

---

Um estudo de caracterização e avaliação de  
critérios de teste estruturais entre os paradigmas  
procedimental e OO

---

*Marllos Paiva Prado*

---



SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 18 de março de 2009

Assinatura: \_\_\_\_\_

# Um estudo de caracterização e avaliação de critérios de teste estruturais entre os paradigmas procedural e OO

*Marllos Paiva Prado*

Orientador: *Prof. Dra. Simone do Rocio Senger de Souza*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC/USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional.

**USP - São Carlos  
Maio/2009**



*À minha mãe, Márcia, e avô, Afrânio  
“in memorian”.*



# Agradecimentos

---

Um cientista se priva, pelo próprio ofício, da emotividade em seu trabalho durante toda sua jornada de pesquisa. Por isso me atrevo, nesses breves agradecimentos, a fugir um pouco do padrão e fazer antes uma homenagem a alguém muito especial.

Trata-se de uma pessoa que está sempre comigo, mesmo estando longe. Alguém que sempre tem a vez (e a voz) quando bate aquela dúvida nos pensamentos. Alguém que me encorajou em todas as batalhas que enfrentei aqui. Que com “mineiras” palavras, me ensinou sobre a vida coisas que só se aprende vivendo uma vida inteira. Se hoje sou grande, mesmo sendo tão “pequeno”, devo isso ao senhor meu companheiro e amigo “vô”. Sei que nunca poderá ler essas linhas como eu gostaria que as lesse. Talvez porque tenha as escrito um pouco tarde para a pressa que a vida lhe impôs. Mas sei que eterna é minha gratidão com o senhor, só por sentir a sua presença e a sua alegria em minha mente, em cada palavra que agora escrevo. Os seus ensinamentos ecoarão para sempre através do meu caráter e dos méritos que em vida eu puder alcançar. Um “obrigado!” digno de “mestre”, como o senhor próprio se antecipou em me chamar!

À minha mãe e à minha irmã Lorena, dedico meu esforço, minha gratidão e meu amor, pela força com que sustentaram os turbulentos caminhos seguidos até aqui! Ao restante da minha família agradeço o apoio e a força, mesmo com o pouco contato que tivemos nesses dois anos.

Agradeço à minha namorada, Natália, pelo amor, carinho, paciência e companheirismo nesses três anos de namoro. A distância fortaleceu e amadureceu ainda mais o afeto que temos um pelo outro!

Agradeço a todos do LaBES e “agregados”, sem exceção, pelas grandes amizades proporcionadas nesses dois anos de convivência. Vocês se tornaram uma extensão de minha família e desnecessário seria citar o nome de cada um já que certamente serão aqueles que lerão esses agradecimentos!

Agradeço à minha orientadora, Dra. Simone e ao meu co-orientador Dr. José Carlos Maldonado, pela orientação, intervenção e conselhos na condução deste trabalho.

Obrigado ao CNPq pelo apoio financeiro.

Um agradecimento especial ao meu ex-orientador de graduação, Auri, por ter me apresentado a área de Testes, pela amizade e pelo suporte dado nos momentos mais críticos dessa jornada.

Encerro agradecendo a Deus, por ter colocado em minha vida todos os ingredientes que precisava para alcançar mais essa conquista!



# **Resumo**

---

---

O Teste de software é uma atividade de garantia da qualidade que tem por finalidade diminuir o número de defeitos do software. Esta atividade contribui para redução do custo de manutenção e para a melhora da qualidade do software, durante o processo de desenvolvimento. Isso tem motivado a investigação e proposta de estratégias, técnicas, critérios e ferramentas de teste para diferentes paradigmas de desenvolvimento, tais como procedural, orientado a objetos e orientado a aspectos.

Vários estudos experimentais têm sido desenvolvidos para avaliar e comparar critérios de teste. Grande parte desses experimentos foram realizados com programas construídos sob um mesmo paradigma ou desconsiderando a influência do mesmo sobre os resultados. Entretanto, é importante avaliar o impacto de um paradigma específico sobre a atividade de teste uma vez que alguns defeitos podem estar relacionados ao seu uso.

Este trabalho apresenta um estudo experimental realizado para caracterizar e avaliar o custo de aplicação e a dificuldade de satisfação de critérios de teste, comparando dois paradigmas: o orientado a objetos e o procedural. O estudo considera critérios de teste funcionais e estruturais e utiliza um conjunto de programas do domínio de Estrutura de Dados. Os termos e fases do processo de experimentação controlada foram usados, à medida em que estes se mostraram adequados, para definir e executar o presente estudo. Os objetivos com a execução dessa pesquisa foram obter resultados iniciais sobre as questões investigadas bem como gerar artefatos que sirvam de base para a definição e condução de futuros experimentos e a criação de pacotes de laboratório. Além disso, pretende-se apoiar, por meio dos materiais gerados, o treinamento e o ensino da atividade do teste de software.



# **Abstract**

---

---

Software Testing is a quality assurance activity that aims at reducing the number of software faults. This activity contributes for the reduction of maintenance costs and for software quality improvement during the development process. These factors have motivated the investigation and proposal of several testing strategies, techniques, criteria and tools for different programming paradigms, such as procedural, object-oriented and aspect-oriented.

Regarding testing criteria, many experimental studies have been performed to evaluate and compare them. In general, these experiments comprise programs developed under the same paradigm or this influence over the results. However, some faults can be paradigm-related and it is important to evaluate its impact on the testing activity.

This work presents an experimental study developed to characterize and evaluate the application cost and strength of testing criteria, comparing two programming paradigms: object-oriented and procedural. This study considers functional and structural testing criteria and uses a set of programs from the data structure domain. Terms and phases from controlled experimentation process were used, as long as they showed to be adequate, to define and execute the present study. The research aims at obtaining initial results about the questions investigated as well as generating artifacts which support the definition and conduction of future experiments and the creation of laboratory packages. In addition, it intends to support, through the materials generated, the training and teaching of software testing activity.



# Sumário

---

---

<b>Resumo</b>	<b>v</b>
<b>Abstract</b>	<b>vii</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Contexto . . . . .	1
1.2 Motivação . . . . .	4
1.3 Objetivos . . . . .	5
1.4 Organização . . . . .	5
<b>2 Teste de Software</b>	<b>7</b>
2.1 Considerações Iniciais . . . . .	7
2.2 Fundamentos do Teste de Software . . . . .	7
2.3 Teste Funcional . . . . .	10
2.3.1 Particionamento de Equivalência . . . . .	10
2.3.2 Análise do Valor Limite . . . . .	11
2.4 Teste Estrutural . . . . .	11
2.4.1 Critérios Baseados na Complexidade . . . . .	12
2.4.2 Critérios Baseados em Fluxo de Controle . . . . .	12
2.4.3 Critérios Baseados em Fluxo de Dados . . . . .	13
2.4.4 Relação de Inclusão Entre Critérios Estruturais . . . . .	14
2.5 Teste Baseado em Erros . . . . .	15
2.5.1 Critério Análise de Mutantes . . . . .	15
2.6 Teste Baseado em Modelos . . . . .	18
2.7 Critérios de Teste Específicos ao Paradigma Orientado a Objetos . . . . .	18
2.7.1 Teste Estrutural para OO . . . . .	20
2.7.2 Teste de Mutação para OO . . . . .	22
2.8 Ferramentas de Teste de Software . . . . .	23
2.9 Considerações Finais . . . . .	26
<b>3 Engenharia de Software Experimental</b>	<b>27</b>
3.1 Considerações Iniciais . . . . .	27
3.2 Engenharia de Software Experimental . . . . .	28
3.2.1 Processo de experimentação . . . . .	30
3.3 Trabalhos Relacionados . . . . .	36

3.3.1	Comparação de Estudos Experimentais Correlacionados . . . . .	43
3.3.2	Aspectos Relacionados à Validade Externa . . . . .	45
3.4	Considerações Finais . . . . .	45
<b>4</b>	<b>Definição e Planejamento do Estudo</b>	<b>47</b>
4.1	Considerações Iniciais . . . . .	47
4.2	Definição do Estudo . . . . .	47
4.2.1	Definição de Metas . . . . .	47
4.3	Planejamento . . . . .	48
4.3.1	Seleção de Contexto . . . . .	49
4.3.2	Formulação das Hipóteses . . . . .	49
4.3.3	Seleção de Variáveis . . . . .	50
4.3.4	Design do Experimento . . . . .	52
4.3.5	Instrumentação . . . . .	55
4.3.6	Avaliação de Validade . . . . .	72
4.4	Considerações Finais . . . . .	75
<b>5</b>	<b>Condução do Estudo e Análise dos Resultados</b>	<b>77</b>
5.1	Considerações Iniciais . . . . .	77
5.2	Operação . . . . .	77
5.2.1	Preparação . . . . .	77
5.2.2	Execução . . . . .	81
5.2.3	Validação dos Dados . . . . .	83
5.3	Análise e Interpretação dos Resultados . . . . .	84
5.3.1	Descrição Estatística . . . . .	84
5.3.2	Teste de Hipóteses . . . . .	96
5.4	Considerações Finais . . . . .	108
<b>6</b>	<b>Conclusões e Trabalhos Futuros</b>	<b>111</b>
6.1	Contribuições . . . . .	113
6.2	Dificuldades e Limitações . . . . .	114
6.3	Trabalhos futuros . . . . .	115
<b>A</b>	<b>Apendice A</b>	<b>127</b>

# **Lista de Figuras**

---

---

2.1	Ordem parcial entre os critérios Potenciais-Usos básicos, Critérios de fluxo de dados e de fluxo de controle (Maldonado, 1991). . . . .	15
2.2	Hierarquia entre os critérios de testes estruturais intramétodos. Fonte: (Vincenzi, 2004). . . . .	22
3.1	Visão geral da fase de planejamento. Fonte: (Wohlin et al., 2000) . . . . .	32
3.2	Empacotamento ao longo do processo de experimentação (Amaral, 2003). . . . .	35
4.1	<i>Design</i> Aninhado em dois estágios: No primeiro o fator de interesse <b>paradigma</b> , com dois tratamentos: OO e procedural; No segundo o fator bloqueante <b>critério</b> com três níveis: Funcional, Estrutural Fluxo de Controle e Estrutural Fluxo de Dados. . . . .	54
4.2	Diagrama das quatro etapas da estratégia de teste definida. . . . .	55
4.3	Template do documento de especificação. Primeira página. . . . .	57
4.4	Template do documento de especificação. Segunda página. . . . .	58
4.5	Template do documento de especificação. Terceira página. . . . .	59
4.6	Template do documento de especificação. Quarta página. . . . .	60
4.7	Template do documento de implementação em C. Primeira página. . . . .	62
4.8	Template do documento de implementação em C. Segunda Página . . . . .	63
4.9	Template do documento de implementação em C. Terceira Página . . . . .	64
4.10	Template do formulário de classes de equivalência. . . . .	66
4.11	Template do Formulário de testes OO. Primeira Página . . . . .	68
4.12	Template do formulário de testes OO. Segunda Página . . . . .	69
4.13	Template do formulário de testes OO. Terceira Página . . . . .	70
4.14	Template do formulário de testes OO. Quarta Página . . . . .	71
5.1	Máquina virtual definida para o estudo, em execução. . . . .	79
5.2	Estrutura de diretórios e arquivos para cada programa do estudo. . . . .	81
5.3	Interface de aplicativo para geração de seqüências aleatórias. . . . .	82
5.4	Trecho de código contendo caso de teste não implementável na linguagem do paradigma oposto . . . . .	96
5.5	Resultados do teste de normalidade sobre a variável $D_a$ e gráfico <i>q-q plot</i> correspondente . . . . .	99
5.6	Resultados do teste de normalidade sobre a variável $D_b$ e gráfico <i>q-q plot</i> correspondente . . . . .	100
5.7	Resultados do teste de Wilcoxon ( <i>Signed-Rank</i> ) para $H_0a$ . . . . .	102

