparado	Teste de Comparação

O método estatístico é demonstrado na Figura 5.3 e foi definido de acordo com o processo apresentado anteriormente na Figura 5.2. No método é apresentada cada técnica estatística para avaliação de todas as hipóteses do experimento, incluindo as hipóteses de cunho estatístico. Como definido no processo o agrupamento dos dados foi realizado para adaptar os dados as hipóteses do experimento, ou seja, foram agrupados os dados dos critérios (cobertura, quantidade de casos de teste e quantidade de defeitos) por técnica (*ad hoc*, funcional e estrutural), como apresentado na Tabela 5.4.

Por meio do teste de Shapiro-Wilk é possível verificar a normalidade de uma distribuição. Após a aplicação deste teste foi possível observar que em nenhum dos critérios avaliados (cobertura, número de casos de teste e número de defeitos encontrados) foi possível encontrar a normalidade. Ou seja, o *p-value* da distribuição para todos os casos foi menor que 0,05. O resultado do teste pode ser visualizado na Tabela 5.6.

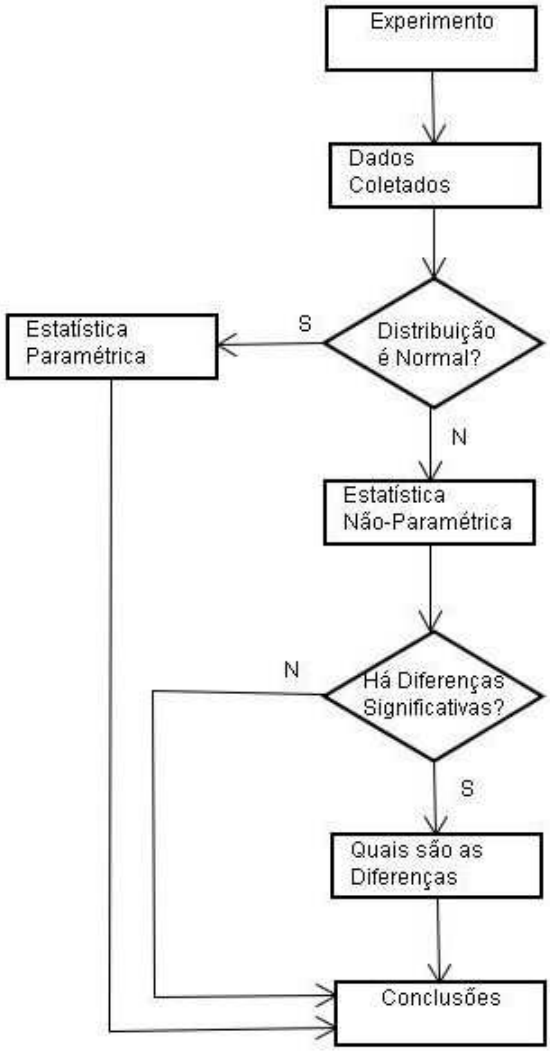

Figura 5.2: *Processo estatístico.*

Assim, a distribuição apresentada pelos dados vinculados as hipóteses é conhecida como distribuição não-normal.

Após descobrir a não-normalidade de uma distribuição é necessário analisar os dados mais cuidadosamente por meio da estatística não-paramétrica. Caso a normalidade fosse encontrada, testes como T-Student, NOVA e ANOVA [46] poderiam ser utilizados para realizar a análise dos dados. Como os dados seguem a distribuição não-paramétrica é necessário trabalhar com estatísticas que são robustas a normalidade, ou seja, não haverá perda de significância durante a análise e interpretação dos resultados. Vale ressaltar que devido a distribuição não-normal, a mediana é a medida de tendência central utilizada para sintetizar os dados. Caso fosse utilizada a média, valores muito altos ou muito baixos poderiam comprometer os valores apresentados pela média comum. A mediana é uma medida de tendência central que utiliza a classificação ou rank para ser avaliada, portanto os valores muito altos ou muito baixos não exercem influência sobre os resultados de tendência central.

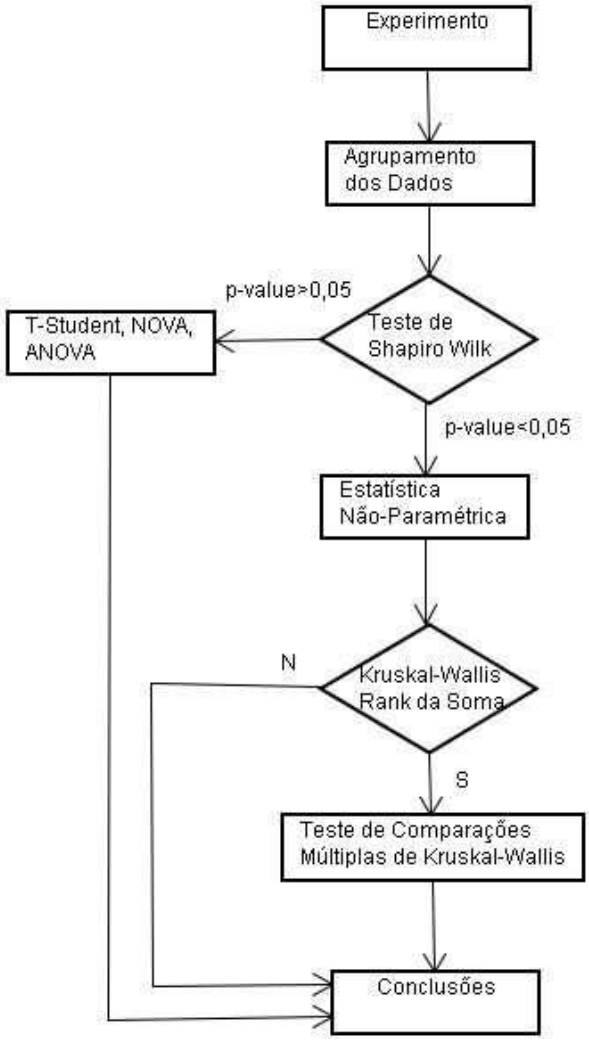

Figura 5.3: *Método estatístico adotado.*

O teste de Kruskal-Wallis [46] é um teste robusto a normalidade, e é utilizado para verificar se há diferenças significativas entre as medianas das técnicas avaliadas. Esse teste realiza o rank dos valores e por meio da mediana acusa se há diferenças significativas entre as amostras. A diferença é observada pelo *p-value* da comparação das três técnicas, quando o *p-value* possuir valor maior que 0,5 quer dizer que não há diferenças significativas entre os critérios comparados. A aplicação desse teste para os critérios avaliados em cada técnica podem ser visualizados na Tabela 5.7.

Após concluir que existe diferenças entre as técnicas é necessário descobrir quais são essas diferenças. Assim o teste de de Múltipla Comparação de Kruskal-Wallis [46] pode ser utilizado para verificar quais são as diferenças significativas entre o rank de três ou mais fatores. Este teste também é robusto a normalidade e a quantidade de amostras. O teste realiza comparações das combinações duas a duas técnicas avaliadas. O resultado da aplicação do teste de Múltipla Comparação de Kruskal-Wallis pode ser analisado na Tabela 5.8. Vale ressaltar que para todas as estatísticas utilizadas foi adotada a margem

Tabela 5.6: *Teste de Normalidade - Shapiro Wilk*

Resultado/Critérios	Todos-Nós	Casos de Teste	Erros
p-value	0,0001	0,0082	0,0002
Distribuição Normal	Não	Não	Não

Tabela 5.7: *Teste de Kruskal-Wallis - Rank Sum Test*

Critérios/Técnicas	*ad hoc*	Funcional	Estrutural	*p-value*	Há Diferença Significativa
Todos-Nós	55,5	73,0	61,0	0,000019	Sim
Casos de Teste	8,0	11,0	8,0	0,009148	Sim
Erros	2,0	2,0	2,0	0,930200	Não

Tabela 5.8: *Teste de Comparação Múltipla de Kruskal-Wallis*

Todos-Nós	Diferença Observada	Diferença Crítica	Diferença
ad hoc X Estrutural	37,531753	20,86848	Sim
ad hoc X Funcional	3,86463	19,95248	Não
Estrutural X Funcional	33,667123	20,96088	Sim
Casos de Teste	Diferença Observada	Diferença Crítica	Diferença
ad hoc X Estrutural	25,485814	20,69736	Sim
ad hoc X Funcional	5,804322	19,56142	Não
Estrutural X Funcional	19,681492	20,51164	Não
Defeitos	Diferença Observada	Diferença Crítica	Diferença
ad hoc X Estrutural	4,3252033	20,500141	Não
ad hoc X Funcional	3,8666667	19,48089	Não
Estrutural X Funcional	0,4585366	20,31164	Não

de acerto de 95%, ou seja, as amostras tem apenas 5% de chances de desviarem dos resultados apresentados na respecitva população.

- **Redução de Dados**

Os critérios para redução de dados utilizados no experimento foram necessários apenas nos casos em que as informações enviadas pelos participantes aos organizadores do experimento não estavam devidamente preenchidas ou não foram respondidas. Nesses casos foram descartadas essas amostras deixando apenas as amostras que os participantes tiveram a preocupação no preenchimento completo e executaram os procedimentos de forma adequada. Cerca de 20% dos dados foram descartados para não comprometer os resultados do estudo experimental. Os dados apresentados na Tabela 5.4 já contemplam essa redução. A tabela completa, sem a redução, pode ser vista no Apêndice A.9.

- **Teste de Hipóteses**

Os testes de hipóteses apresentados pelo processo estatístico são apresentados na Tabela 5.5, e no final dessa seção é apresentado o teste da hipótese do pacote de experimentação proposto pelo trabalho.

A primeira hipótese necessária avaliar é a apresentada pelo teste de normalidade utilizando o Shapiro-Wilk Test, vide Tabela 5.6. Em todos os critérios o *p-value* apresentado foi inferior a 0,05, portanto H_0 é rejeitada, ou seja, a distribuição não é normal.

A segunda hipótese é definida pelo Rank Sum Test Kruskal-Wallis [46] que apresenta por critério quais técnicas são consideradas diferentes estatísticamente. Nos critérios de cobertura e casos de teste o *p-value* apresentou valor inferior a 0,05, ou seja, há diferença entre as técnicas (*ad hoc*, funcional e estrutural) nesses dois critérios. Ou seja, a hipótese H_2 é aceita. No critério quantidade de defeitos revelados o *p-value* apresentado foi superior a 0,05, ou seja, não há diferenças entre as técnicas nesse critério, a hipótese H_2 é rejeitada para esse critério.

A terceira hipótese e última hipótese busca mostrar a diferença entre os critérios que tiveram as hipóteses H_2 aceitas, ou seja, comparações duas a duas são realizadas, e a diferença observada pela aplicação do Teste de Comparação Múltipla de Kruskal-Wallis apresentaram valores mais significantes (na qual há diferença) no critério de cobertura entre as técnicas *ad hoc* X estrutural e funcional X estrutural cuja hipótese H_4 foi aceita.