5.8	Resultados do teste de Wilcoxon ( <i>Signed-Rank</i> ) para $H_0b$	102
5.9	Resultados do teste de normalidade sobre a variável $D_a$ e gráfico <i>q-q plot</i> correspondente	104
5.10	Resultados do teste de normalidade sobre a variável $D_b$ e gráfico <i>q-q plot</i> correspondente	105
5.11	Resultados do teste de Wilcoxon ( <i>Signed-Rank</i> ) para $H_0a$	106
5.12	Resultados do teste de Wilcoxon ( <i>Signed-Rank</i> ) para $H_0b$	107

# **Lista de Tabelas**

---

---

5.1	Tabela de programas usados no estudo com identificador, nome e descrição de funcionalidade. . . . .	82
5.2	Métricas de implementação para programas procedimentais. . . . .	86
5.3	Métricas de implementação para programas orientados a objetos. . . . .	87
5.4	Medidas de teste para critérios funcionais. . . . .	88
5.5	Métricas de teste estrutural fluxo de controle. . . . .	91
5.6	Métricas de teste estrutural fluxo de dados. . . . .	93
5.7	<i>Strength</i> dos critérios entre paradigmas. Nas colunas da esquerda, a cobertura refere-se ao conjunto OO-adequado sobre os programas procedimentais. Na coluna da direita, a cobertura refere-se ao conjunto procedural-adequado sobre os programas OO. . . . .	97



# **Lista de Siglas**

---

---

AM	-	Análise de Mutantes
API	-	Application Programming Interface
GFC	-	Grafo de Fluxo de Controle
IEEE	-	Institute of Electrical and Electronics Engineers
OO	-	Orientação a Objetos
SOA	-	Service Oriented Architecture
VV&T	-	Verificação, Validação e Teste



# Introdução

## 1.1 Contexto

A construção de software é uma atividade muitas vezes complexa, onerosa e sujeita a diversos tipos de problemas que podem levar à obtenção de um produto final que não corresponda às expectativas do usuário. Muitos desses problemas relacionam-se à existência de defeitos no software, os quais podem acarretar em um comportamento anormal da solução desenvolvida. Tais defeitos por sua vez, podem apresentar diversas causas, sendo as mesmas freqüentemente associadas a algum tipo de engano cometido por aqueles que conceberam ou desenvolveram o produto de software em questão.

Um dos alicerces para o desenvolvimento de qualquer sistema de software está fundamentado na forma como os problemas do mundo real são entendidos e transcritos para uma linguagem computacional, permitindo que soluções para o mesmo sejam definidas e atendam às necessidades levantadas. O paradigma de desenvolvimento é uma das abordagens para tratar essa questão. Sabe-se que cada paradigma define a construção das estruturas que compõe o software de diferentes formas. O uso de um determinado paradigma implica em diferenças desde a forma de se abstrair e representar essas estruturas (Takashi et al., 1990) até o processo e tecnologias empregados (Pressman, 2002).

Dois importantes paradigmas utilizados no desenvolvimento de sistemas de software são o procedural e o orientado a objetos. O paradigma procedural foi o primeiro paradigma para o desenvolvimento de software e acompanhou a evolução natural das linguagens de programação. Neste, as estruturas de um programa são definidas à semelhança de funções matemáticas, nos quais parâmetros de entrada são fornecidos, as funções desejadas são computadas com base nos

parâmetros fornecidos e ao final um valor de resposta pode ou não ser retornado. A estas estruturas denomina-se funções ou procedimentos, sendo que um programa desenvolvido neste paradigma é composto por um ou mais blocos de procedimentos. Uma das característica desse paradigma é a alta taxa de dependência entre os procedimentos e dados, principalmente na definição de grandes sistemas, o que pode elevar os custos de manutenção uma vez que pequenas alterações nos dados podem implicar em profundas alterações em outras partes do software.

O paradigma orientado a objetos por sua vez, caracteriza-se pelo entendimento do software como um conjunto de entidades com características próprias e bem definidas, denominadas objetos, as quais podem interagir entre si para realização de uma operação. Um objeto é definido por meio de uma classe, a qual é composta de atributos (dados) e métodos (procedimentos/funções). Algumas das características da orientação a objetos são: encapsulamento dos dados na classe, ocultamento de informação, herança, polimorfismo e acoplamento dinâmico.

A existência de particularidades e a diferença no modo de entender os problemas imposta pelos paradigmas, fornecem indícios da possível existência de diferentes fontes de erros na composição do software, em razão dos mesmos (Vincenzi et al., 2007).

Uma forma de se minimizar a presença de erros em um programa seria empregando o teste exaustivo, no qual todos os valores do domínio de entrada são testados. Esta, contudo, é uma prática reconhecida, em geral, como impraticável. Sendo assim, técnicas e critérios de teste têm sido propostos, os quais definem um meio para selecionar casos de teste com grande probabilidade de identificar os erros existentes. As principais técnicas de teste para programas são as técnicas funcional, estrutural e baseada em erros. Cada uma dessas técnicas envolve um conjunto de critérios sendo que a principal característica que as distingue consiste na fonte de informação utilizada para definir os requisitos de teste. Na técnica funcional, por exemplo, utiliza-se a especificação do programa. A técnica estrutural, por sua vez, utiliza a estrutura interna do programa. Já a técnica baseada em erros, considera os principais tipos de erros cometidos na codificação da implementação, ao longo do processo de desenvolvimento. Apesar de muitos dos critérios dessas técnicas terem sido propostos quando o paradigma procedural ainda era o único empregado pela indústria de desenvolvimento, os mesmos também aplicam-se a programas orientados a objetos. Entretanto, não se tem conhecimento de estudos experimentais, até o presente momento, que tenham comparado esses critérios de teste com relação aos paradigmas.

A comparação entre técnicas e critérios de teste é realizada utilizando-se algumas propriedades de teste. Dentre as mais importantes, citam-se o custo, eficácia e dificuldade de satisfação (*strength*).

O custo de teste pode ser entendido de várias maneiras. Em se tratando de “custo de critérios”, por exemplo, a métrica mais utilizada pelos estudos experimentais relaciona-se à medição do tamanho do conjunto de teste (total de casos de teste) gerado para satisfação do critério avaliado. Uma outra opção consiste em medir o número de elementos requeridos gerados e elementos requeridos não-executáveis para esse critério. Para a análise do custo da atividade de teste, todavia, outras métricas devem ser consideradas como os tempos gasto na derivação dos elementos

requeridos, construção e implementação dos casos de teste e identificação de elementos requeridos não-executáveis.

Ressalta-se ainda a importância de se avaliar a atividade de teste considerando-se os diferentes domínios de aplicação. O teste de aplicações comerciais diferencia-se por exemplo do teste de sistemas críticos, em que o tempo de resposta deve ser considerado como um requisito importante para avaliação dos teste. Sistemas concorrentes também podem apresentar, em decorrência do paralelismo, diferenças quanto à estratégia de teste aplicada para detecção de erros durante os testes. Outros exemplos típicos são sistemas embarcados, nos quais restrições de memória e processamento devem ser considerados e aplicações Web, em decorrência de propriedades como arquitetura distribuída, comportamento dinâmico e escalabilidade.

Ainda que seja uma discussão recorrente decidir qual dos dois paradigma de desenvolvimento - procedural ou OO - conduz projetos de software a melhores resultados como por exemplo, ganhos de produtividade, diferenças de performance, melhora na manutenibilidade ou reusabilidade do produto, cabe ressaltar que durante uma comparação desse tipo não deve-se subestimar a influência dos mais diversos fatores envolvidos. Esses fatores consistem, além dos próprios paradigmas, da natureza de cada programa avaliado (tamanho, estrutura, complexidade), a forma como esses programas foram escritos, a experiência das pessoas envolvidas, as dificuldades e facilidades obtidas pela adoção de cada ferramenta específica, o rigor e o contexto em que a comparação é feita etc. Essa gama de possibilidades deixa claro que, afirmar que um determinado paradigma é “melhor” ou “pior” do que outro, sem delimitar as variáveis e o contexto em que a comparação é feita, pode conduzir a conclusões precipitadas e imaturas sobre a questão. Sendo assim, uma metodologia experimental de pesquisa faz-se necessária durante esse tipo de investigação.

A Engenharia de Software Experimental tem motivado o emprego de procedimentos para o planejamento, condução e avaliação de estudos experimentais dentro da engenharia de software. A área de testes por sua vez, apesar de compreender diversos estudos experimentais a respeito das estratégias, técnicas e critérios de teste existentes, apresentou até um passado não muito distante uma forma de experimentação que não permitia a repetição/ampliação desses estudos em outros cenários. Com o recente amadurecimento e difusão dos conceitos da engenharia de software experimental, entretanto, a academia tem se conscientizado da necessidade e importância dessa metodologia na condução de estudos experimentais, o que tem tornado possível o desenvolvimento e alcance de resultados mais objetivos, claros e precisos dos temas investigados.

A condução de um experimento controlado ou mesmo um *quasi*-experimento, em geral, é uma atividade cara e que exige muitas prerrogativas para uma execução adequada, como por exemplo planejamento correto, conhecimento aprofundado do domínio investigado, tempo disponível, gastos, organização, experiência teórica e prática do projetista do experimento, além é claro de artefatos e participantes disponíveis para aplicação dos tratamentos. Um dos grandes problemas enfrentados pelos pesquisadores, durante a realização de novos estudos experimentais em teste de software é a falta de uma infra-estrutura de materiais, ferramentas e resultados adequadamente preparados, para serem usados como recurso ou embasamento do novo experimento. Essa não

é uma preocupação nova e esforços nesse sentido podem ser constatados em Rothermel (2005). Nesse trabalho os autores fazem um *survey* de vários artigos com resultados experimentais sobre técnicas de teste, reportando as dificuldades encontradas na condução dos mesmos, além de analisarem vários desafios enfrentados pelos pesquisadores durante a execução de experimentos controlados. Em resposta a esses obstáculos, os autores apresentam uma infra-estrutura a qual, em suma, é constituída de programas, versões, casos de teste, defeitos e *scripts* já estabilizados e que estariam prontos para realização e replicação de experimentos controlados.

Além da preocupação com a definição de novos experimentos em teste de programas, em Myers et al. (2004), nota-se preocupação com o ensino da atividade de teste desde o aprendizado da própria prática de programação. Segundo o autor, esta medida seria uma forma de habituar os alunos desde cedo à realização dos testes juntamente com o desenvolvimento do programa, de uma forma sistemática e integrada. Uma outra questão relacionada é tratada em (Osterweil, 1996), no qual observa-se que a maturação da pesquisa em teste de software não tem sido acompanhada da prática pela indústria na mesma proporção e um esforço conjunto entre ambas as partes deve ser tomado para uma incorporação efetiva dessa atividade no processo de desenvolvimento.