No critério de quantidade de casos de testes houve diferença significativa na comparação das técnicas *ad hoc* X estrutural, na qual também foi aceita H_4. E por fim no critério de número de defeitos revelados o não houve diferença significativa entre as comparações, ou seja, foi rejeitada a hipótese H_4, confirmando a rejeição da hipótese H_2. Para as comparações com menor significância foram avaliadas no critérios de cobertura entre as técnicas *ad hoc* X funcional, na qual rejeitou-se a hipótese H_4 para essa comparação. Para o critério de quantidade de casos de teste as técnicas *ad hoc* X funcional e funcional X estrutural também tiveram a hipótese H_4 rejeitadas.

Para finalizar os testes e permitir uma conclusão sobre o estudo do presente trabalho, é necessário realizar o teste das hipóteses apresentadas na definição do pacote de experimentação. Conforme mostrado anteriormente não há diferenças entre as técnicas avaliadas no critério quantidade de defeitos revelados, sendo assim a hipótese H_{02} foi aceita e consequentemente a hipótese H_{12} rejeitada. Isso pode ser observado pela média geral dos dados que no caso de defeitos revelados ficou em 2,3 para a técnica *ad hoc*, 2,1 para a técnica funcional e 2,3 para a técnica estrutural, ou seja, praticamente a mesma quantidade de defeitos foram revelados.

Na avaliação das técnicas no critério quantidade de casos de teste houve diferença estatística significativa na comparação entre as técnicas avaliadas, os valores médios apresentados por cada técnica nesse critério foram 8,9 para a técnica *ad hoc*, 9,1 para a técnica funcional e 12 para a técnica estrutural. Com a apresentação desses valores é possível aceitar a hipótese H_{13} e consequentemente rejeitar a hipótese H_{03}, mostrando que há diferenças entre as técnicas e que a técnica estrutural apresenta melhor valores nesse critério.

Já no critério cobertura de comandos a avaliação das técnicas também apresentou diferença significativa entre as técnicas avaliadas. A média apresentada pela técnica *ad hoc* foi de 55% de cobertura, no caso da técnica funcional o percentual ficou em 57% e a técnica estrutural obteve o maior percentual de cobertura de código com 72%. Dessa forma hipótese H_{12} torna-se aceita por existir diferença estatística significativa entre as técnicas e por apresentar o maior valor de cobertura para a técnica estrutural, consequentemente rejeita-se a hipótese alternativa H_{02}.

5.2 Análise dos Dados

Além do teste das hipóteses do experimento, também é importante realizar uma análise do cruzamento dos dados coletados do perfil dos participantes com os resultados e hipóteses válidas ou inválidas. Alguns fatos interessantes sobressaem nesse cruzamento de informações e propicia maiores argumentos na criação da estratégia de teste proposta por esse trabalho para a garantia de uma maior qualidade aos produtos Java ME.

Características dos participantes do experimento como idade, experiência profissional e acadêmica, experiência em linguagens de programação é demonstrado na Figura 5.4.

Conforme apresentado pela Figura 5.4, a média de idade ficou em média de 24 anos de idade, lembrando que os participantes eram em sua grande maioria estudantes de graduação e pós-graduação e profissionais que atuam no mercado de trabalho. A experiência profissional em empresas privadas ou públicas ficou em média com 3 anos de experiência. Como alguns dos participantes estavam envolvidos em atividades acadêmicas a média de experiência nesse quesito ficou em 2,5 anos. O único pré-requisito do curso era possuir conhecimentos na linguagem de programação *Java* a experiência nessa linguagem ficou em média 1,5 anos e em outras linguagens em 1,7 anos. Um fato interessante que deve ser levado em consideração é que a média de experiência com testes para todos os participantes ficou em torno de 0,8 anos, ou seja, mesmo tendo experiência de 3 anos no mercado de trabalho ou academia, a experiência desses profissionais e/ou estudantes no teste de software é relativamente pequeno, chegando a um terço de sua experiência.

Um questionário sobre atividades desempenhadas pelos participantes em meio acadêmico ou industrial foi criado visando traçar o perfil e o conhecimento dos participantes durante o experimento e o curso oferecido. Na Figura 5.5 é possível observar que a grande maioria, 78% declararam que gostam de executar a atividade de teste de software. Especificamente sobre a técnica funcional apenas 25% dos participantes declaram conhecer a técnica funcional, e apenas 20% a aplicaram. Sobre a técncia estrutural apenas 80% dos participantes informaram não conhecer a técnica estrutural e dos que conheciam apenas 10% tiveram oportunidade para aplicá-la na prática. Com essas informações

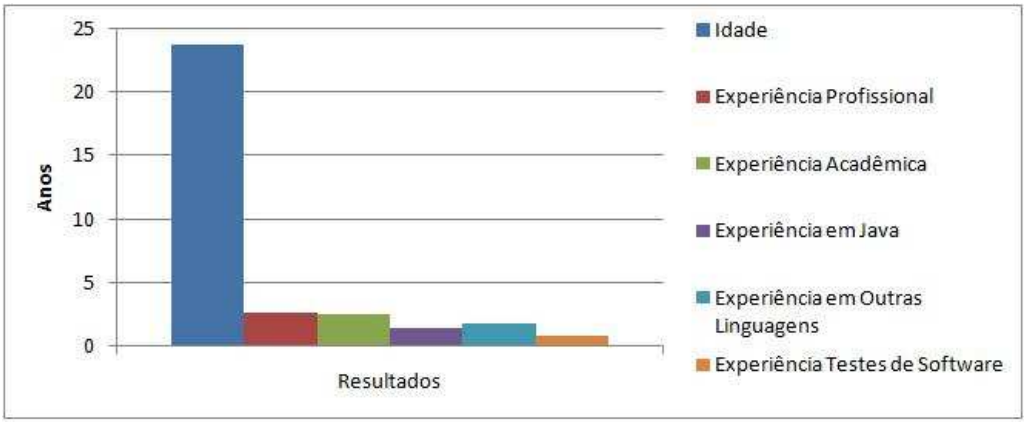

Figura 5.4: *Perfil dos Participantes.*

é possível observar que a técnica funcional é mais conhecida entre os participantes do que a técnica estrutural. E que mesmo alguns conhecendo uma das técnicas, poucos tiveram oportunidade para aplicar os conhecimentos adquiridos das mesmas.

Apesar da técnica funcional ser mais conhecida pelos participantes do que a técnica estrutural, o resultado dos estudos mostram que o uso da técnica estrutural propicia melhores resultados na cobertura e na criação de casos de teste. Um fator que pode explicar esse melhor resultado pela técnica estrutural é o uso da ferramenta JaBUTi/ME que oferece um ambiente cujo testador possui informações de quanto (em percentual) ainda falta para exercitar do programa, isso induz o testador a criar novos casos de testes que passem pelos pontos ainda não executados do software. A facilidade de utilizar a ferramenta também é um fator positivo, pois a JaBUTi/ME era novidade para todos participantes.

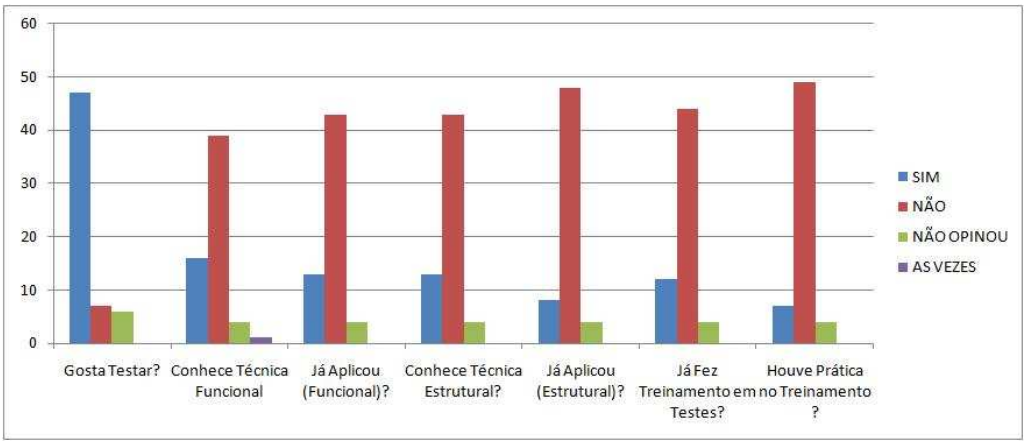

Figura 5.5: *Resultado do Questionário.*

Os participantes também foram questionados se já haviam participado de algum tipo de treinamento de teste de software, cerca de 73% dos participantes nunca haviam participado de desse tipo de treinamento. E dos 23% que participaram apenas 11% tiveram atividade prática, conforme ilustrado na Figura 5.5. Com esses dados é possível analisar

que a experiêcia dos participantes foi pouca em teste de software o que pode justificar a baixa quantidade de defeitos revelados por todas as três técnicas avaliadas, em que a estatística mostrou que para esse experimento não houve diferenças significativas entre as técnicas nesse quesito. Por outro lado, os participantes teceram elogios e apoiaram a iniciativa de haver um curso de testes que mostram diferentes técnicas e há tempo para a execução na prática. E que é muito raro ocorrer treinamentos gratuitos e aberto a comunidade como os que foram realizados nesse trabalho.

Um fator interessante nesse trabalho foi que os participantes adquiriram conhecimentos na área de teste de software, sabendo diferenciar as três técnicas apresentadas no curso, aplicaram os conhecimentos de cada técnica separadamente e os instrutores conseguiram uma massa de dados coletados importante para ajudar a entender as dificuldades de se testar um software Java ME além de apresentar estatisticamente a avaliação entre as técnicas e o apoio que a ferramenta JaBUTi/ME apresentou nesse contexto.

Os dados capturados na aplicação de cada técnica por parte dos participantes permitiram que a ferramenta JaBUTi/ME fosse utilizada tanto para os participantes praticarem o uso da técnica estrutural, quanto para as coletar informações de cobertura durante a execução das outras técnicas. As informações de cobertura de cada programa podem ser analisadas nos gráficos ilustrados na Figura 5.6, Figura 5.7 e Figura 5.8.

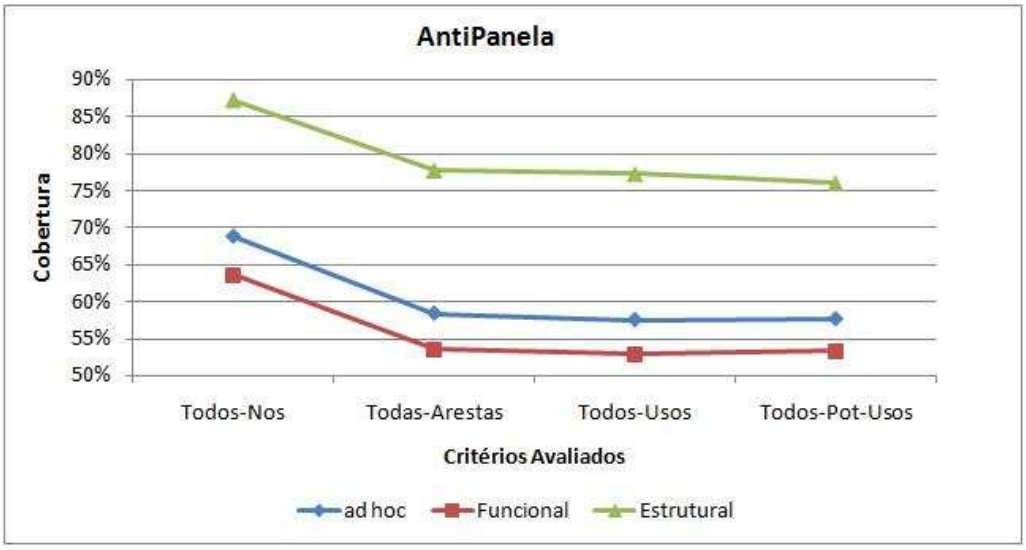

Figura 5.6: *Avaliação do Programa Antipanela por Técnica.*

Na Figura 5.6 é possível observar que no resultado final das três replicações do pacote de experimentação no programa AntiPanela, o mais simples dos programas do pacote, a técnica estrutural apresentou maiores percentuais de cobertura em todos os critérios avaliados pela ferramenta JaBUTi/ME e a técnica *ad hoc* ficou à frente da técnica funcional também nesses critérios.