## 1.2 Motivação

Dentro do contexto em que este trabalho se insere, as principais motivações para o seu desenvolvimento são:

- O Teste de software é uma das atividades mais utilizadas para a garantia da qualidade no desenvolvimento de software;
- O paradigma de programação adotado pode apresentar impacto sobre a realização da atividade de teste;
- A avaliação experimental é fundamental para comparar técnicas e critérios de teste aplicados em diferentes contextos;
- A necessidade de avaliar propriedades da atividade de teste considerando o paradigma de programação envolvido, assim como o domínio de aplicação;
- A carência de artefatos para que estudos experimentais em teste de software sejam conduzidos;
- A necessidade recorrente de material de apoio para o ensino e treinamento na área de teste de software.

## 1.3 Objetivos

O objetivo deste trabalho é conduzir um estudo experimental em teste de software, comparando critérios de teste estruturais aplicados a um conjunto de programas do domínio de estrutura de dados, implementados nos paradigmas OO e Procedimental. Com o alcance desse objetivo pretende-se:

- Conseguir dentro de um contexto restrito e bem definido, realizar um estudo que caracterize e avalie o custo e a dificuldade de satisfação (*strength*) de critérios de teste funcionais e estruturais para programas nos paradigmas procedural e OO. Essa informação pode ser útil para esclarecer o impacto do paradigma sobre cada critério de teste e assim ajudar na escolha de qual abordagem adotar em um projeto, ou pelo menos, estimar com mais clareza as diferenças dos testes, pelo uso do paradigma adotado. Para tanto, a avaliação deverá seguir as etapas estabelecidas para o processo de experimentação, conforme definido pela engenharia de software experimental e portanto constará das principais tarefas envolvidas nas fases de planejamento, condução e análise dos dados.
- Como pode-se notar, nem sempre todos os recursos necessários estão alocados ou ao alcance do experimentador o que torna a execução dessa atividade, na prática, muito mais árdua e dispendiosa do que possa parecer. Sendo assim, o segundo objetivo do trabalho está direcionado no sentido de colaborar com a definição de futuros experimentos e pacotes de laboratório. Para tanto, busca-se a consolidação de um conjunto de critérios, ferramentas, programas e resultados que caracterizem o domínio investigado, viabilizando a replicação/ampliação de novos estudos experimentais com base nos materiais e resultados gerados.
- O terceiro objetivo do trabalho está comprometido com o desenvolvimento de um arcabouço de artefatos que possam ser reaproveitados no contexto de treinamento e capacitação em testes. Sendo assim, uma das consequências desse objetivo foi a escolha do domínio de estrutura de dados para seleção dos programas a serem utilizados na investigação do trabalho. Essa decisão teve como objetivo ampliar a gama de problemas envolvidos com cada solução testada, viabilizando ao aluno situações diferentes para a aplicação de cada técnica aprendida e atrelando o aprendizado de testes às fases iniciais do ensino de programação e desenvolvimento de software.

## 1.4 Organização

Este trabalho está organizado em seis capítulos. No Capítulo 2 é descrita a atividade de Teste de Software. São abordados os fundamentos, termos, fases, técnicas, critérios e ferramentas de teste existentes, considerando o paradigma procedural e OO.

No Capítulo 3 são apresentados os conceitos sobre a Engenharia de Software Experimental. São abordados os principais tipos de estudos experimentais existentes e o processo de experimentação para a condução de um estudo experimental. Ao final do capítulo são apresentados alguns trabalhos relacionados ao tema desta dissertação, ou seja, estudos experimentais que compararam critérios de teste e os principais resultados obtidos.

No Capítulo 4 são descritas as etapas preparatórias do processo de experimentação. Dessa forma, inicialmente é descrita a definição do experimento, no qual identifica-se a necessidade e viabilidade de obtenção dos resultados por meio da execução de um estudo nos moldes de um experimento. Em seguida é formalizado o plano, que estabelece o contexto, as hipóteses investigadas, as variáveis envolvidas, o projeto (ou *design*) experimental e a forma de instrumentar e de avaliar os resultados do experimento. Esse plano serve como roteiro para o experimentador durante a condução do restante do experimento.

No Capítulo 5 são descritos os passos seguintes na execução do estudo, ou seja, todos os detalhes sobre a forma como o experimento foi conduzido, os dados coletados e a análise realizada sobre os resultados obtidos.

Por fim, o Capítulo 6 contém as conclusões deste trabalho e trabalhos futuros identificados.

# Teste de Software

## 2.1 Considerações Iniciais

Este capítulo apresenta os principais conceitos sobre a atividade de teste de software. Para tanto, aspectos fundamentais, técnicas, critérios e ferramentas de teste são apresentados.

Na Seção 2.2 conceitos e termos básicos da área de teste são descritos. Na Seção 2.3 é apresentada a técnica de teste funcional, bem como os principais critérios vinculados a esta. A Seção 2.4 discute a técnica de teste estrutural e os critérios baseados em fluxo de controle e fluxo de dados. As Seções 2.5 e 2.6 abordam, respectivamente, as técnicas de teste baseadas em erro e em modelos. Na Seção 2.7 são apresentadas as particularidades do teste estrutural e de mutação para o paradigma OO. A Seção 2.8 é dedicada à análise de ferramentas de teste disponíveis.

## 2.2 Fundamentos do Teste de Software

Teste de software é considerado uma atividade de garantia de qualidade e pode ser aplicada em todas as fases do ciclo de desenvolvimento. Seu objetivo consiste em revelar a presença de erros em programas Myers (1978).

Dentro da área de teste existe a necessidade de se definir claramente algumas palavras-chaves, evitando desta forma, confusões e ambigüidades acerca dos seus significados. Para o contexto deste trabalho, será adotado o padrão IEEE 610.12-1990(IEEE, 1990) para classificar os termos defeito, engano, erro e falha<sup>1</sup>:

<sup>1</sup>Exceto quando se tratar de trechos ou referências a trabalhos de outros autores, no qual serão mantidos os termos originais

- defeito (*fault*) - Passo, processo ou definição de dados incorreto (ex.: instrução ou comando incorreto);
- engano (*mistake*) - Ação humana que ocasiona um resultado incorreto (ex.: ação incorreta realizada pelo programador);
- erro (*error*) - Diferença entre valor obtido e valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa;
- falha (*failure*) - produção de uma saída incorreta com relação à especificação.

O processo de teste pode ser dividido em diferentes níveis ou fases, no qual apresenta-se primeiro o teste de unidade, em seguida o teste de integração e por último o teste de sistema.

- **Teste de Unidade**

No teste de unidade, é considerada a menor estrutura que compõe o software sob teste, podendo esta ser uma função, procedimento, método ou classe, o que dependerá do paradigma e linguagem considerados. A finalidade desse teste é encontrar defeitos de lógica e implementação em cada unidade.

O conceito de unidade depende também da interpretação do autor. De acordo com o padrão IEEE 610.12-1990 (IEEE, 1990) uma unidade é um componente de software que não pode ser subdividido. No paradigma procedural a unidade corresponde a um procedimento ou subrotina, que é a menor unidade possível de ser executada. Na orientação a objetos por sua vez, a unidade pode ser entendida como uma classe (Perry e Kaiser, 1990; Binder, 1999) ou um método (Vincenzi, 2004). A interpretação justificada pelos autores que adotam a classe como menor unidade é de que na orientação a objetos a menor unidade funcional, ou seja, que não depende necessariamente de outras para executar sozinha, é a classe. Contudo, uma classe pode ser vista como uma composição de métodos e atributos, na qual métodos colaboram entre si para executar uma determinada função. Essa interação entre métodos pode ser vista como uma integração a ser testada, o que fornece subsídios para a interpretação do método como menor unidade (Vincenzi, 2004; Colanzi, 1999).

No teste de unidade, a implementação de *drivers* e *stubs* pode ser requerida. De acordo com Vincenzi (2004) “um *driver* consiste em um elemento que coordena o teste da unidade testada  $F$ , sendo responsável por ler os dados de teste fornecidos pelo testador, repassar esses dados na forma de parâmetros para  $F$ , coletar os resultados relevantes produzidos por  $F$ , e apresentá-los para o testador. Um *stub* é uma unidade que substitui, na hora do teste, uma unidade usada (chamada) por  $F$ . Na maior parte dos casos, um *stub* é uma unidade que simula o comportamento da unidade chamada por  $F$  com o mínimo de computação ou manipulação de dados.”

- **Teste de Integração**

O teste de integração se faz necessário à medida em que as unidades do software passam a ser integradas para desempenho de uma função no software. O teste de unidade prévio sobre as unidades que serão integradas não é suficiente, pois não é capaz de detectar alguns tipos de erros relativos aos diferentes contextos em que a unidade pode ser inserida. Por exemplo, uma unidade pode sofrer influências não previstas de outras unidades, subfunções ou métodos combinados podem gerar resultados inesperados e estruturas de dados globais podem apresentar problemas (Delamaro, 1997).

Apesar dos benefícios obtidos pela complementaridade desse nível de teste, deve-se considerar que o custo do mesmo pode ser elevado caso postergue-se seu emprego durante o processo de teste. Isso porque um erro revelado no teste de integração pode levar a modificações de unidades já testadas, o que implica em novos testes de unidade e de integração. Logo, o ideal é iniciá-lo tão cedo quanto possível e paralelamente ao teste de unidade. Isso minimiza custos de teste e colabora para melhora da qualidade do software em desenvolvimento.

- **Teste de Sistema**

O teste de sistema é aplicado quando as partes do software se encontram integradas e funcionando. Esse nível de teste serve para detectar possíveis erros na integração do software com recursos (banco de dados, serviços) e plataforma (hardware) ao qual está vinculado, verificando assim se o funcionamento e o desempenho obtido correspondem ao esperado. Nessa etapa de teste são observados também requisitos não funcionais como por exemplo, requisitos de desempenho, estresse e segurança.

Independentemente da fase de teste, algumas etapas para execução dos testes são definidas: (i) planejamento, (ii) projeto de casos de teste, (iii) execução e (iv) análise dos resultados (Myers et al., 2004; Beizer, 1990; Pressman, 2002). Essas etapas devem ser desenvolvidas ao longo do próprio processo de desenvolvimento do software.

Para garantir a ausência de erros em um programa, a realização de testes considerando-se todos os valores do domínio de entrada (teste exaustivo) se faz necessária. Esse tipo de teste, contudo, é reconhecido em geral como impraticável (Fabbri et al., 2007). Desta forma, técnicas e critérios de teste vêm sendo definidos, os quais definem o tipo de informação que deve ser utilizada para selecionar casos de teste com maior probabilidade de identificar os erros existentes. As principais técnicas de teste para programas são as técnicas funcional, estrutural e baseada em erros.

Uma técnica de teste é composta por *critérios de teste*. De acordo com (Zhu et al., 1997), um critério de teste define os requisitos de teste que precisam ser cobertos para avaliar a qualidade dos testes. A satisfação de um critério de teste se dá por meio de casos de teste que atendam os requisitos de teste. Os requisitos de teste são propriedades derivadas do artefato de software a serem satisfeitas. Essas propriedades podem estar relacionadas ao universo de entrada do software,

à sua estrutura (ou a um modelo representativo da mesma). Por exemplo, um critério poderia estabelecer que todas as instruções do programa devem ser testadas; nesse caso, cada instrução seria um requisito de teste. Um caso de teste é “um conjunto de entradas, condições de execução e saídas esperadas de um determinado programa ou unidade, desenvolvido com um objetivo específico (como executar uma determinada parte do software ou verificar a conformidade com um requisito específico)” (Lemos, 2006).

Desta forma, seja um programa  $P$ , um conjunto de casos de teste  $T$  e um critério  $C$ , diz-se que  $T$  é  $C$ -adequado para o teste de  $P$ , se  $T$  satisfizer os requisitos de teste estabelecidos pelo critério  $C$ .

Um critério de teste pode ser compreendido como sendo de seleção ou adequação, sendo ambos correspondentes. Um critério de seleção é um procedimento para selecionar um conjunto de teste; um critério de adequação é um procedimento para avaliar o conjunto de teste (Souza, 1996). A correspondência pode ser ilustrada da seguinte forma: Dado um critério de adequação A, existe um critério de seleção S que define: “selecione um conjunto de teste que satisfaça o critério A”. Analogamente, considere um critério de seleção S; existe um critério de adequação A que define: “Um conjunto de teste é adequado se ele é gerado pelo método S” (Maldonado et al., 2004). Logo, critério de teste é uma denominação que designa ambas definições (Maldonado, 1991).

## 2.3 Teste Funcional

A técnica de teste funcional, também conhecida por teste de caixa-preta (*black-box*) consiste em verificar para uma dada entrada fornecida se a saída gerada é consistente com aquela especificada, ou seja, a forma como o resultado obtido foi construído não importa para avaliação da satisfação ou não do teste. Para derivar os requisitos e casos de teste para a técnica funcional, o testador utiliza a especificação do programa (Beizer, 1990).

As principais fraquezas na aplicação do teste funcional estão centradas no fato de que é altamente dependente de uma definição e interpretação correta da especificação funcional do software além de não possibilitar a quantificação da atividade de teste, não garantindo portanto que pontos específicos ou críticos do software sejam explicitamente testados (Vincenzi, 2004).

Os critérios de teste da técnica funcional, em geral, buscam delimitar ou dividir o conjunto de entradas possíveis de forma a facilitar a derivação de casos de teste efetivos para a revelação de defeitos. A seguir, alguns dos principais critérios de teste funcionais existentes, para programa, (Pressman, 2002) são apresentados.

### 2.3.1 Particionamento de Equivalência

O critério Particionamento de Equivalência consiste em dividir o domínio de entrada, a partir da especificação, em intervalos (classes de equivalências) válidos e inválidos, os quais são compostos

respectivamente de dados válidos e inválidos. Cada um desses intervalos ou classes de equivalência devem representar um conjunto de valores capazes de executarem as mesmas funções de software, ou seja, testar usando um dado de teste de uma classe de equivalência teria o mesmo impacto que testar usando todos os valores daquela classe (Myers et al., 2004).

A aplicação do critério Particionamento de Equivalência se dá por meio de quatro passos:

- Identificar as classes de equivalência.
- Criar um caso de teste para cada classe de equivalência válida.
- Criar um caso de teste para cada classe de equivalência inválida.
- Comparar o resultado obtido pela execução de cada casos de teste com seu respectivo resultado esperado.

A satisfação desse critério se dá pelo exercício de pelo menos um dado de teste em cada classe de equivalência definida.

### 2.3.2 Análise do Valor Limite

Este critério é complementar ao Particionamento de Equivalência e avalia os valores limites de cada intervalo estabelecido sobre o domínio de entrada. Desta forma, testar usando este critério significa testar o valor mínimo e o antecessor imediato (ou o valor máximo e o imediatamente subsequente) nesses intervalos. Segundo Pressman (2002), os erros geralmente ocorrem nos limites dos domínios de entrada. Desta forma, pode se dizer que a aplicação desse critério serve para verificar a forma como funções do programa que funcionam adequadamente para um valor intermediário de um intervalo, funcionariam para valores limites no mesmo. Um exemplo típico de erro detectado por esse critério é aquele relacionado a comandos de iteração, onde o programador pode cometer enganos no tratamento dos valores limites, gerando defeitos que ocasionem em erros.

## 2.4 Teste Estrutural

A técnica estrutural também é conhecida como teste caixa-branca, em oposição ao teste de caixa preta, pois utiliza informações da estrutura interna (implementação) do programa para derivar os casos de teste.

A maioria dos critérios dessa técnica utiliza uma representação do programa em grafo, denominada Grafo de Fluxo de Controle (GFC) (Rapps e Weyuker, 1982). O GFC é um grafo direcionado composto por nós e arestas relacionados, onde cada nó representa um conjunto de comandos que são executados seqüencialmente e as arestas, os desvios entre esses nós. A execução de um nó implica na execução seqüencial de todos os comandos que o constituem e a execução de uma aresta,

no exercício do desvio correspondente. Um caminho em um GFC equivale a uma seqüência finita de nós ( $n_1, n_2, \dots, n_k$ ), para  $k \geq 2$ , tal que existam arestas de  $n_i$  para  $n_{i+1}$  e ( $1 \leq i < k$ ). Um caminho simples é aquele em que todos os nós que o compõem são distintos, exceto possivelmente o primeiro e o último. Caso o primeiro e último nó sejam também distintos em um caminho simples, o mesmo é dito livre de laço. Um nó de entrada no GFC é aquele no qual ocorre o primeiro comando do programa correspondente e não existe nenhuma aresta incidindo sobre o mesmo. Um nó de saída é aquele onde a computação do programa correspondente ao GFC termina e não há arestas partindo do mesmo. Um caminho completo é um caminho cujo nó inicial da seqüência é o nó de entrada, e o nó final o nó de saída do GFC (Zhu et al., 1997).

O uso da técnica estrutural é complementar à técnica funcional (Pressman, 2002), uma vez que não se pode confiar em um programa no qual certos caminhos não foram percorridos (Rapps e Weyuker, 1982). Contudo, a técnica de teste estrutural apresenta a necessidade de determinação de caminhos e associações não-executáveis<sup>2</sup> (Howden, 1987; Rapps e Weyuker, 1985). A determinação de elementos não-executáveis é um trabalho difícil de ser totalmente automatizado e que em geral depende da capacidade de inferência e raciocínio humano (Weyuker, 1990).

Os critérios de teste estruturais se subdividem em três categorias de critérios principais: critérios baseados em fluxo de controle, fluxo de dados e baseados na complexidade.

### 2.4.1 Critérios Baseados na Complexidade

Esses critérios têm seus requisitos de teste derivados da complexidade do programa. No caso do critério de McCabe, por exemplo, os requisitos de teste são derivados da complexidade ciclomática. A complexidade ciclomática é uma métrica de software que proporciona uma medida quantitativa de complexidade lógica do programa. Desta forma, essa métrica define o número de caminhos do GFC linearmente independentes que devem ser executados. Maiores detalhes sobre essa técnica podem ser encontrados em McCabe (1996).

### 2.4.2 Critérios Baseados em Fluxo de Controle

Esses critérios utilizam informações sobre o fluxo de controle para derivar os casos de teste. A justificativa para uso dos mesmos baseia-se no fato de que não se pode confiar em um programa cujos elementos de controle não tenham sido exercitados pelo menos uma vez. Dentre os principais critérios desta categoria, podem-se citar:

- Critério Todos-Nós: Diz-se que um conjunto de teste  $T$  é adequado ao critério Todos-Nós para um programa  $P$  se ele cobre todos os nós do GFC de  $P$ . Apesar de ser um critério essencial, já que verifica a execução de cada comando, não é capaz de encontrar erros bastante simples em um programa (Myers et al., 2004).

---

<sup>2</sup>um caminho não executável é um caminho impossível de ser exercitado qualquer que seja o caso de teste fornecido. Isso se deve à combinação contraditória de condições lógicas a serem satisfeitas para o exercício do caminho (Howden, 1987).

- Critério Todas-Arestas: Esse critério testa se todos os desvios estabelecidos em um programa são executados pelo menos uma vez. Um conjunto de teste  $T$  é adequado ao critério Todas-Arestas para um programa  $P$  se ele cobre todas as aresta do GFC de  $P$ . É um critério mais completo que o critério Todos-Nós já que o inclui, ou seja, a cobertura do critério Todas-Arestas garante a cobertura do critério Todos-Nós. O inverso nem sempre é verdadeiro.
- Critério Todos-Caminhos: Requer que todos os caminhos possíveis de um GFC sejam executados. Um conjunto de teste  $T$  é adequado ao critério Todos-Caminhos para um programa  $P$  se ele cobre todos os caminhos possíveis do GFC de  $P$ . Esse critério apesar de ideal é impraticável na maioria dos casos, dado que a quantidade de caminhos possíveis pode ser demasiadamente grande ou infinita até mesmo para GFC's simples.(Myers et al., 2004)

### 2.4.3 Critérios Baseados em Fluxo de Dados

Os critérios dessa classe utilizam informações sobre os dados do programa para derivar os requisitos de teste. Dentre os critérios dessa classe, destaca-se a família de critérios proposta por Rapps e Weyuker (1982, 1985). Esses critérios avaliam se a definição e uso das variáveis ao longo do programa são feitos adequadamente. Para tanto, usa uma variação estendida do GFC denominada Grafo Definição-Uso, ou Grafo Def-Uso (GDU). Esse grafo contém além da estrutura de um GFC, anotações sobre a definição e/ou uso das variáveis contidas em cada nó. A esta relação definição/uso de uma mesma variável, denomina-se *associação*. A definição de uma variável ocorre sempre que um valor é atribuído à mesma. O uso de uma variável por sua vez, pode se dar de duas formas: uso predutivo (*p-uso*), no qual a variável é usada para avaliar uma condição e uso computacional (*c-uso*), no qual a variável é usada para computar um valor.

Outro conceito importante é o de caminho livre de definição. Um caminho livre de definição para uma variável  $a$  dos nós  $j$  a  $k$ , é um caminho  $(j, n_1, \dots, n_m, k)$ ,  $m \geq 1$  no qual  $a$  é definida em  $j$  e não há redefinições de  $a$  do nó  $n_1$  até  $n_k$ , incluindo estes. (Rapps e Weyuker, 1982).

Os principais critérios dessa classe, definidos por Rapps e Weyuker são:

- Critério Todas-Definições (all-defs): Requer que cada definição de variável no GDU seja exercitada pelo menos uma vez por um c-uso ou p-uso.
- Critério Todos-Usos (all-uses): Requer que cada associação definição/c-uso e associação definição/p-uso, para toda variável existente no GDU, sejam exercitadas pelo conjunto de teste, por pelo menos um caminho livre de definição.
- Critério Todos-Du-Caminhos (all-du-paths): Requer que cada associação definição/c-uso e associação definição/p-uso, para toda variável existente no GDU sejam exercitadas pelo conjunto de teste, por todos os caminhos livres de definição e livres de laço que cubram essa associação.

Além destes, existem outros critérios de fluxo de dados como: Todos-P-Usos, Todos-P-Usos/Alguns-C-Usos e Todos-C-Usos/Alguns-P-Usos, os quais são variações menos completas do critério Todos-Usos.