No programa CarManager que possui complexidade média (dentre os três utilizados no pacote), a técnica estrutural também apresentou melhores percentuais de cobertura

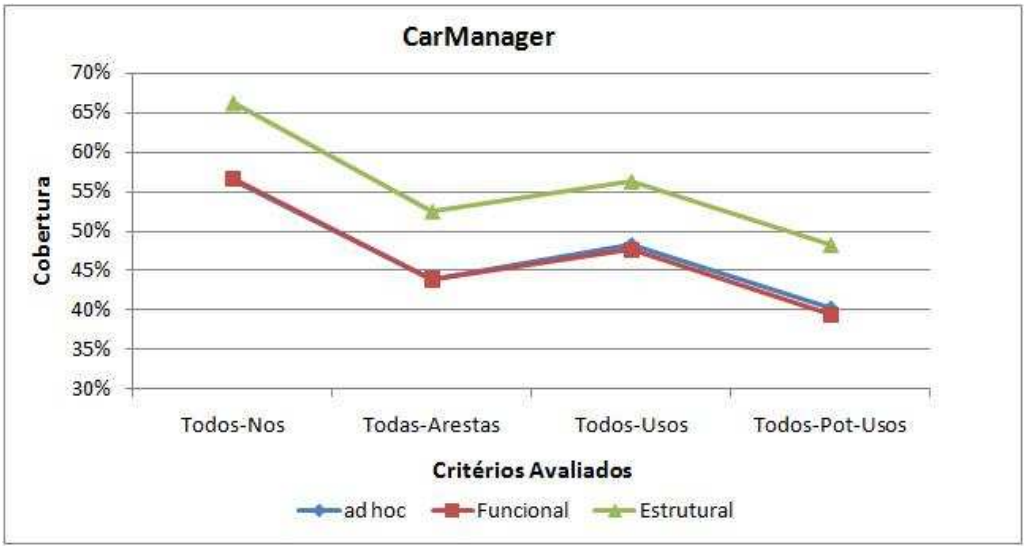

Figura 5.7: *Avaliação do Programa CarManager por Técnica.*

nos critérios apoiados pela JaBUTi/ME, porém as técnicas *ad hoc* e funcional apresentaram resultados muito próximos, praticamente empatadas do ponto de vista estatístico.

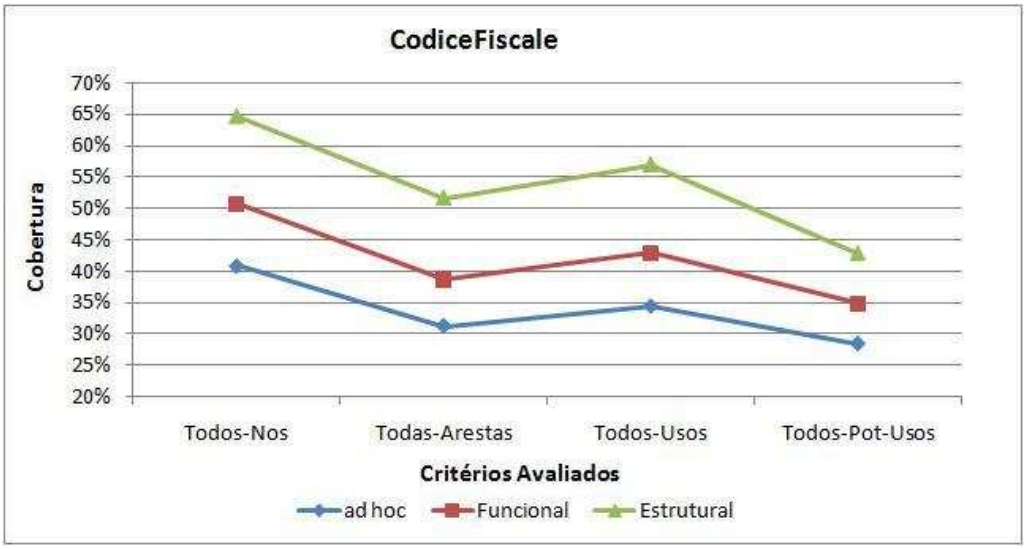

Figura 5.8: *Avaliação do Programa CodiceFiscale por Técnica.*

Analisando o programa CodiceFiscale, o mais complexo analisados no pacote de experimentação, ele também apresentou maior cobertura na técnica estrutural nos critérios apoiados pela JaBUTi/ME e a técnica funcional obteve maiores percentuais que a técnica *ad hoc*.

Partindo desses dados é possível observar que a técnica estrutural apresentou neste pacote de experimentação, os melhores resultados em percentuais de cobertura em todos os critérios analisados. Também é possível observar que de acordo com a complexidade dos programas as técnicas *ad hoc* e funcional se intercalam nos percentuais de cobertura.

Os casos de teste possuem fundamental importância na definição da qualidade de um produto de software, é por meio deles que consegue-se detectar a presença de defeitos nos programas, sendo assim é importante analisar a quantidade e qualidade dos casos de teste criados, além desse quesito ser de fundamental importância para a definição de uma da estratégia de teste para software de dispositivos móveis. Conforme ilustrado na Figura 5.9 é possível observar que a técnica estrutural consegue prover uma maior quantidade de casos de testes para um mesmo programa em relação as outras técnicas *ad hoc* e funcional. Também é possível analisar que a técnica funcional foi capaz de criar uma leve diferença superior, em média, na quantidade de casos de testes, quando relacionada com a técnica *ad hoc*.

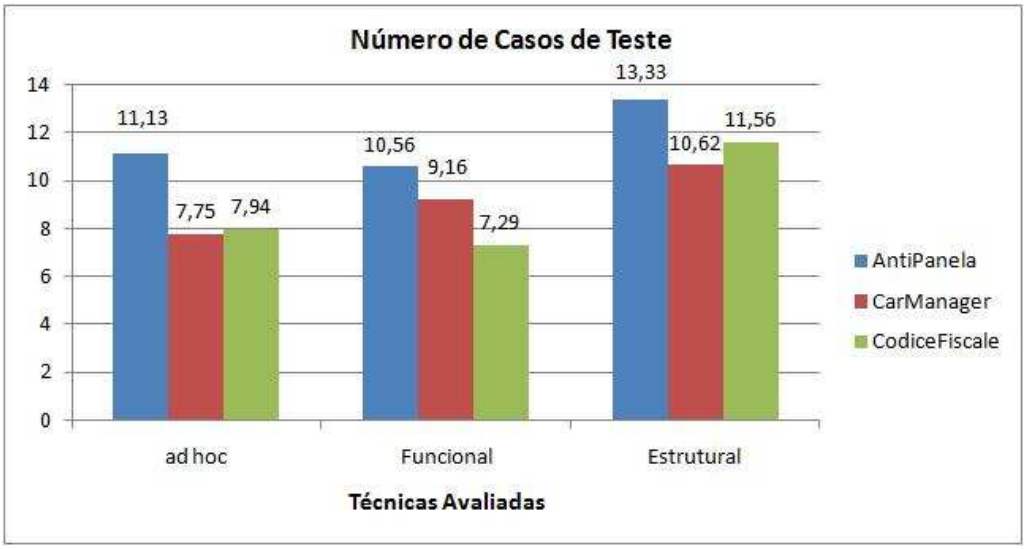

Figura 5.9: *Número de Casos de Teste por Técnica.*

Como o objetivo do teste de software é encontrar defeitos nos programas, é de fundamental importância avaliar o número de defeitos que foram revelados nos programas que fazem parte deste pacote de experimentação. Na Figura 5.10 é possível observar graficamente a quantidade de defeitos que foram revelados durante a execução dos testes nas replicações do experimento. Nesse quesito a análise estatística mostrou que as três técnicas avaliadas são equivalentes, embora seja possível realizar a análise dos dados por programa. No gráfico é possível observar que cada técnica revelou uma maior quantidade de defeitos em um programa. A técnica *ad hoc* conseguiu revelar mais defeitos no programa AntiPanela, que possui a menor complexidade dentre os três. Já a técnica funcional conseguiu revelar um número maior de defeitos no programa de complexidade média, dentre os três do pacote, o programa CarManager. E a técnica estrutural revelou uma maior quantidade de defeitos no programa de maior complexidade, dentre os avaliados, o programa CodiceFiscale. Vale ressaltar que foram inseridos em cada programa uma quantidade média de 10 (dez) defeitos. Tal resultado evidencia o aspecto complementar das técnicas e critérios de teste e enfatiza que boas estratégias de teste

devem combinar critérios de diferentes técnicas de modo a maximizar a capacidade de detecção de defeitos do conjunto de teste resultante.

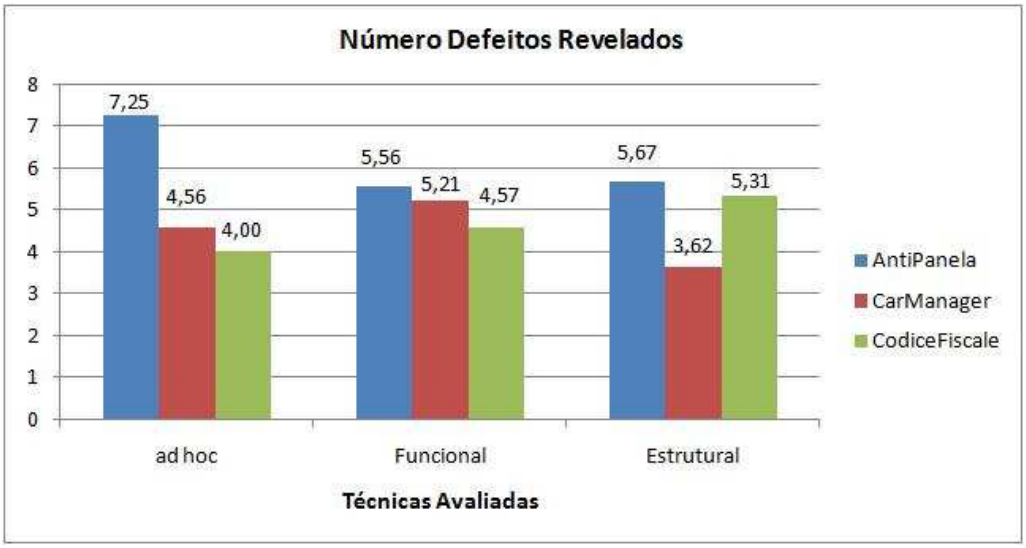

Figura 5.10: *Número de Defeitos Encontrados por Técnica.*

A tabela com os dados mais detalhados que foram responsáveis pela produção dos gráficos analisados acima, pode ser visualizado na Tabela 5.9. Vale ressaltar que as coberturas não foram atingidas em sua totalidade devido algumas características e regras propostas pelo pacote de experimentação, como a igualdade de tempo na aplicação dos testes em cada técnica.