Estendendo a família de critérios de fluxo de dados, Maldonado (1991) propôs a Família de Critérios Potenciais-Usos. Os critérios básicos pertencentes a essa família são: *Todos-Potenciais-DU-Caminhos*, *Todos-Potenciais-Usos/DU* e *Todos-Potenciais-Usos*. Para que o último critério citado seja satisfeito é requerido que pelo menos um caminho livre de definição seja exercitado de uma variável definida em um nó  $k$ , para todo nó e todo arco possível de ser alcançado a partir de  $k$  (Maldonado, 1991). A diferença entre os critérios definidos por Rapps e Weyuker e os da família Potenciais-Usos é que nesse último, dada a definição de uma variável, não é preciso existir o uso da mesma em um nó para que a associação seja testada. Basta que seja possível existir um uso (potencial-uso) dessa variável nesse nó por um caminho livre de definição. A essa associação definição/potencial-uso denomina-se *potencial-associação*.

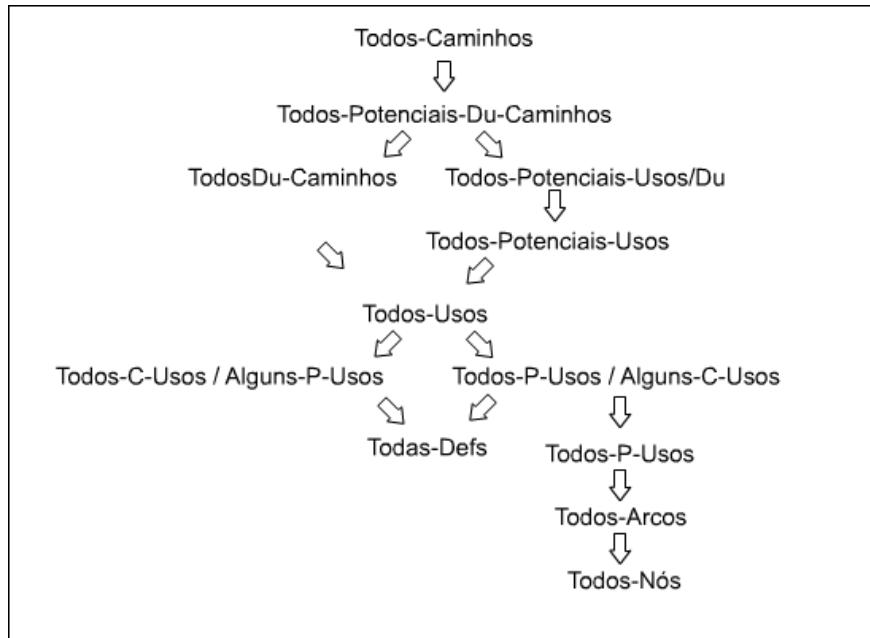
#### **2.4.4 Relação de Inclusão Entre Critérios Estruturais**

Estudos teóricos sobre critérios de teste têm sido apoiados principalmente pela análise de complexidade e relação de inclusão entre critérios. Em geral, a complexidade de um critério é determinada pela quantidade máxima de casos de teste requeridos no pior caso. Em alguns casos, como nos critérios estruturais baseados em fluxo de dados, a complexidade pode ser exponencial, o que motiva a realização de estudos experimentais para avaliar o custo prático de aplicação dos mesmos (Maldonado et al., 2004).

De acordo com Maldonado (1991), três propriedades básicas devem ser consideradas para definição de um critério  $C$ :

- incluir o critério Todas-Arestas, ou seja, um conjunto de casos de teste que exerce os elementos requeridos pelo critério  $C$  deve exercitar todas as arestas do programa;
- requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; isto equivale ao critério  $C$  incluir o critério Todas-Defs;
- requerer um conjunto de teste finito.

A relação de inclusão entre critérios estruturais, pode ser entendida conforme descrito em Rapps e Weyuker (1985): “Dados dois critérios  $C_1$  e  $C_2$  diz-se que  $C_1$  inclui  $C_2$  se para todo conjunto de casos de teste  $T_1$   $C_1$ -adequado,  $T_1$  é  $C_2$ -adequado e existe um  $T_2$   $C_2$ -adequado que não é  $C_1$ -adequado;  $C_1$  e  $C_2$  são equivalentes se para qualquer  $T$   $C_1$ -adequado,  $T$  é  $C_2$ -adequado e vice-versa”. A Figura 2.1 mostra a hierarquia de inclusão entre os principais critérios estruturais. Quanto mais alto um critério está nessa hierarquia, mais casos de teste são necessários para satisfazê-lo e por consequência mais cara é a aplicação desse critério.



**Figura 2.1:** Ordem parcial entre os critérios Potenciais-Usos básicos, Critérios de fluxo de dados e de fluxo de controle (Maldonado, 1991).

## 2.5 Teste Baseado em Erros

A técnica de teste baseada em erros (também denominada baseada em mutação ou ainda defeitos) utiliza informações sobre os tipos de erros mais comuns cometidos durante o processo de desenvolvimento de software e sobre os tipos específicos de erros que se deseja revelar (DeMillo et al., 1978). A idéia básica dessa técnica consiste em construir programas com “pequenas modificações” em relação a um programa alvo  $P$ , de forma a torná-lo defeituoso. Isso colabora para o discernimento entre erros naturais/artificiais. O principal critério dessa técnica é conhecido como Análise de Mutantes (DeMillo, 1978).

### 2.5.1 Critério Análise de Mutantes

No critério análise de mutantes, essas “pequenas modificações” são introduzidas através do que se denominam *operadores de mutação*. Tais operadores de mutação representam regras a serem aplicadas sobre um programa original  $P$  e são definidos com base nos erros que são comumente cometidos na linguagem alvo. Essas regras ditam quais alterações devem ser inseridas para construir programas ligeiramente diferentes a partir de  $P$ , os quais são denominados *mutantes*.

O objetivo do critério é definir um conjunto de teste que seja capaz de demonstrar que o programa em teste não possui os erros representados nos seus mutantes. Os mutantes são criados a partir dos operadores de mutação e representam os erros mais comuns cometidos na linguagem alvo.

A análise de mutantes baseia-se em duas hipóteses básicas:

- A *hipótese do programado competente*: “O programa escrito pelo programador competente é bem próximo do programa correto”. Assumindo a validade dessa hipótese, pode-se afirmar que defeitos são inseridos nos programas, por meio de pequenos desvios sintáticos, os quais apesar de não resultarem em erros sintáticos, modificam a semântica do programa e desta forma, conduzem o mesmo a um comportamento incorreto. Tais erros são revelados na análise de mutantes, identificando-se os desvios sintáticos mais comuns e, por meio da aplicação de pequenas transformações sobre o programa em teste, encorajando o testador a construir casos de teste que mostrem que tais transformações levam a um programa incorreto (Agrawal, 1989).
- a *hipótese do efeito do acoplamento*: “Revelando-se erros simples, erros mais complexos são também revelados”. Essa hipótese assume que erros complexos estão relacionados a erros simples. Alguns estudos empíricos (Offutt, 1989; Acree et al., 1979) confirmam essa hipótese.

A aplicação do critério análise de mutantes, dado um programa  $P$  e um conjunto de teste  $T$ , pode ser dividida em quatro passos básicos (Vincenzi, 1998):

- 1. Geração do conjunto de mutantes  $M$ :**

Os operadores de mutação são aplicados sobre um programa<sup>3</sup>  $P$  para gerar um conjunto de mutantes  $M$ .

- 2. Execução do programa  $P$ :**

O programa  $P$  é executado com o conjunto de teste  $T$  e é verificado se o resultado gerado corresponde ao esperado. Caso alguma saída incorreta seja gerada, o processo é encerrado. Do contrário, prossegue-se com o próximo passo. Em geral a tarefa de decidir se a saída gerada corresponde à esperada para cada caso de teste, é cabida ao testador, o qual passa a desempenhar o papel de oráculo (Weyuker, 1982).

- 3. Execução do conjunto de mutantes  $M$ :**

Aplica-se o conjunto de teste  $T$  a cada mutante  $m_i$  em  $M$ . Se um caso de teste é capaz de revelar um resultado em  $m_i$  diferente do resultado obtido pela aplicação do mesmo caso de teste ao programa  $P$ , então o conjunto de teste é capaz de expor a diferença  $m_i$  e  $P$ , e  $m_i$  é considerado morto. Do contrário, o mutante  $m_i$  permanece “vivo” o que pode incorrer em duas possibilidades: o conjunto de teste  $T$  não é capaz de revelá-lo e precisa ser melhorado; ou o mutante  $m_i$  é equivalente ao programa  $P$ .

Ao término da execução do conjunto de mutantes, o escore de mutação pode ser calculado. Esse escore servirá para indicar a qualidade do conjunto de teste  $T$  adotado. A fórmula para escore de mutação é a seguinte:

---

<sup>3</sup>Muitas vezes o programa original pode conter várias entidades às quais se aplique um operador de mutação, o que leva a geração de mais de um mutante, tendo em vista que o operador de mutação é aplicado sobre uma entidade de cada vez.

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)} \quad (2.1)$$

Na qual,

- $ms(P, T)$ : escore da mutação;
- $DM(P, T)$ : número de mutantes mortos por  $T$ ;
- $M(P)$ : número total de mutantes gerados;
- $EM(P)$ : número de mutantes equivalentes identificados.

O escore é sempre um valor no intervalo  $[0, 1]$ . Quanto mais próximo de 1 for o resultado, mais adequado é  $T$ .

#### 4. Análise dos mutantes vivos em $M$ :

Requer maior intervenção humana. De acordo com o escore de mutação, é decidido se o teste deve continuar. Em caso afirmativo, cada mutante  $m_k$  vivo deve ser analisado para saber se é ou não equivalente ao programa  $P$ . Como o problema de determinar a equivalência entre dois programas é um problema indecidível, faz-se necessário o uso de algumas heurísticas, por exemplo conforme em Budd (1981); Simão e Maldonado (2000). Caso o mutante seja equivalente, então deve ser descartado. Do contrário, o conjunto de teste não foi capaz de distingüí-lo do programa  $P$ . Deve-se então retornar ao passo 2, e gerar um novo conjunto de teste, refazendo o processo até que se obtenha um bom conjunto de teste.

O maior problema relacionado à aplicação desse critério é o custo. A quantidade de mutantes gerados pode ser muito grande, já que existem vários operadores de mutação e entidades às quais estes se aplicam dentro dos programas que compõem um determinado sistema. Logo, isso implica em uma elevada demanda computacional para execução dos mutantes gerados. Soma-se a isso a questão de determinar a equivalência ou não do mutante gerado em relação ao programa original, um problema indecidível do ponto de vista teórico-computacional - o que exige esforço intelectual humano para sua identificação e consequentemente, tempo. Na tentativa de reduzir esses esforços, vários trabalhos foram desenvolvidos ((Acree et al., 1979); (Mathur, 1991); (Offutt et al., 1993); (Offutt et al., 1996a); (Wong et al., 1997); (Barbosa et al., 2001)). Esses trabalhos propõem a derivação de um subconjunto menor de mutantes, mas que seja igualmente capaz de revelar as discrepâncias reveladas pelo conjunto total. Ou seja, o conjunto de teste gerado para “matar” esse subconjunto de mutantes será igual ou quase igual àquele requerido para “matar” o conjunto todo. Vale ressaltar que apesar dessas abordagens funcionarem em determinados contextos, a necessidade de identificação de mutantes equivalentes permanece.