Algumas informações para melhoria do pacote de experimentação foram coletadas por meio dos Formulários 4 e 5, que encontram-se nas Seções A.3.4 e A.3.5 do Apêndice A e podem ser conferidas na Tabela 5.10. De acordo com os resultados do Formulário 5, 72% dos alunos declararam que não possuía domínio na disciplina de testes. Após a aplicação do experimento 88% dos participantes declararam que se sentem seguros para aplicar as técnicas aprendidas durante o curso e 100% informaram que o curso agregou novos conhecimentos para a vida acadêmica ou profissional dos participanes. Por meio deste formulário os participantes realizavam sua auto-avaliação, nesse quesito a média ficou em 7,9 (escala de 0 a 10) e a nota para o curso como um todo ficou em média de 8,6 (escala de 0 a 10). Ou seja, os participantes gostaram do curso, a média de auto-avaliação e de avaliação do curso foram muito boas, portanto os objetivos traçados para a satisfação dos participanes foram atingidos.

Diante dos resultados apresentados, da análise estatística das técnicas e do perfil dos participantes é proposta uma estratégia de teste de software para dispositivos móveis utilizando Java ME como plataforma de implementação, essa estratégia é descrita a seguir.

Tabela 5.9: *Cobertura Média dos Critérios e Número de Casos de Teste*

	AntiPanela			**CarManager**			**CodiceFiscale**		
Critérios de Teste/Técnica	ad hoc	Funcional	Estrutural	ad hoc	Funcional	Estrutural	ad hoc	Funcional	Estrutural
Todos-Nos	68,8%	63,5%	87,3%	56,6%	56,6%	66,3%	40,9%	50,7%	64,7%
Todas-Arestas	58,4%	53,5%	77,7%	43,9%	43,8%	52,5%	31,3%	38,8%	51,7%
Todos-Usos	57,5%	52,9%	77,2%	48,2%	47,6%	56,3%	34,5%	43,0%	57,0%
Todos-Pot-Usos	57,6%	53,3%	75,9%	40,2%	39,3%	48,3%	28,5%	34,9%	42,8%
Casos de Teste	11,13	10,56	13,33	7,75	9,16	10,62	7,94	7,29	11,56
Erros	7,25	5,56	5,67	4,56	5,21	3,62	4,00	4,57	5,31
Erros Inseridos	9,00			10,00			10,00		

Tabela 5.10: *Avaliação do Curso e Auto-Avaliação*

Avaliação do Curso e Auto-Avaliação	Possuia domínio em testes?	Agregou conhecimento?	Seguro para aplicar o conhecimento?	Pretende aplicar o conhecimento?	Seguir carreira em Teste?	Nota para seu aprendizado no curso	Nota para o curso
SIM	28%	100%	88%	100%	44%	7,9	8,6
NÃO	72%	0%	12%	0%	56%		

5.3 Estratégia de Teste

Considerando as técnicas avaliadas, conforme análise estatística e os resultados obtidos no presente trabalho uma estratégia de teste de software para dispositivos móveis utilizando Java ME é proposta. Vale ressaltar que cada técnica avaliada possui suas características e em geral busca garantir mais qualidade no produto e que nada impede de utilizá-las em conjunto, em que cada uma foca em um objetivo e assim a qualidade de cada característica do produto é alcançada.

Para os programas mais intuitivos e com mais recursos visuais do que de processamento (cálculos, lógicas e algoritmos complexos) a técnica *ad hoc* é mais recomendável. O uso da técnica funcional como complemento nesse tipo de software é de grande importância para que a visão final do usuário seja avaliada, buscando satisfazer e até superar suas expectativas.

Para os programas que possuem cálculos, lógicas e algoritmos mesmo não sendo complexos é recomendado o uso da técnica funcional como garantia da qualidade buscada pelo usuário e das avaliação da estrutura do software por meio da técnica estrutural. Caso

o aplicativo tenha partes essenciais do programa a técnica estrutural não deve deixar de ser aplicada visando obter uma máxima cobertura do critério Todos-Nós.

Por tanto, para testar produtos Java ME é importante primeiramente avaliar as características principais dos produtos e então traçar uma das estratégias com pelo menos duas das técnicas. *Ad hoc* e funcional para programas simples e intuitivos, e funcional e estrutural para programas mais complexos. De qualquer modo, sempre que exista tempo e custo suficientes, investir na atividade de teste e avaliar a qualidade dos conjunto de teste produzidos por meio de critérios estruturais contribui para reduzir o número de defeitos presentes no produto antes de sua liberação e melhorar a satisfação do cliente.

5.4 Considerações Finais

Neste capítulo foi apresentado o pacote de experimentação proposto, detalhando as informações de sua organização, preparação, execução e análise estatística dos dados. Uma estratégia de testes baseada nos dados e nas conclusões estatísticas do trabalho foi proposta a fim de buscar agregar mais qualidade nos produtos de software móveis disponíveis no mercado voltados para a plataforma Java ME.

CAPÍTULO 6

Conclusão

A atividade de teste possui em geral, uma grande responsabilidade na garantia da qualidade dos produtos de software. Muitos métodos, normas e processos foram definidos para normatizar os passos para a criação de software, em muitos deles é possível averiguar que a atividade de teste é obrigatória na maioria das vezes. O principal objetivo dessa atividade na construção do software é permitir a satisfação do cliente ou usuário na utilização do produto de software.

Muitos produtos tecnológicos surgem a cada dia, dentre eles estão os dispositivos móveis como celulares, PDAs e outros e inerentes a eles surgem os software específicos para esse tipo de dispositivos. Esses dispositivos possuem características próprias como limitações de armazenamento e processamento. Por meio deste trabalho foi possível observar que essas restrições dos dispositivos móveis não permitem que ferramentas que utiliza-se para testar os software convencionais sejam utilizadas nesses tipos de dispositivos. E que testar um software construído para dispositivos móveis não é uma tarefa simples.

Algumas técnicas de testes foram apresentadas, dentre elas pode-se destacar a técnica *ad hoc* que possui como característica principal a experiência do testador para realização dos testes ou o uso de algoritmos aleatórios para a criação de casos de teste. A técnica funcional foi outra a ser apresentada, ela destaca o uso do software como uma caixa-preta em que não é seu objetivo saber como o produto foi estruturado e construído, ela apenas se preocupa com a entrada de dados e compara o resultado ou saída do produto de software. A última técnica apresentada foi a técnica estrutural que, ao contrário da funcional, é voltada para a avaliação da estrutura interna do software, como foi implementado e produzido.

Como essas técnicas são bastante utilizadas para software convencionais, o trabalho buscou avaliar a aplicação dessas técnicas no teste de software dos dispositivos móveis. Para realizar essa avaliação os conceitos da engenharia de software experimental foram utilizados, e principalmente o pacote de experimentação. Essa avaliação também foi apoiada pela ferramenta de testes JaBUTi/ME que oferece os recursos necessários para que o teste de software seja executado no próprio dispositivo móvel ou em emuladores.

O pacote de experimentação ajudou a organizar e criar as diretrizes detalhadas para a execução e replicação do experimento. O pacote proposto foi replicado três vezes e os dados coletados permitiram uma análise estatística e as validações das hipóteses. Alguns dos resultados já eram esperados pelos autores deste trabalho, embora outros foram revelações importantes para a criação de uma base de dados estatísticos, e que estudos futuros nessa área possam utilizá-los.

Dos dados coletados foi possível concluir neste trabalho, que a técnica estrutural permitiu a criação de conjuntos de teste que apresentaram os melhores resultados nos critérios de cobertura de código e maior número de casos de teste criados. Essa era uma das informações esperadas, pois em geral, quando se tem um maior número de casos de teste projetados para cobrir trechos de código ainda não executados a cobertura tende a crescer, como o ocorrido. Uma revelação importante foi que estatísticamente as três técnicas avaliadas são significativamente equivalentes na revelação de defeitos. Como foi possível observar o tempo utilizado para execução dos teses em cada técnica foi o mesmo, assim independente da utilização de técnicas simples ou complexas no mesmo intervalo de tempo não mostrou diferenças entre as técnicas nesse quesito. Embora a expectativa dos autores seja que a técnica estrutural que é considerada mais complexa em sua execução, em ambiente real mais tempo seria considerado a essa técnica possivelmente mais erros seriam revelados. Isso é importante pois essa técnica fornece recursos para a criação de novos casos de testes com o apoio de ferramentas como a JaBUTi/ME, ao contrário das outras técnicas que param ao satisfazer o testador ou ao concluir os casos de teste definidos na especificação do programa.

A Ferramenta JaBUTi/ME apoiou de forma efetiva a técnica estrutural, a maior prova disso são que os melhores resultados foram obtidos nessa técnica, ou seja, dos poucos recursos utilizados pelos participantes que possuíam pouca experiência em testes e a grande maioria nunca haviam utilizado a ferramenta, ela mostrou ser fácil de trabalhar e conseguiu agregar mais qualidade no software, no sentido de comprovar quais partes do código foram ou não executadas durante os testes. Levando em consideração os outros quesitos de cobertura e criação de casos de teste pois, no quesito defeitos as técnicas se mostraram estatisticamente equivalentes.

As análises parciais dos dados coletados neste trabalho, da primeira e da segunda replicação podem ser conferidos nos artigos publicados [20] e [21], respectivamente. Embora o tamanho das amostras individuais sejam relativamente pequeno os resultados demonstram poucas variações entre elas. Este trabalho acumulou os resultados adquiridos nas três replicações e com uma amostra maior é possível obter maior confiabilidade estatística nos resultados.

Por fim, este trabalho proporcionou mostrar a importância na realização de experimentos na engenharia de software para entender comportamentos e características

de software para dispositivos móveis (Java ME). Muitas lições foram aprendidas durante o seu desenvolvimento e, dentre as principais contribuições resultantes do mesmo tem-se:

- a criação de uma base de dados inicial sobre teste de programas Java ME;
- a publicação de artigos mostrando a comunidade científica o trabalho realizado;
- a criação e disponibilização de um pacote de experimentação para Java ME viabilizando sua replicação por outras pessoas interessadas;
- o oferecimento de cursos de teste de software como atividade de extensão à comunidade;
- a participação em palestras e oficinas, disseminando o conhecimento adquirido;
- as discussões sobre a melhoria na qualidade do software produzidos para dispositivos móveis;
- a identificação de melhorias que podem ser efetuadas na ferramenta de teste utilizada; e
- a criação de uma estratégia de teste preliminar voltada para produtos de software Java ME.

Tais resultados abrem espaço para que novos trabalhos sejam desenvolvidos sendo alguns deles descritos a seguir.

6.1 Trabalhos Futuros

Primeiramente, um pacote de experimentação é definido para padronizar a coleta de dados em determinada área do conhecimento. Nesse sentido, o pacote proposto pode ser estendido para avaliar as mesmas técnicas e critérios de teste em uma quantidade cada vez maior de programas Java ME. Isso favoreceria o aumento na base de conhecimento sobre esses critérios aplicados a programas Java ME e melhoraria, incrementalmente, a significância estatística dos resultados obtidos.