## 2.6 Teste Baseado em Modelos

O Teste Baseado em Modelos (ou estados) consiste em representar o comportamento do sistema em um modelo baseado em estados (Máquina de Estados Finito, Statecharts, Redes de Petri, etc.) e a partir desse, gerar seqüências de testes para avaliar se o programa gerado a partir dessa especificação está correto (teste de conformidade).

Os testes baseados em Máquinas de Estados Finitos são bastante utilizados no contexto de sistemas orientados a objetos para testar comportamento, uma vez que as entidades (objetos) são dotadas de estado, comportamento e interagem em colaboração (por troca de mensagens) para desempenho de uma tarefa. Além disso, cada objeto é responsável por seu estado, sendo que esses estados são modificados em decorrência da seqüência de interação estabelecida com os outros objetos envolvidos. O comportamento resultante, portanto, é decorrente da interação conjunta do comportamento individual de cada objeto (Vincenzi, 2004). Alguns trabalhos investigam o comportamento de sistemas OO por testes baseados em estados (Kung et al., 1996; Hoffman, 1997; Binder, 1999).

Conforme observado por (Binder, 1999) esse tipo de teste não é suficiente para detecção de erros em sistemas OO, devendo ser complementado por outras técnicas de teste baseadas em programa, como os testes estruturais.

Apesar de na prática serem infinitos os valores possíveis de atributos e seqüência de mensagens trocadas entre objetos, alguma restrição sobre esses elementos pode existir, o que pode ser representado e testado por meio de máquina de estados. Além disso, modelos são adequados para representarem várias escalas de um sistema, mantendo a mesma notação. Como os sistemas OO têm seu comportamento fortemente relacionado com as distintas escalas de complexidade em que está implementado (classe, cluster, subsistema etc) o teste dos modelos que o representam é justificado dentro do paradigma em questão (Binder, 1999, 1994), contribuindo também para a qualidade de posteriores testes de conformidade.

Alguns dos métodos para geração de sequência de teste baseados em Máquinas de Estado Finito mais conhecidos são: métodos W, DS (Gönenç, 1970), UIO (Sabnani e Dahbura, 1988) e Wp (Fujiwara et al., 1991). No contexto de Statecharts, pode-se citar a família de critérios estruturais definidos por (Souza et al., 2000). Essa família de critérios foi mapeada para Redes de Petri em (Simão et al., 2003).

## 2.7 Critérios de Teste Específicos ao Paradigma Orientado a Objetos

As técnicas e critérios até aqui foram apresentados de forma geral, sem se preocupar com as particularidades inseridas pelo paradigma. Tratando-se do paradigma procedural, toda a teoria

apresentada até este ponto mostra-se suficiente para execução dos testes de unidade. No teste orientado a objetos contudo, devem ser consideradas algumas questões relativas às particularidades desse paradigma. Essa seção foi parcialmente extraída de Maldonado et al. (2007) e Vincenzi (2004).

Na programação estruturada o programa é organizado em termos de um conjunto de dados e funções/procedimentos que manipulam tais dados. Contudo, quando grandes soluções precisam ser desenvolvidas nesse paradigma, o relacionamento e dependência entre dados e funções torna-se alto e complexo de ser manipulado, devido à suscetibilidade do software desenvolvido dessa forma sofrer grandes impactos por pequenas modificações. Uma das mudanças inseridas pelo paradigma OO nesse contexto, na tentativa de contornar tais problemas, foi a diversidade de abstrações criadas, visando dentre outros a organização, hierarquização e isolamento dessas estruturas. Com relação à organização, algumas abstrações importantes são os conceitos de classe, objeto, atributos, métodos, estado e comportamento. A hierarquia estabelecida pelo paradigma OO, insere-se no contexto de herança, na qual uma classe (denominada subclasse) pode herdar métodos e atributos de outra classe já definida (denominada superclasse) e polimorfismo, característica que possibilita que uma mesma variável denote instâncias (objetos) de várias subclasses que, usualmente, devem possuir uma única superclasse em comum. O isolamento é alcançado pelo encapsulamento (uma forma de se ocultar informação e modularizar o sistema) e troca de mensagens, as quais permitem que os objetos interajam entre si, sem interferirem de forma intrusiva no funcionamento um do outro.

Conforme descrito anteriormente, a unidade de um software construído sob o paradigma OO, pode ser entendida como um método ou uma classe. Entendendo-se o método como a menor unidade sob teste, pode-se dizer que a classe funciona no teste como *driver* desse método, ou seja, recebe as entradas, repassa-as ao método na forma de parâmetro, coleta as informações relevantes e as apresenta ao usuário. Sem a classe, um método não pode ser executado e portanto testado. Desse modo, o teste de unidade é também denominado de intra-método (Harrold e Rothermel, 1994).

Considerando este contexto, pode-se notar que os métodos (unidades) pertencentes a uma mesma classe, interagem entre si, para desempenho de uma determinada função, o que caracteiza uma integração e portanto uma necessidade de teste. A esse teste de integração denomina-se teste inter-método (Harrold e Rothermel, 1994). Harrold e Rothermel (1994) definem ainda os conceitos de teste intraclasse e interclasse, nos quais o primeiro decorreria do fato do usuário de um objeto poder invocar indiscriminadamente os métodos públicos disponibilizado pelo mesmo, o que poderia levar esse objeto a um estado inconsistente; e o segundo da possibilidade de ocorrência do mesmo problema, pela invocação indiscriminada de métodos públicos em duas ou mais classes distintas. Esses tipos de teste, portanto, aumentariam a confiança de que diferentes sequências de invocações resultariam em uma interação adequada dos objetos envolvidos. O teste de sistema, por ser baseado em critérios funcionais, não apresenta diferenças relevantes entre os paradigmas procedural e OO.

## 2.7.1 Teste Estrutural para OO

Harrold e Rothermel (1994) definiram as seguintes representações de programa: Grafo de Chamadas de Classe, Grafo de Fluxo de Controle de Classe e Grafo de Fluxo de Controle de Classe Encapsulada. Essa representações servem para auxiliar no teste dos níveis intra-método, inter-método e intra-classe. Com base nessas representações Harrold e Rothermel (1994) definiram associações definição-uso para teste de fluxo de dados em programas OO em cada um desses níveis de teste.

Assim, seja  $C$  uma classe em teste,  $d$  um comando contendo uma definição de variável e  $u$  um comando contendo o uso dessa variável:

**Para o teste intra-método:** “Seja  $M$  um método de  $C$ . Se  $d$  e  $u$  estão em  $M$  e existe um programa  $P$  que chama  $M$  tal que  $(d, u)$  é um par def-uso exercitado em uma simples invocação de  $M$ , então  $(d, u)$  é um par def-uso intra-método.”

**Para o teste inter-método:** “Seja  $M_0$  um método público de  $C$  e seja  $\{M_1, M_2, \dots, M_n\}$  o conjunto de métodos que são chamados, direta ou indiretamente, quando  $M_0$  é invocado. Suponha que  $d$  está em  $M_j$ , sendo que tanto  $M_i$  quanto  $M_j$  estão em  $\{M_1, M_2, \dots, M_n\}$ . Se existe um programa  $P$  que chama  $M_0$  tal que, em  $P$ ,  $(d, u)$  é um par def-uso exercitado durante uma simples invocação de  $M_0$  por  $P$ , e  $M_i \neq M_j$  e  $M_i$  e  $M_j$  são invocações separadas do mesmo método, então  $(d, u)$  é um par def-uso inter-método.”

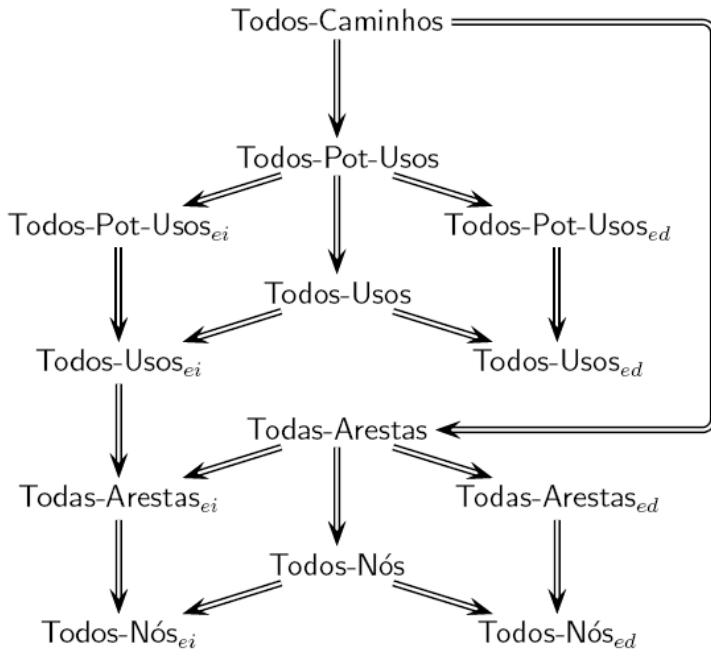
**Para o teste intraclassse:** “Seja  $M_0$  um método público de  $C$  e seja  $\{M_1, M_2, \dots, M_n\}$  o conjunto de métodos que são chamados, direta ou indiretamente, quando  $M_0$  é invocado. Seja  $N_0$  um método público de  $C$  e seja  $\{N_1, N_2, \dots, N_n\}$  o conjunto de métodos que são chamados, direta ou indiretamente, quando  $N_0$  é invocado. Suponha que  $d$  está em alguns dos métodos em  $\{N_1, N_2, \dots, N_n\}$ . Se existe um programa  $P$  que chama  $M_0$  tal que, em  $P$   $(d, u)$  é um par def-uso exercitado durante uma simples invocação de  $M_0$  e  $N_0$ , tal que  $(d, u)$  é um par def-uso e que a chamada a  $M_0$  é feita após  $d$  ter sido executado e  $M_0$  encerra sua execução antes que  $u$  seja executado, então  $(d, u)$  é um par def-uso intraclassse.