O envio de dados entre o dispositivo móvel e o servidor de teste é um gargalo quando softwares Java ME com grande interatividade são testados. Desse modo, investigar formas alternativas de instrumentação e transferência de dados entre os dispositivos e o servidor de teste é necessário para atender a demanda para o teste desses produtos.

Além disso, os dispositivos móveis veem evoluindo em armazenamento de dados e processamento. Nesse sentido também é possível estender o pacote proposto e aplicá-lo em dispositivos móveis reais permitindo avaliar e comparar com os resultados obtidos neste trabalho. Com essa evolução dos dispositivos é importante buscar utilizar os novos recursos, talvez daqui algum tempo seja possível executar a própria JaBUTi/ME nos dispositivos móveis e quando esse tempo chegar será necessário aplicar estudos para saber como se comporta a ferramenta nesses tipos de aparelhos.

Não apenas os celulares, PDAs, dentre outros veem evoluindo, existem novas tecnologias como o JavaCard e o JavaTV que também possuem softwares inerentes a esses dispositivos e por consequencia há necessidade de testá-los e estabelecer uma estratégia que garanta a qualidade dos softwares que esses dispositivos usam. Avaliar a aplicação da JaBUTi/ME nesses novos contextos ou até mesmo propor uma versão alternativa para atendê-los é uma linha de pesquisa que pode ser desenvolvida.

A criação de outro pacote de experimentação que visa avaliar outras estratégias de teste também torna-se viável após a execução deste trabalho e então realizar uma comparação entre as pesquisas alcançadas.

A tecnologia móvel está em constante evolução não se pode prever o que virá nos próximos cinco anos, mas uma coisa é certa, experimentos deverão ser realizados para entender melhor como essas tecnologias influenciam em nossas vidas e entendê-las em essência para que os produtos de software para os dispositivos móveis tenham excelência em seus objetivos e tenham cada vez mais produtos móveis interessantes e úteis.

Referências Bibliográficas

[1] **NBR ISO 9000**. Associação Brasileira de Normas Técnicas (ABNT).

[2] **NBR ISO 9001**. Associação Brasileira de Normas Técnicas (ABNT).

[3] **NBR ISO 9002**. Associação Brasileira de Normas Técnicas (ABNT).

[4] **NBR ISO 9003**. Associação Brasileira de Normas Técnicas (ABNT).

[5] **ISO/IEC 15504 : Information technology – Process Assessment**. International Organization for Standardization, 2006.

[6] ANATEL. **Brasil tem mais de 147 milhões de celulares**. Technical report, Agência Nacional de Telecomunicações, http://www.anatel.gov.br/Portal/exibirPortalInternet.do, Dezembro 2008. 2008.

[7] BASILI, V. R. **The experimental paradigm in software engineering**. *Springer-Verlag, Lecture Notes in Computer Science*, p. 706, 1993.

[8] BASILI, V. R.; PERRICONE, B. T. **Software errors and complexity: An empirical study**. 27(1):42–52, Jan. 1994.

[9] BASILI, V. R.; SELBY, R. W. **Comparing the effectiveness of software testing strategies**. *IEEE Trans. Softw. Eng.*, 13(12):1278–1296, 1987.

[10] BASILI, V. R.; SHULL, F.; LANUBILE, F. **Building knowledge through families of experiments**. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[11] BECK, K. **Junit pocket guide**. Technical report, 2004.

[12] BEIZER, B. **Software testing techniques**. Van Nostrand Reinhold Company, 2nd ed. edition, 1990.

[13] CHAIM, M. J. **Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseados em analise de fluxo de dados**. Master's thesis, DCA/FEE/UNICAMP – Campinas, SP, Brasil, 1991.

[14] CHAIM, M. L. **Depuração de Programas Baseada em Informação de Teste Estrutural**. Tese de doutorado, Faculdade de Engenharia de Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, SP, Brasil, Nov. 2001.

[15] CHEVALLEY, P. **Applying mutation analysis for object-oriented programs using a reflective approach**. In: *8th Asia-Pacific Software Engineering Conference – APSEC'01*, p. 267–272, Macao, China, Dec. 2001. IEEE Computer Society Press.

[16] CHOI, B. J.; DEMILLO, R. A.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A. P.; OFFUTT, A. J.; PAN, H.; SPAFFORD, E. H. **The mothra toolset**. *Proceedings of de 22nd Annual Hawaii international Conference on Systems Sicences*, p. p. 275–284, 1989. Koa, Havaí.

[17] CONTE, T. U.; TRAVASSOS, G. H. **Processos de desenvolvimento para aplicações web: Uma revisão sistemática**. In: *Anais do XI Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia 2005)*, 2005.

[18] DAHM, M. **Byte code engineering**. Java-Information-Tage 1999 (JIT'99), Sept 1999.

[19] DE CAMPOS AVILLANO, I. **Algoritmos e Pascal**. 2a edicao edition, 2006.

[20] DE DEUS, G. D.; VINCENZI, A. M. R.; DELAMARO, M. E. **Criação de um pacote de experimentação para a avaliação de critérios de teste estruturais em produtos j2me**. In: *V Experimental Software Engineering Latin American Workshop - ESELAW'2008*, p. 1–10, Salvador, BA, Nov. 2008.

[21] DE DEUS, G. D.; VINCENZI, A. M. R.; DE LUCENA, F. N.; DELAMARO, M. E. **Avaliação da qualidade de produtos j2me por meio do uso de pacotes de experimentação**. *VIII - Simpósio Brasileiro de Qualidade de Software (SBQS)*, p. p. 264–278, jun 2009.

[22] DELAMARO, M. E. **Proteum: Um ambiente de teste baseado na análise de mutantes**. Master's thesis, ICMC/USP, São Carlos, SP, Brasil, 1993.

[23] DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. Elsevier, Rio de Janeiro, RJ, 2007.

[24] DELAMARO, M. E.; VINCENZI, A. M. R.; MALDONADO, J. C. **A strategy to perform coverage testing of mobile applications**. In: *I International Workshop on Automation of Software Test – AST'2006*, p. 118–124, New York, NY, USA, May 2006. ACM Press.

[25] DÓRIA, E. S. **Replicação de estudos empíricos em engenharia de software**. Master's thesis, ICMC/USP, 2001.

[26] E B. YANG, H. U. **A structural test selection criterion**. *Information Processing Letters*, p. 157–163, 1988.

[27] E C. M. LOTT, E. K. **An empirical evaluation of three defect-detection techniques**. *V European Software Engineering Conference*, p. 362–383, 1995. Londres, UK.

[28] E J. R HORGAN, H. A. **Dynamic program slicing**. *ACM SIGPLAN Notices*, p. 246–256, 1990.

[29] FELIZARDO, K. R. **Cotest – uma ferramenta de apoio à replicação de um experimento baseado em código fonte**. Master's thesis, Universidade Federal de São Carlos - UFSCar, 2003.

[30] FÖLSING, A. **Albert Einstein. A Biography**. Viking, New York, 1997.

[31] FUSARO, P. F. L. V. G. **A replicated experiment to assess requirements inspection techniques**. *Empirical Software Engineering Journal*, 2(1):39–57, 1997.

[32] G. J. MYERS, C. SANDLER, T. B. E. T. M. T. **The Art of Software Testing**. Jon Wiley & Sos, Nova Yourk, NY, EUA, 2 ed. edition, 2004.

[33] GILL, A. **Introduction to the Theory of Finite-State Machine**. Nova York, NY, EUA, 1962.

[34] HETZEL, W. C. **An experimental analysis of program verification methods**. Tese de doutoramento, University of North Carolina at Chapel Hill, 1976.

[35] HÖHN, E. N. **Técnicas de leitura de especificação de requisitos de software: estudos empíricos e gerência de conhecimento em ambientes acadêmico e industrial**. Master's thesis, ICMC/USP, 2004.

[36] IBM. **Rational unified process (rup)**. Página Web, 2008. http://www-01.ibm.com/software/awdtools/rup/.

[37] IEEE. **IEEE Computer Dictionary - Compilation of IEEE Standard Computer Glossaries, 610-1990**, 1990.

[38] JURISTO, N.; MORENO, A. M.; VEGAS, S. **Towards building a solid empirical body of knowledge in testing techniques**. *SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[39] K. ARAKI, Z. F. E. J. C. **A general framework for debugging**. *IEEE Software*, p. p. 14–20, 1991.

[40] LARMAN, G. **Utilizando UML e Padrões**. 2a edição edition, 2004.

[41] LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. 1997.

[42] MA, Y.-S.; KWON, Y.-R.; OFFUTT, J. **Inter-class mutation operators for Java**. In: *13th International Symposium on Software Reliability Engineering - ISSRE'2002*, p. 352–366, Annapolis, MD, Nov. 2002. IEEE Computer Society Press.

[43] MALDONADO, J. C. **Critérios potenciais usos: Uma contribuição ao teste estrutural de software**. Tese de doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, jul 1991.

[44] MENDONÇA, M. G.; MALDONADO, J. C.; DE OLIVEIRA, M. C. F.; CARVER, J.; FABBRI, S. C. P. F.; SHULL, F.; TRAVASSOS, G. H.; HÖHN, E. N.; BASILI, V. R. **A framework for software engineering experimental replications**. In: *XIII IEEE International Conference on Engineering of Complex Computer Systems – ICECCS'2008*, p. 203–212, Belfast, Northern Ireland, Mar./Apr. 2007. IEEE Press.

[45] MEYER, J.; DOWNING, T. **Java Virtual Machine**. 1997.

[46] MONTGOMERY, D. C. **Design and Analysis of Experiments**. John Wiley & Sons, Inc., 5th edition, 2001.

[47] MUCHOW, J. W. **Core J2ME Technology**. 1st edition edition, December 2001.

[48] MYERS, G. J. **A controlled experiment in program testing and code walkthroughs/inspection**. *Communication of the ACM*, p. 21(9):760–768, set. 1978.

[49] MYERS, G. J. **The Art of Software Testing**. Wiley, 1979.

[50] NASCIMENTO, C. A. R. **Para ler Galileu Galilei**. Nova Stella e Educ, 1990.

[51] NEWS, S. **Siam news. society for industrial and applied mathematics**. http://www.cs.clemson.edu/ steve/Spiro/arianesiam.htm, October 1996. Vol. 29. Number 8.

[52] PARA PROMOÇÃO DA EXCELÊNCIA DO SOFTWARE BRASILEIRO, A. **Mps.br – guia de avaliação**, Maio 2006.

[53] PRESSMAN, R. S. **Engenharia de Software**. McGraw-Hill, 6 edition, 2006.

[54] ROBSON, C. **Real World Research: A Resource for Social Scientists and Practitioners-Researchers**. 1993.

[55] RUNESON, P.; HÖST, M. **Guidelines for conducting and reporting case study research in software engineering**. p. 131–164, December 2008.

[56] S. LINKMAN, A. M. R. V. E. J. M. **An evaluation of systematic functional testing using mutation testing**. *7th International Conference on Empirical Assessment in Software Engineering – EASE. The IEE*, abr. 2003.

[57] SCHILDT, H. **C Completo e Total**. Makron Books, São Paulo, 3a edição revisada e atualizada edition, 1996.

[58] (SEI), C. M. S. E. I. **Cmmi for systems engineering/software engineering (cmmi-se/sw)**. Página Web, 2002. http://www.sei.cmu.edu/.

[59] SHULL, F.; MENDONÇA, M. G.; BASILI, V.; CARVER, J.; MALDONADO, J. C.; FABBRI, S.; TRAVASSOS, G. H.; FERREIRA, M. C. **Knowledge-sharing issues in experimental software engineering**. *Empirical Software Engineering*, 9(1-2):111–137, 2004.

[60] V. BASILI, R. W. SELBY, D. H. H. **Experimentation in software engineering**. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, 1986.

[61] VINCENZI, A. M.; DELAMARO, M. E.; MALDONADO, J. C. **JaBUTi/ME – user's guide**. Technical report, UNISANTOS, 2007.

[62] VINCENZI, A. M.; WONG, W. E.; DELAMARO, M. E.; SIMÃO, A. S.; MALDONADO, J. C. **JaBUTi – Java Bytecode Understanding and Testing – user's guide – version 1.0**. Technical report, ICMC/USP, 2004. em preparação.

[63] VINCENZI, A. M. R. **Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação**. Tese de doutoramento, Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, May 2004. Disponível em: http://www.teses.usp.br/teses/disponiveis/55/55134/tde-17082004-122037. Acesso em: 21/10/2004.

[64] VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. **JaBUTi: A coverage analysis tool for Java programs**. In: *XVII SBES – Brazilian Symposium on Software Engineering*, p. 79–84, Manaus, AM, Brazil, Oct. 2003. Brazilian Computer Society (SBC).

[65] WEISER, M. **Programmers use slices when debugging**. *Commun. ACM*, 25(7):446–452, 1982.

[66] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering – an introduction.** Technical report, LundUniversity, Sweden, 2000.

[67] WONG, W. E.; GOKHALE, S. S.; HORGAN, J. R.; TRIVEDI, K. S. **Locating program features using execution slices**. In: *Proceedings of the Second IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, p. 194–203, Richardson, TX, March 1999.

APÊNDICE A

Materiais do Experimento

A.1 Especificações dos Programas

A especificação de cada programa foi criada para auxiliar os participantes em entender os programas a serem testados durante o experimento. O programa B.M.I foi utilizado para o treinamento e os demais para a exeução dos testes pelos próprios participantes. Cada especificação pode ser analisada nas seções seguintes:

A.1.1 Especificação do Programa B.M.I

Calcular o B.M.I. ou I.M.C., em duas unidades de medida, quilogramas por metro e libra por polegada. Os dados de entrada são peso e altura. O valor do IMC deve ser mostrado ao usuário e classificado de acordo com os oito níveis conhecidos. O software deve ter a opção de mostrar informações/ajuda ao usuário. A fórmula para o cálculo do IMC é IMC = peso/altura2. Obs.: A classificação de IMC deve obedecer ao seguinte critério:

- Menor que 15 = Fome
- Maior que 15 e menor que 17.5 = Anorexia
- Maior que 17.5 e menor que 18.5 = Abaixo do peso
- Maior que 18.5 e menor que 25 = Normal (eutrôfico)
- Maior que 25 e menor que 30 = Sobrepeso
- Maior que 30 e menor que 35 = Obesidade Grau I
- Maior que 35 e menor que 40 = Obesidade Grau II
- Maior que 40 e menor que 60 = Obesidade Grau III (Obesidade Mórbida)

A.1.2 Especificação do Programa AntiPanela

O objetivo desse programa é não permitir a “formação de panelas” na escolha de equipes ou times. O software deve permitir o cadastro e a listagem de jogadores. Deve

também permitir o sorteio de duas equipes na qual a quantidade de jogadores por equipe deve ser informada. Ao final deverá ser emitida uma listagem com cada uma das equipes.

A.1.3 Especificação do Programa CarManger

É um software para um cliente americano, ou seja, sua interface deve ser escrita em inglês. O objetivo desse programa é acompanhar o consumo de combustível de um veículo automotor. Deve-se permitir cadastrar e listar informações sobre o consumo e abastecimento do veículo e ao ser solicitado deve-se emitir um relatório informando dados estatísticos sobre o consumo. Devem ser informados ao sistema, a data do fato, quantidade e preço do combustível, e quilomentragem do veículo. Informações adicionais e se o tanque foi completamente preenchido, também devem ser registradas. Um relatório deve ser emitido com informações estatísticas como, média de preço do combustível consumido no período, total de litros abastecidos no período, valor total pago no total do combustível, média de consumo, quantidade de abastecimentos e outros. O sistema deve permitir que as informações possam ser enviadas para um arquivo texto no dispositivo móvel.

A.1.4 Especificação do Programa CodiceFiscale

Codice Fiscale é um código que identifica uma pessoa na Itália, como o nosso CPF. O sistema deve calcular, validar e enviar via SMS o código a partir de um dispositivo móvel. Na Itália, o Código Fiscal é emitido no momento do nascimento de um(a) Italiano(a). Ele possui o seguinte formato “SSSNNNYYMDDZZZZX” e são:

- **SSS** são as três primeiras consoantes do sobrenome (family name) (a primeira vogal seguida de um X é utilizada se não existirem consoantes suficientes).
- **NNN** é o primeiro nome, a partir do qual a primeira, terceira e quarta consoantes são utilizadas (as exceções são tratadas de forma similar ao sobrenome).
- **YY** são os dois últimos dígitos do ano de nascimento. M é a letra para o mês de nascimento - as letras são usadas na seguinte ordem “ABCDEHLMPRST”, na qual A corresponde a janeiro, B a fevereiro e assim sucessivamente.
- **DD** é o dia do nascimento. Para diferenciar entre os sexos, soma-se 40 ao dia do nascimento das mulheres (assim sendo, uma mulher nascida em 03 de maio será 43)
- **ZZZZ** é o código de área do município onde a pessoa nasceu. Para pessoas nascidas em outros países, um código para o país é utilizado.
- **X** é um caractere de paridade calculado somando-se os caracteres nas posições pares e ímpares e computando o módulo do resultado por 26.

Tabela A.1: *Caracteres Alfabéticos Pares*

Caractere	Valor	Caractere	Valor	Caractere	Valor	Caractere	Valor
0	1	9	21	I	19	R	8
1	0	A	1	J	21	S	12
2	5	B	0	K	2	T	14
3	7	C	5	L	4	U	16
4	9	D	7	M	18	V	10
5	13	E	9	N	20	W	22
6	15	F	13	O	11	X	25
7	17	G	15	P	3	Y	24
8	19	H	17	Q	6	Z	23

Tabela A.2: *Caracteres Alfabéticos Ímpares*

Caractere	Valor	Caractere	Valor	Caractere	Valor	Caractere	Valor
0	0	9	9	I	8	R	17
1	1	A	0	J	9	S	18
2	2	B	1	K	10	T	19
3	3	C	2	L	11	U	20
4	4	D	3	M	12	V	21
5	5	E	4	N	13	W	22
6	6	F	5	O	14	X	23
7	7	G	6	P	15	Y	24
8	8	H	7	Q	16	Z	25

Tabela A.3: *Resto para o Dígito Verificador*

Resto	Letra	Resto	Letra	Resto	Letra	Resto	Letra
0	A	7	H	14	O	21	V
1	B	8	I	15	P	22	W
2	C	9	J	16	Q	23	X
3	D	10	K	17	R	24	Y
4	E	11	L	18	S	25	Z
5	F	12	M	19	T	-	-
6	G	13	N	20	U	-	-

Tabela A.4: *Exemplo de Codice Fiscale*

Nome:	**GILCIMAR**	Sobrenome:	**DIVINO DE DEUS**			
Sexo:	**M**	Data:	**03/04/1982**			
Província:	**ROMA**	Cidade:	**ROMA**		Código:	**H501**
SSS	NNN	YY	N	DD	ZZZZ	X
DVN	GCM	82	D	3	H501	
7+21+20	6+5+12	19+2	7	0+7	7+13+0+0	
	Soma Pares e Ímpares:		126	mod 26	22	**W**

CODIGO FISCAL ITALIANO: **DVNGCM82D03H501W**

A.2 Guia para Utilização do Experimento

Os guias a seguir demonstram em passos como os participantes devem proceder para executar os testes em cada uma das técnicas.

A.2.1 Instruções para aplicação da técnica ad hoc

1) Primeiramente, preencha os dados do Formulário 1 - Perfil do Participante;

TÉCNICA ad hoc

1) Leia detalhadamente a descrição do programa;
2) Se achar necessário repita o passo 1 até possuir o entendimento da especificação do programa;
3) Após ter a idéia do que o programa faz, execute-o e prepare-se para executar os casos de testes;
4) Avalie quais testes que você poderia fazer para encontrar erros no programa;
5) Anote cada caso de teste no Formulário 06 - Casos de Teste, execute-o e anote sua saída;
6) Caso tenha encontrado algo inesperado, registre no Formulário 03 - Defeitos Encontrados;
7) Repita os passos 4 a 6 até que você acredite não haver erros ou mais erros no programa;
8) Entregue os formulários preenchidos ao instrutor do curso;

A.2.2 Instruções para aplicação da técnica funcional

1) Primeiramente, preencha os dados do Formulário 1 - Perfil do Participante;

TÉCNICA FUNCIONAL

1) Leia detalhadamente a descrição do programa;
2) Se achar necessário repita o passo 1 até possuir o entendimento da especificação do programa;
3) Após ter a idéia do que o programa faz, execute-o e prepare-se para executar os casos de testes;
4) Avalie através da técnica funcional, quais testes que você poderia fazer para encontrar erros no programa;
5) Anote cada caso de teste no Formulário 06 - Casos de Teste, execute-o e anote sua saída;
6) Caso tenha encontrado algo inesperado, registre no Formulário 03 - Defeitos Encontrados;
7) Repita os passos 4 a 6 até que você acredite não haver erros ou mais erros no programa;
8) Entregue os formulários preenchidos ao instrutor do curso;

A.2.3 Instruções para aplicação da técnica estrutural

1) Primeiramente, preencha os dados do Formulário 1 - Perfil do Participante;

TÉCNICA ESTRUTURAL

1) Leia detalhadamente a descrição do programa;
2) Se achar necessário repita o passo 1 até possuir o entendimento da especificação do programa;
3) Abra a ferramenta JaBUTi/ME e crie um projeto para o seu experimento. Coloque o nome do projeto no seguinte padrão: [Nome do programa]_[Iniciais de seu nome];
4) Abra o programa a ser testado;
5) Analise a estrutura do programa apresentado pela JaBUTi/ME, avalie os pesos e visualize o GFC;
6) Através da técnica estrutural, avalie quais testes que você poderia fazer para realizar a cobertura do programa;
7) Anote cada caso de teste no Formulário 06 - Casos de Teste, execute-o, anote sua saída e percentual de cobertura alcançado;
8) Caso tenha encontrado algo inesperado, registre no Formulário 03 - Defeitos Encontrados;
9) Repita os passos 5 a 8 até que você acredite não haver erros ou mais erros no programa;
10) Entregue os formulários preenchidos ao instrutor do curso;

A.3 Formulários do Pacote de Experimentação

Os formulários criados para capturar o perfil dos participantes e dados da execução dos casos de testes são apresentados a seguir:

A.3.1 Formulário 1 - Perfil do Participante

Nome: _____

Idade: ____ Instituição de Ensino: _____ Curso: _____

Experiência em computação: Profissional ___ anos. Acadêmica ___ anos.