Em Vincenzi (2004) um conjunto de critérios estruturais foram propostos para derivação de requisitos a partir de *bytecodes* de programas OO implementados em Java. A motivação para isso é permitir que não só código Java seja testado, mas que o teste estrutural possa ser aplicado sobre qualquer componente escrito em linguagem que gere *bytecodes*. Isso é especialmente interessante em cenários nos quais o código fonte não esteja disponível, apenas o código objeto. Além dos critérios, uma ferramenta denominada JaBUTi, foi desenvolvida para apoiar a aplicação desses critérios. Para aplicar esses critério de teste entretanto, é preciso apresentar antes os conceitos de Grafo de Instruções (GI) e Grafo Def-Uso (GDU). O Grafo de instruções consiste de um grafo no qual cada “nó” é composto por uma instrução *bytecode* e sua numeração é feita em termos do contador de programa dessas instruções. As arestas nesse grafo podem ser de dois tipos: regulares, correspondente à existência de transferência de controle entre as respectivas instruções; e de exceção, considerando a tabela de exceções. Os desvios condicionais são identificados pela

semântica das instruções responsáveis por tais comandos. Da mesma forma, as definições e usos de variáveis são identificadas por meio da análise semântica sobre as instruções de *bytecodes*. O Grafo de Instrução, apesar de consistir em uma forma precisa de se percorrer o conjunto de instruções em um método, identificando variáveis definidas e usadas em cada instrução, em geral resulta em grafos de grandes proporções até para métodos relativamente pequenos, por conter um grande número de nós e arestas. Assim, quando todas as informações necessárias, relativas ao conjunto de instruções correspondentes a cada método são coletadas, as instruções que são executadas em seqüência podem ser compactadas em um único conjunto (ou bloco). A execução em seqüência é determinada pelo fluxo de controle normal das instruções do método e não considera a influência de possíveis interrupções. Dessa forma, cada conjunto ou bloco formado dá origem a um nó no GDU. As definições e usos de variáveis em cada nó do GDU são determinados pela união das definições e usos de variáveis identificadas em cada instrução que compõem esse novo nó. Com base nessa nova estrutura formada (GDU) são derivados os requisitos de teste para oito critérios de teste estruturais baseados em fluxo de controle e em fluxo dados, propostos em Vincenzi (2004):

- Critério Todos-Nós<sub>ei</sub>: “Exige a cobertura de todos os comandos não relacionados ao tratamento de exceção.”
- Critério Todos-Nós<sub>ed</sub>: “Exige a cobertura de todos os comandos relacionados ao tratamento de exceção.”
- Critério Todas-Arestas<sub>ei</sub>: “Exige a cobertura de todos os desvios condicionais do método (desvios não decorrentes do lançamento de uma exceção).”
- Critério Todas-Arestas<sub>ed</sub>: “Exige a cobertura de todos os desvios de execução decorrentes do lançamento de uma exceção.”
- Critério Todos-Usos<sub>ei</sub>: “Exige a cobertura de todas as associações definição-uso não relacionadas ao tratamento de exceção.”
- Critério Todos-Usos<sub>ed</sub>: “Exige a cobertura de todas as associações definição-uso relacionadas ao tratamento de exceção.”
- Critério Todos-Pot-Usos<sub>ei</sub>: “Exige a cobertura de todas as potenciais-associações definição-uso não relacionadas ao tratamento de exceção.”
- Critério Todos-Pot-Usos<sub>ed</sub>: “Exige a cobertura de todas as potenciais-associações definição-uso relacionadas ao tratamento de exceção.”

Vincenzi (2004) estabelece uma hierarquia entre os seus critérios estruturais e os critérios tradicionais de fluxo de controle e de dados. Essa hierarquia é apresentada na Figura 2.2.



**Figura 2.2:** Hierarquia entre os critérios de testes estruturais intramétodos. Fonte: (Vincenzi, 2004).

## 2.7.2 Teste de Mutação para OO

Conforme discutido em Maldonado et al. (2007), a maioria dos trabalhos que investigam mutantes para OO estão voltados para as linguagens C++ e Java. Devido à semelhança sintática entre a linguagem C e essas linguagens, Vincenzi (2004) investiga o reaproveitamento dos operadores de mutação disponíveis para C que implementam os mesmos tipos de defeitos nessas linguagens OO.

A partir das conclusões obtidas em (Maldonado et al., 2007), sobre o reaproveitamento dos operadores de mutação, tem-se que para C++ todos os operadores são aproveitáveis, uma vez que C++ é um superconjunto da linguagem C. Em Java contudo, o reaproveitamento foi de 73.75% ( 59 dos 80 operadores considerados). Os motivos que impediram que 20 desses operadores não fossem reaproveitados são causas decorrentes da inexistência em Java de determinados comando e estruturas presentes em C (goto, struct, ponteiros, por exemplo). Além disso, toda expressão lógica em Java é do tipo booleana, o que impede que alguns enganos como, uso de "=" ao invés de "==" na comparação de igualdade, sejam cometidos, o que diminui o número de operadores de mutação usados em expressões lógicas.

Assim como alguns comandos e estruturas não existem na linguagem Java mas estão presentes em C, o inverso também é verdadeiro. Logo, com o objetivo de suprir esta carência para Java, Vincenzi (2004) estabeleceu sete operadores de mutação adicionais aos já existentes.

Além disso, observou-se que os operadores de mutação considerados essenciais para o teste de unidade em programas C (Barbosa et al., 2001) também são aplicáveis em programas Java e C++.

(Vincenzi, 2004).

## 2.8 Ferramentas de Teste de Software

A atividade de teste envolve em geral a manipulação sistemática de uma quantidade significante de dados e programas. A aplicação de critérios de teste manualmente pode exigir um grande esforço humano e muitas vezes é impraticável. Sendo assim, o uso de ferramentas de teste que apóiem essa atividade pode representar uma diminuição representativa dos custos envolvidos e melhorar a produtividade do testador, já que reduz os riscos de erros cometidos pelo mesmo e permite a automatização de algumas tarefas. Além disso, a ausência de uma ferramenta para aplicação dos critérios poderia limitar o testador a uma quantidade pequena e muito simples de programas (Horgan e Mathur, 1992). A comunidade científica e a indústria têm se empenhado na definição e construção de algumas ferramentas que se aplicam às técnicas de teste discutidas.

Para o teste de programas escritos em linguagem procedural, algumas ferramentas foram encontradas:

- *xSuds* (Agrawal et al., 1998): consiste em um conjunto de ferramentas de apoio ao teste, análise e depuração de programas. Uma das ferramentas do xSuds é a ATAC (Horgan e London, 1991) que apóia a aplicação de critérios de teste estrutural em programas escritos em C e C++.
- *Cantata++*: ferramenta comercial desenvolvida pela IPL Software Products Group(IPL, 2008). Algumas de suas características são: teste de unidade e de integração; cobertura de comandos, nós, arestas, condições. Além disso, permite configuração dos requisitos de cobertura pelo usuário por meio da definição de regras simples; opção de remoção/desabilitação de casos de teste que não contribuem para aumento da cobertura; geração de relatórios em formato “.cvs”; construção de Stubs e Wrappers para simular e controlar interfaces externas; permite teste de cobertura nas linguagens ANSI C (89 and 99), ISO C++, EC++ e Java;
- *Proteum(Program Testing Using Mutants)* (Delamaro e Maldonado, 1996): desenvolvida pelo grupo de Engenharia de Software no Instituto de Ciências Matemáticas de São Carlos - USP, oferece alguns recursos aos testadores, dos quais destacam-se: definição de casos de teste; execução do programa sob teste; seleção dos operadores de mutação<sup>4</sup> e definição de suas respectivas porcentagens, para geração dos mutantes; aplicação dos casos de teste sobre os mutantes; análise dos mutantes remanescentes e escore de mutação. A *Proteum* dispõe ainda de interface gráfica e suporte multi linguagem o que permite a configuração da ferramenta para programas escritos em outras linguagens de programação (Delamaro e

<sup>4</sup>Na Proteum os operadores de mutação disponíveis podem ser divididos em quatro classes: mutação sobre comandos, mutação sobre variáveis, mutação sobre constantes e mutação sobre operadores.

Maldonado, 1993). Além disso, disponibiliza 71 operadores de mutação. Trata-se de uma ferramenta orientada a **sessão de teste**, ou seja, a atividade de teste pode ser desenvolvida em etapas pois a ferramenta armazena os estados intermediários da aplicação de teste, possibilitando o testador retomar um teste a partir do ponto em que o mesmo foi interrompido, sem a necessidade de reiniciá-lo (Delamaro, 1993). A *Proteum/IM* (Delamaro, 1997) é uma extensão da versão original da *Proteum*, desenvolvida para apoiar o critério Mutação de Interface. Logo, possui operadores de mutação específicos para esse critério. Além dos operadores de mutação, o que a diferencia da *Proteum* original é o fato de oferecer recursos para testar a conexão entre as unidades do software. Mais recentemente, as ferramentas *Proteum* e *Proteum/IM* foram integradas em um único ambiente de teste *Proteum/IM 2.0* (Maldonado et al., 2000b), apoiando o teste de unidade e de integração em programas procedimentais (Domingues, 2002).

- *Poke-Tool*: desenvolvida por Chaim (1991), apóia a aplicação dos critérios Todos-Nós, Todas-Arestas, Todos-DU-Caminhos, Todos-P-Usos, Todos-Usos, Todos-Potenciais-Usos e Todos-Potenciais-Usos/DU para programas escritos em linguagem C. Além disso, trabalha de forma integrada com a ferramenta ViewGraph (Vilela et al., 1997) que permite a visualização do GFC correspondente ao programa sob teste. A ferramenta é orientada à sessão e algumas de suas funcionalidades incluem: geração de sessão de teste; inserção e execução de casos de teste; instrumentação do programa sob teste; avaliação da cobertura dos testes; geração do conjunto de associações necessárias para satisfazer o critério; associações não exercitadas e GDU correspondente ao programa em teste.

Dentre as ferramentas para teste de programas escritos em linguagem OO, podem-se citar:

- *Cobertura*: a qual está disponível gratuitamente para uso (Doliner, 2005). A ferramenta aplica teste estrutural baseado em controle (cobertura de comandos e desvios) sobre programas escritos em Java, instrumentando-os automaticamente. Além disso, gera relatórios gráficos indicando a porcentagem de cobertura de comandos e desvios atingidos após a execução dos casos de teste, para cada classe do programa.
- *C++ Test* (PARASOFT Corporation, 2002a): ferramenta para teste de unidade em códigos escritos em C/C++. Realiza teste funcional, estrutural e de regressão. Permite a criação automática de *drivers* e *stubs*, por meio da especificação dos valores de retorno, ou entrada manual dos *stubs* pelo testador. Automatiza a geração de casos de teste e resultados esperados. Gera e executa automaticamente casos de teste para o teste estrutural, armazenando-os para o teste de regressão (Domingues et al., 2002).
- *Jtest* (PARASOFT Corporation, 2002b): realiza testes sobre classes escritas em linguagem Java. Permite análise estática, teste funcional, teste estrutural e de regressão. Gera automaticamente um conjunto essencial de casos de teste para o teste funcional, visando atingir

uma cobertura tão completa quanto possível. O usuário pode complementar o conjunto de teste com seus próprios casos de teste. Apresenta os resultados da execução do conjunto de teste em forma de árvore, permite que o testador validar os resultados e notifica o mesmo no caso de ocorrerem erros no teste de regressão ou funcional. Na análise estática a ferramenta previne erros padronizando o código o que consequentemente, reduz a possibilidade de erros serem inseridos no mesmo (Domingues et al., 2002).