Experiência em programação: Java ___ anos. Outros ___ anos.

Experiência em testes ___ anos.

Gosta de testar programas?

( )S ( )N

Costuma testar BEM os programas que faz?

( )S ( )N ( )Às vezes

Costuma achar erros nos programas que faz?

( )S ( )N ( )Às vezes

Conhece a técnica funcional de teste de software?
( )S ( )N Se SIM já aplicou? ( )S ( )N
Conhece a técnica estrutural de teste e software?
( )S ( )N Se SIM já aplicou? ( )S ( )N
Já testou programas que você não fez?
( )S ( )N Se SIM encontrou muitos erros? ( )S ( )N
Já participou de algum treinamento sobre teste de software?
( )S ( )N Se SIM houve prática? ( )S ( )N

A.3.2 Formulário 2 - Definição dos Grupos

Grupos: G1 ( ) G2 ( ) G3 ( ) G4 ( ) G5 ( ) G6 ( )
Participante 01: _____
Participante 02: _____
Participante 03: _____
Participante 04: _____
Participante 05: _____

A.3.3 Formulário 3 - Defeitos Encontrados

Nome do Participante: _____
Nome do Programa: _____
Observações:
Caminho Percorrido/Dados de Entrada é uma descrição do que foi executado e quais dados foram fornecidos ao programa para produção de determinada saída.
Saída Produzida é a descrição do que ocorreu ou foi mostrado pelo programa após a execução de um caso de teste.
Saída Esperada, de acordo com a especificação do programa, é o que seria uma saída válida para o caso de teste que foi executado.
Erro(S/N) é um indicativo de erro, S para sim e N para não.
Observações é qualquer informação adicional que seja relevante para a execução ou que ocorreu durante o caso de teste.

A.3.4 Formulário 4 - Sugestões

Nome: _____
Sugestões para o próximo experimento:
_____
_____

Tabela A.5: *Caracteres Alfabéticos Pares*

N.º Caso	**Caminho Percorrido / Dados de Entrada**	**Saída Produzida**	**Saída Esperada**	**Erro (S/N)**	**Observações**
01					

_____
_____
_____
_____
_____
_____

A.3.5 Formulário 5 - Avaliação do Experimento

Nome: _____

Possuía domínio do assunto apresentado antes do curso?

( )S ( )N

O aprendizado do curso agregou conhecimento?

( )S ( )N

Você sente seguro para aplicar esse conhecimento em outros programas?

( )S ( )N

Você pretende aplicar o conhecimento adquirido em sua vida profissional ou acadêmica?

( )S ( )N

Você pretende seguir carreira na área de teste de software?

( )S ( )N

Nota para o seu aprendizado no curso (0-10): ____
Nota para o curso (0-10): ____

A.4 Defeitos Inseridos nos Programas

Os defeitos inseridos em cada um dos programas podem ser conferidos nas tabelas abaixo, eles estão classificados de acordo com as categorias definidas na Tabela . Nas tabelas abaixo cada defeito está descrito e explicado o comportamento do software quando o mesmo for executado.

A.4.1 Programa CarManager

Nessa seção são apresentados os defeitos que foram acrescentados no programa CarManager com o objetivo dos participantes localizá-los com a aplicação de cada uma das técnicas discutidas durante os treinamentos.

A Tabela **??** apresenta os defeitos acrescentados, em suas respectivas linhas de código suas alterações e a descrição do comportamento do software durante a execução. As classificação do erro também pode ser encontrada juntamente com o nome do arquivo alterado.

Tabela A.6: *Defeitos Inseridos no Programa CarManager*

Nr.	Linha	Original	Alteração	Classe	Tipo	Arquivo	Descrição
1	65	new Command(“About”, Command.SCREEN, 7);	new Command(“Abouty”, Command.SCREEN, 7);	C	Cosmetic	CarManagerMIDlet	Erro de digitação.
2	423	exporter.export(fillupEntries, path);	*Apagado*	O	Interface	CarManagerMIDlet	Não realize a exportação dos dados.
3	45 a 49	append(new StringItem(“Car Manager v1.4”, “Copyright (C) 2005-2006 Tommi Laukkanen” + “http://www.substance-ofcode.com”)); append(new StringItem(“License”, “GPL (See “ + “http://www.gnu.org/copyleft/gpl.html for more information)”));	*Apagado*	O	Cosmetic	AboutForm	Não aparece a mensagem da tela About.

Tabela A.6: *Defeitos Inseridos no Programa CarManager*

4	83	int fillupCount = 0;	int fillupCount = -1;	C	Initialization	ReportForm	Qtde de abastecidas é sempre 1 und. menor que a real. Media de abastecimento e preço por litro ficam errados.
5	88	if(entry.getFullFillUp() !=0)	if(!(entry.getFullFillUp() !=0))	C	Control, Interface		Se o registro estiver gravado como encheu o tanque, ao editar o item “Full fill-up” não fica marcado, e vice-versa.
6	171	if(selection>0)	*Apagado*	O	Control, Interface, Data	CarManagerMIDlet	Ao apagar o último item da lista, nenhum erro ocorre. Mas em sequência cadastrar outro acontece erro, há uma tentativa de selecionar uma posição não existente na lista -1.
7	75	settings.setEntryType(entryType);	settings.setEntryType(0);	C	Computation	SettingsForm	A configuração de ambiente apesar de marcado o item correto os cálculos sempre levam em consideração a opção “Total fill-up”.
8		Erro já existente na versão original		O	Computation, Data		Ao tentar editar, remover um item com lista vazia acontece erro de arrayIndexOut.

Tabela A.6: *Defeitos Inseridos no Programa CarManager*

9		Erro já existente na versão original		O	Computation, Data		Ao remover todos itens quando a lista é vazia, é lançada uma exceção file not found.

A.4.2 Programa CodiceFiscale

Nessa seção são apresentados os defeitos que foram acrescentados no programa CodiceFiscale com o objetivo dos participantes localizá-los com a aplicação de cada uma das técnicas discutidas durante os treinamentos.

A Tabela A.7 apresenta os defeitos acrescentados, em suas respectivas linhas de código suas alterações e a descrição do comportamento do software durante a execução. As classificação do erro também pode ser encontrada juntamente com o nome do arquivo alterado.

Tabela A.7: *Defeitos Inseridos no Programa CarManager*

Nr.	Linha	Original	Alteração	Classe	Tipo	Arquivo	Descrição
1	191	If(s.lenght()==0)	If(s.lenght()==1)	C	Control	CodiceFiscaleMid-let	Exibe a mensagem informando que o código está vazio, porém o mesmo possui tamanho igual a um.
2	15	“Fiscal”	“Discal”	C	Cosmetic	.properties	No título do formulário principal para o cálculo é exibida a palavra “Discal” no lugar de “Fiscal”.
3	199	alert.setString(I18N.get-Instance().translate(“ alert.verify.too_long”));	alert.setString(I18N.get-Instance().translate(“ alert.verify.too_long”) + “XXX”);	C	Cosmetic	CodiceFiscaleMid-let	Exibe a mensagem de alerta com lixo em seu final, “XXX”.
4	213	else	*Apagado*	O	Interface, Control	CodiceFiscaleMid-let	Quando o CF é válido está sendo exibida a mensagem de inválido.
5	254	break;	*Apagado*	C	Control, In-terface	CodiceFiscaleMid-let	É exibido o item de informações quando é escolhido o item ajuda no menu principal.

Tabela A.7: *Defeitos Inseridos no Programa CarManager*

6	456	startPos = 0;	startPos = chars.length;	C	Initialization	CodiceFiscaleMidlet	Quando o numero do para envio de SMS inicia com mais de um sinal de mais “+”, o sistema informa que conseguiu enviar SMS. Não está fazendo validação corretamente.
7	389	FEMMINA= “F”	FEMMINA= “X”	C	Initialization	CodiceFiscale	O valor da combo sexo no formulário para calculo está com valor “X” no sexo Feminino.
8	249	nomeLunCons = nomeCons.length();	nomeLunCons = nomeCons.length()+1;	C	Computation, Data	CodiceFiscale	Quando a qtde de consoantes do nome é menor ou igual a 3 aparece erro ao capturar parte desse nome. Ex.: “DIVINO”

9	16
	
	167
		cognome = cognome.toUpperCase();
		
		s = s.toUpperCase();
		this.codiceFiscale = newCodiceFiscale.toUpperCase();
			Apagado
			
			Apagado
				O	Interface, Computation e Initialization	CodiceFiscale	Deveria converter as 3 primeiras letras do CF para maiúsculas e isso não acontece.

Tabela A.7: *Defeitos Inseridos no Programa CarManager*

	34		this.codiceFiscale = newCodiceFiscale;				
10				C	Interface, Control		Ajuda e propriedades com mesmo conteudo

A.4.3 Programa AntiPanela

Nessa seção são apresentados os defeitos que foram acrescentados no programa CodiceFiscale com o objetivo dos participantes localizá-los com a aplicação de cada uma das técnicas discutidas durante os treinamentos.

A Tabela A.8 apresenta os defeitos acrescentados, em suas respectivas linhas de código suas alterações e a descrição do comportamento do software durante a execução. As classificação do erro também pode ser encontrada juntamente com o nome do arquivo alterado.

Tabela A.8: *Defeitos Inseridos no Programa AntiPanela*

Nr.	Linha	Original	Alteração	Classe	Tipo	Arquivo	Descrição
1	35	“Jogador”	“Jorgador”	C	Cosmetic	AntiPanelaMidlet	Erro de digitação.
2	91	if (this.nomeJogadorField. getString().equals(“”))	if (this.nomeJogadorField. getString().length() <=1)	C	Control	CadastrarJogador-Form	É emitida a mensagem quando o nome é igual a vazio ou possui um caracter.

3	63
	
	64
	
	191
		String[] nomes-Jogadores = new String[this.jogadores. length];
		for (int i = 0; i < this.jogadores.length; i++)
		for (int i = 0; i < joga-dores.length; i++)
			String[] nomes-Jogadores = new String[this.jogadores. length-1];
			
			for (int i = 0; i < this.jogadores.length-1; i++)
			
			for (int i = 0; i < jogadores.length-1; i++)
				C	Control, Initiali-zation, Interface	EscolherJogadores-List	Mostra um jogador a menos na lista para selecionar os que en-trarão no sorteio.
							
							Para não dar erro de índice de vetor e fazer o sorteio entre os n-1 elementos.

Tabela A.8: *Defeitos Inseridos no Programa AntiPanela*

4	68	“Sortear Time”	“SortearTime”	C	Cosmetic	EscolherJogadores-List	Erro de digitação.
5	123	this.flagsSelecionadas[i] = false;	*Apagado*	O	Computation, Interface	EscolherJogadores-List	Se todos os itens estiverem selecionados não desmarca todos.
6	130	this.flagsSelecionadas[i] = true;	this.flagsSelecionadas[i] = false;	C	Computation, Interface	EscolherJogadores-List	Se nenhum item estiver selecionado, ao escolher a opção selecionar todos, o sistema não seleciona nenhum item.

7	136	for (int i = 0; i < this.flagsSelecionadas. length; i++)
		
		this.flagsSelecionadas[i] = true;
			for (int i = 1; i < this.flagsSelecionadas. length; i++)	C	Initialization, Control, Data	EscolherJogadores-List	Se o último nome da lista de jogadores não estiver selecionado, ao escolher selecionar todos (com alguns já marcados) o sistema não seleciona o último nome.

A.5 Dados sem a Redução

Nesta seção, na Tabela A.9 são apresentados os dados coletados de todos os participantes do experimento na íntegra, sem reduções.

Tabela A.9: *Dados do Experimento Sem Redução*

		ANTIPANELA								CARMANAGER								CODICEFISCALE							
Grupo	**Participante**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**
G1	Plínio de Sá Leitão Júnior	66	56	52	48	10	10	4	2	70	56	61	50	13	10	3	0	73	58	64	54	22	10	2	0
	Georges Felipe e Silva	86	78	79	78	9	8	6	1	48	35	40	34	6	6	3	1								
	Paulo Júnior do Nascimento Lima	61	46	48	55	4	4	1	2	68	54	57	54	5	5	3	1	64	48	52	41	2	2	1	0
	Ricardo Gobbo									51	38	39	35	9	6	3	0								
	Valdemar Vicente Graciano Neto	89	77	80	79	20	14	4	3									44	32	40	31	12	10	2	0
G4	Everton Lima Aleixo	78	72	74	74	12	12	8	3	67	56	60	48	6	6	5	1	71	56	63	49	28	4	2	0
	Cássio Felipe F. de Oliveira	83	67	67	71	35	12	7	3	32	22	27	21	10	5	0	0								
	Diego Roriz	56	46	43	44	11	8	5	2	45	34	38	31	16	12	3	1	78	64	69	0	9	6	1	0
	Diego Guedes	92	81	83	82	13	6	6	1	28	18	22	16	6	5	0	0								
G2	Marcos Paulino Roriz Júnior	94	87	88	86	12	12	7	2	51	37	42	34	6	4	3	2	47	35	43	34	7	6	2	3
	Bruno Gonçalves	45	36	33	32	6	6	5	1	73	60	64	55	8	5	2	3	52	40	44	35	4	4	3	0
	Luiz Fernando Calaça Silva									51	38	41	37	4	3	1	1								
	Vinícius Pessoni									67	55	58	53	8	7	3	1	65	50	56	42	8	6	2	0
G5	Érico Mortari									66	50	52	44	4	4	2	1	18	14	13	12	5	5	2	1
	Bruno Ferreira Machado	88	74	74	75	18	6	5	2	62	50	51	41					59	44	51	37	7	4	1	0

Tabela A.9: *Dados do Experimento Sem Redução*

		ANTIPANELA								CARMANAGER								CODICEFISCALE							
Grupo	**Participante**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**
	Diocleciano N. Neto	94	87	88	86	16	4	1	3	43	35	40	33	14	3	0	0	57	45	50	44	20	12	2	0
G3	Flávio Vicente A. Filho	91	77	77	80	4	1	2	0	73	59	63	57	5	3	0	1	18	13	11	12	11	2	2	0
	James Lima Cipriano Mota	75	66	67	68	3	2	2	0	62	51	52	53					54	42	49	37	5	5	2	0
	Gisele Machado de Souza	91	78	78	79	14	3	5	0	66	51	56	53	15	8	1	0	46	35	40	36	6	5	1	3
	Arthur Henrique Guimarães de Oliveira	71	62	64	65	19	15	3	4									46	34	40	30	18	7	1	0
G6	Bruno Miranda Zafalão																	73	58	64	48	6	6	3	0
	André Mesquita Rincon	63	53	50	50	9	2	4	3	65	51	53	42	11	5	2	0	55	43	48	40	9	6	2	0
	Thiago Pereira da Silva	79	70	73	74	8	8	4	2	48	35	42	32	11	11	2	1	61	49	54	42	27	12	2	1
	Aleixo Alves de Sousa Júnior	44	34	30	31	11	8	4	0									44	33	38	30	7	5	2	0
G1	Taciano Messias Moraes	71	61	57	55	13	8			70	55	58	47	12	3	2	0	76	62	68	56	11	4	3	0
	Cairo S. Araújo	55	45	42	40					45	32	35	33	20	1	0	0	18	13	11	12	6	6	3	0
	Paulo Henrique Jayme Alves	69	59	54	50	13	9	5	1	71	58	62	52	10	5	0	0	63	50	58	44	10	7	4	0
G4	Adriano Cunha	45	37	34	34	12	11	5	3	42	29	33	28	15	10	1	0	73	58	66	52	13	8	3	1
	Breno Pimenta da Costa									38	31	33	28	12	7	0	0	76	62	68	55	24	4	0	0
	Paulo Henrique Rocha Faleiro									60	44	49	37	9	3	2	1	76	61	67	54	10	4	4	0

Tabela A.9: *Dados do Experimento Sem Redução*

		ANTIPANELA								CARMANAGER								CODICEFISCALE							
Grupo	**Participante**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**
	Tatiana Lisita Rilvia									60	44	49	37	9	3	2	1	76	61	67	54	10	4	4	0
G2	Romeu Gomes de Morais Filho	92	81	81	81	12	4	3	0	68	54	59	46	4	3	1	2	66	50	55	45	4	2	1	0
	Rodrigo Sousa Martins	83	70	69	72	19	6	4	1	57	43	49	44					25	19	20	15	1	1	0	1
	Marcelo Gomes Fidelis	96	89	89	86	8	0	0	0	72	59	63	53	16	7	2	4	74	58	64	51	12	8	4	1
G5	Christian Andersen Rezende	95	87	88	85	20	9	5	1	62	46	51	39	12	9	5	1	50	38	45	36	8	3	1	0
	Thiago Eliandro Silva de Castro									66	51	56	42	5	2	0	0								
	Eveton C. de Araujo Assis	73	61	56	55	13	8	3	3	60	48	56	51	12	11	2	5	68	54	58	52	12	7	1	0
G3	Adalberto Ribeiro Sampaio Junior	72	62	65	66	15	10	2	3									18	13	11	12	16	5	4	0
	Max Flavio Cabral	61	50	51	57	7	4	3	0	72	58	63	49	8	3	2	0	41	30	37	26	3	3	0	0
	Claudio Antonio de Araujo	72	62	65	66	6	2	2	3	67	52	57	44	11	0	0	0	18	12	11	12	1	0	0	0
	Sirlene	61	51	46	44	6	1	1	0	70	57	62	55	9	2	0	2								
G6	Marcelo Hiroaki Ito	78	71	73	73	12	10	3	4	72	59	62	56	15	6	3	2	59	49	54	48	10	4	3	1
	Marcos Antonio de Paula	34	23	19	20	7	2	0	2	63	46	49	41												
	Elida Rocha de Borba	22	14	12	10	15	9	3	2	54	40	42	36	9	2	1	1								

Tabela A.9: *Dados do Experimento Sem Redução*

		ANTIPANELA								CARMANAGER								CODICEFISCALE							
Grupo	**Participante**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**	**T-Nos**	**T-Arestas**	**T-Usos**	**T-PUsos**	**Casos Teste**	**Erros**	**Erros Reais**	**Erros Novos**
G1	Márcio Rodrigues Oliveira	21	14	11	9	1	0	0	0	71	58	63	50	1	0	0	0	19	14	13	12	10	6	2	1
	Bruno Pereira Maia	81	72	72	75	14	9	3	1																
	Ricardo Ramos Garcia Ayala	88	75	74	75	5	1	1	0	74	61	65	54	11	10	7	1	78	65	71	60	10	6	5	1
G4	Bruno Miranda Zafalão	78	68	70	71	4	3	0	0	69	54	56	48	1	0	0	0	74	61	66	55	0	0	0	0
	Márcio Roberto Monteiro de Sousa	50	38	37	39	2	1	1	0	67	54	58	44	3	2	0	0	76	62	69	56	8	4	4	0
G2	Luiz Henrique Machado									18	11	13	9	4	1	0	0								
	Leandro Ferreira Almeida									45	36	43	34	8	3	2	1								
	Leonardo Lacerda Feliz de Souza	92	80	80	80	10	2	2	2	54	41	46	36	9	4	1	1	69	53	57	46	6	4	2	0
G5	André Ribeiro de Miranda	98	89	89	86	14	9	3	1	52	36	39	32	5	4	2	0	18	13	11	12	6	2	1	0
	Rogério Bendo Fiuza	97	91	91	87	12	2	2	0	51	40	45	40	5	3	2	1	42	30	35	28	2	0		
G3	Marilene Vieira Ferreira	41	35	34	37	12	5	3	1	68	54	58	49	8	1	0	0	18	13	11	12	6	0	0	0
	Eduardo Rodrigues Costa	82	68	71	71	18	9	3	2	69	57	61	48	11	4	2	0	47	35	41	32	5	3	2	0
	Amivaldo Batista dos Santos	43	34	28	25	16	4	2	0	75	64	68	61	13	1	0	0	29	22	23	19	1	1	1	0
G6	Túlio Nogueira Galli																	50	38	43	36	3	3	2	0
	Leandro Leal Parente	63	53	49	44	8	5	2	2	70	54	57	48	12	1	0	0	18	13	11	12	1	1	0	1

Media ad hoc	69	58	57	58	11	7,3	3,7	1,5	57	44	48	40	7,8	4,6	1,8	1,4	41	31	34	28	7,9	4,0	1,6	0,4
Media Funcional	64	54	53	53	11	5,6	2,7	1,6	57	44	48	39	9,2	5,2	1,8	0,4	51	39	43	35	7,3	4,6	1,7	0,5
Media Estrutural	87	78	77	76	13	5,7	3,3	1,3	66	53	56	48	11	3,6	1,0	0,5	65	52	57	43	12	5,3	2,5	0,2
DP ad hoc	19	18	20	20	8,2	4,3	2,6	1,2	13	12	12	11	3,9	2,7	1,3	1,5	18	15	18	13	6,9	3,0	1,2	0,8
DP Funcional	20	19	21	22	4,8	4,0	1,2	1,5	15	14	14	11	5,0	3,5	2,0	0,5	19	15	17	14	4,8	3,1	1,0	0,9
DP Estrutural	15	16	17	17	4,3	3,5	2,0	1,1	7,3	7,6	7,6	8,1	2,8	3,2	1,1	0,8	20	17	19	19	7,4	2,7	1,5	0,4
MIN ad hoc	21	14	11	9	1	0,0	0,0	0,0	18	11	13	9	4	1	0	0	18	12	11	12	1	0	0	0
MIN Funcional	22	14	12	10	3	1,0	0,0	0,0	28	18	22	16	1	0	0	0	18	13	11	12	1	0	0	0
MIN Estrutural	45	36	33	32	6	0	0	0	48	35	42	32	5	0	0	0	18	13	11	0	0	0	0	0
MAX ad hoc	92	81	83	82	35	14	8	3	73	60	64	55	16	11	5	5	73	58	64	48	27	12	4	3
MAX Funcional	91	78	78	80	19	15	5	4	74	61	65	54	20	12	7	1	74	58	64	52	20	12	4	3
MAX Estrutural	98	91	91	87	20	12	7	3	75	64	68	61	15	11	3	2	78	65	71	60	28	10	5	1