- *Panorama C/C++* (International Software Automation, 1999) e *Panorama for Java*: pacote de cinco ferramentas composto por: OO-Test, OO-SQA, OO-Analyser, OO-Browser e OO-Diagrammer. Colaboram para as atividades de teste de software, garantia da qualidade e reengenharia e auxiliam nas fases de projeto, codificação e documentação do software. A OO-Test executa análise de cobertura de arestas, análise da freqüência das arestas executadas, análise da eficiência dos casos de teste e minimização do conjunto de casos de teste (Domingues et al., 2002).
- *PureCoverage* (RATIONAL Software Corporation, 2000) ferramenta para análise de cobertura para códigos C++ e Java. Verifica as áreas do código que foram ou não exercitadas pela execução dos testes. Permite vários tipos de visualização dos resultados, auxiliando o testador a compreender quais regiões do código foram ou não testadas (cobertura de código em arquivo, módulo e linha de código) (Domingues et al., 2002).
- *JaBUTi*, desenvolvida por (Vincenzi et al., 2003), constitui um ambiente completo para teste estrutural de programas OO em Java. Permite o teste em programas cujo código executável constitua-se de bytecodes e portanto é útil para o teste quando o programa fonte não está disponível (teste de componentes). Permite automatização de várias tarefas como instrumentação de código, vizualização de GFC, porcentagem atingida pelos casos de teste na satisfação de cada um dos critérios, interação por interface gráfica etc.
- *MuJava* (Ma et al., 2006). MuJava é uma ferramenta desenvolvida em âmbito científico e tem como propósito estimular estudos experimentais envolvendo mutantes. Dentre as funcionalidades oferecidas pela mesma, destacam-se: geração e execução de mutantes; implementação operadores de mutação tanto a nível de classe como de métodos; disposição de interface gráfica para interação; cálculo de escore de mutação para os conjuntos de teste aplicados sobre os mutantes gerados; definição de conjunto de teste em uma classe separada, na qual cada método representa um caso de teste.

Conforme pôde ser observado, muitas dessas ferramentas foram desenvolvidas em âmbito acadêmico e estão sendo constantemente evoluídas para atender novas necessidades e incorporar outras funcionalidades. De qualquer forma, a colaboração e apoio à atividade de teste às quais essas ferramentas têm se prestado na comunidade científica e indústria revelam sua importância no aumento da qualidade e confiabilidade do software sob teste, viabilizando a automatização e siste-

matização de tarefas que estão propensas ao erro humano, seja pelo esforço repetitivo ou por sua capacidade natural de cometer erros.

## 2.9 Considerações Finais

Neste capítulo foram apresentados os principais conceitos relacionados à atividade de teste de software. As técnicas de teste funcional, estrutural e baseada em erros foram descritas, assim como os seus principais critérios. Além disso, uma seção foi dedicada ao teste no paradigma OO. Ao final, algumas ferramentas desenvolvidas para apoiar o teste de programas procedural e OO foram apresentadas.

Pode-se observar que existe uma quantidade considerável de critérios de teste disponíveis e portanto, é extremamente útil avaliar os benefícios e custos associados aos mesmos. Nesse cenário, o desenvolvimento de estudos experimentais vem se destacando. No próximo capítulo são apresentados os fundamentos e processos aplicados na condução de estudos experimentais, assim como os trabalhos já desenvolvidos, relacionados à avaliação comparativa de critérios de teste.

# Engenharia de Software Experimental

## 3.1 Considerações Iniciais

Na engenharia de software experimental os estudos experimentais relacionam-se à necessidade de conseguir resultados objetivos e significativos para a compreensão, controle e avaliação de técnicas e tecnologias de desenvolvimento de software (Höhn, 2007). Essa necessidade originou-se de crenças muito fortes dentro da área, que dizem respeito à forma como o software tem sido construído, ou seja, técnicas são aplicadas confiando-se nos resultados que estas podem gerar; a capacidade das pessoas envolvidas completarem um projeto é uma crença presente; o uso de uma determinada ferramenta é entendido como uma forma de se reduzir o tempo de desenvolvimento, entre outras. Grande parte dessas “crenças” estão sujeitas à contestação quanto à validade e o contexto em que se aplicam. Além disso, revelam o grau de incerteza sobre os fundamentos nos quais a engenharia de software tem-se se apoiado (Juristo e Moreno, 2001), o que motiva estudos mais precisos e que sejam capazes de dar um respaldo científico (e menos especulativo) acerca dessas questões, a exemplo do que é feito em outras engenharias.

A Seção 3.2 apresenta os fundamentos relacionados à realização de estudos experimentais dentro da engenharia de software. Para tanto, apresenta-se inicialmente os principais tipos de estudos experimentais existentes, alguns conceitos e termos básicos adotados pela área e em seguida, o processo de experimentação usado na condução de um experimento controlado.

A Seção 3.3 aborda os principais estudos experimentais realizados na área de teste de software relacionados à avaliação de critérios de teste. A seção faz uma análise sobre comparação de custo e eficácia em estudos experimentais aplicados a teste de software, apresenta os estudos experimentais realizados que avaliam critérios estruturais e baseados em erro e trata dos trabalhos

de experimentação que envolvem a técnica de teste funcional. Na Seção 3.3.1 são mostrados os resultados de um trabalho comparando vários estudos experimentais semelhantes que aplicam técnicas de teste para programas. Por fim, a Seção 3.3.2 faz algumas considerações sobre questões relacionadas à validade externa de estudos experimentais.

As considerações finais são feitas na Seção 3.4.

## **3.2 Engenharia de Software Experimental**

Os estudos experimentais servem para caracterizar uma tecnologia ou método em uso dentro de um determinado contexto (Mafra, 2005). Dentro da engenharia de software experimental várias classificações para tipos de experimentos foram propostas. Contudo, a mais utilizada e aceita pela literatura, categoriza os estudos primários em um desses três tipos: pesquisa de opinião (*survey*), estudo de caso e experimento controlado.

Nas áreas de ciências aplicadas, tais estudos geralmente correspondem a diferentes níveis de evolução na condução do experimento. Por exemplo, na área farmacêutica, quando se deseja analisar a eficácia de uma substância sobre uma bactéria, primeiro é estudado se esta mesma substância tem algum efeito sobre a bactéria. Desta forma, isolam-se bactéria e substância em tubo, controlando todas as outras condições indesejadas que possam interferir no resultado. Essa primeira fase é realizada *in vitro* ou seja, em condições controladas de laboratório e seus resultados devem ser replicáveis por outros pesquisadores. Uma vez que os resultados conduzidos mostram-se promissores, uma nova fase pode ser seguida. Do contrário, os pesquisadores precisariam tomar novos rumos em suas pesquisas. Nessa outra fase, as condições são menos controladas. Retomando o exemplo da indústria farmacêutica, nesta etapa, ao invés de tubos, seres vivos seriam usados, por exemplo animais de laboratório. Caso os conjuntos dos resultados fortaleçam as evidências, o experimento é passado para a terceira e última fase. Nesta, um estudo gradual é feito sobre conjuntos de pessoas. Caso os resultados obtidos sejam satisfatórios, a substância é incorporada pela indústria em forma de remédios a serem administrados a pacientes (Juristo e Moreno, 2001).

Analogamente, no desenvolvimento de software, uma nova idéia pode ser verificada inicialmente pelos seus inventores, em laboratório. Um experimento de laboratório dessa forma, corresponde a um projeto de desenvolvimento, sem pressão de mercado e prazos. Além disso, o processo usado e pré-conhecimento dos participantes dentre outros fatores, podem ser controlados. A fase seguinte onde os experimentos são do tipo estudos de caso e os participantes pioneiros de tecnologia, revela os limites da inovação proposta por meio desses experimentos do tipo *in vivo*. Somente após comprovação dos limites e conhecimento dos riscos usando-se a nova idéia, outras parcelas da indústria assumirão a proposta, conhecendo as possíveis vantagens que terão. Os resultados dessa nova técnica/proposta serão publicados por alguns desenvolvedores possibilitando que após alguns anos sejam confirmadas como uma solução de fato ao invés de mera especulação pela comunidade (Juristo e Moreno, 2001).

De acordo com Juristo e Moreno (2001), um experimento na engenharia de software pode assumir as três fases ou tipos apresentados nas situações anteriores, nos casos em que se assemelham a uma ciência aplicada, ou consistir de apenas um destes. Contudo, sabe-se que a comunidade de software não segue tão a risca essas fases de experimentação para tirar vantagem dos benefícios de uma nova idéia assim como as demais áreas de conhecimento o fazem.

Na engenharia de software a diferenciação ou classificação de um estudo em um determinado tipo depende principalmente do grau de controle e objetivos pretendidos para os mesmos. A seguir uma descrição acerca de cada um desses estudos é apresentada.

1. **Pesquisa de opinião (survey):** Trata-se de um tipo de estudo que visa caracterizar retrospectivamente uma ferramenta ou técnica já usada ou em uso. De acordo com Travassos (2002) os objetivos de um *survey* são descritivos, explanatórios ou explorativos. No primeiro caso, a intenção é determinar a distribuição de características, permitindo inferências sobre algo; no segundo, explicar-se as razões de um fenômeno ou fato; e o terceiro, levantar subsídios iniciais para que uma investigação mais profunda possa ser feita.

A forma de se coletar dados qualitativos e quantitativos para o *survey* se dá principalmente por meio de questionários e entrevistas, os quais são feitos por amostragem e representam a população a ser estudada. A generalização dos resultados de uma população é feita após análise de suas amostras (Höhn, 2007).

O *survey* possibilita a obtenção de um grande número de variáveis. Contudo, o mais importante nesse tipo de estudo é ser capaz de levantar um grande volume de valores para uma quantidade menor (e representativa) de variáveis, uma vez que isso facilita o trabalho de análise (Travassos, 2002). Por fim, este tipo de estudo não possui nenhum controle de medição ou de execução sobre suas variáveis (Travassos, 2002).

2. **Estudo de Caso:** Objetiva o acompanhamento de um projeto, atividade ou tarefa (Travassos, 2002). Diferentemente do *survey*, não é um estudo realizado em retrospectiva, mas por meio da observação contínua de um (ou um conjunto limitado de) atributo (s), correlacionando-os. Apesar de não ter controle sobre sua execução (o que dificulta sua replicação), permite controle sobre a medição de variáveis. Contudo, existem fatores<sup>1</sup> de confusão nesse tipo de estudo, ou seja é difícil diferenciar o efeito proveniente de fatores distintos (Travassos, 2002).

As formas de coleta de dados em um estudo de caso podem ser diversas, durante a execução do mesmo. Devido à falta de controle de execução, a validade interna do experimento fica comprometida. Todavia, um estudo experimental permite um contexto rico (Rothermel, 2005) além de freqüentemente ser mais barato (do ponto de vista de recursos e tempo) que um experimento controlado.

---

<sup>1</sup>*Fator* é a denominação dada às variáveis independentes em um experimento. Essa variáveis compõem a entrada do processo de experimentação e representam as causas que determinam os resultados do experimento (Travassos, 2002).

3. **Experimento controlado:** Este tipo de estudo primário é caracterizado pelo controle sistemático das variáveis e do processo. Sendo o estudo mais rigoroso, geralmente é o mais caro de ser praticado. Dentre os objetivos desse tipo de estudo experimental pode-se citar: “confirmação de teorias, confirmar conhecimento convencional, explorar os relacionamentos, avaliar a predição dos modelos ou validar as medidas” (Travassos, 2002). Além disso, o experimento controlado envolve a formulação de uma hipótese que precisa ser verificada.

Os objetos em um experimento controlado são compostos por um conjunto de características (ou seja, as variáveis do experimento). A pesquisa é então planejada para medição dos valores sobre cada característica, os quais serão comparados posteriormente.

A validade interna desse tipo de estudo é garantida pelo rigor com que as condições são controladas. Além disso, permite conclusões sobre causalidade. A generalização dos resultados, porém, pode ser limitada pelo controle aplicado.

A diferenciação entre estudo de caso ou experimento controlado está ligada principalmente ao nível de controle. Quando o alto controle de variáveis é determinante para validação de hipóteses, o experimento controlado mostra-se mais adequado. Se não existe ou não é necessário tal nível de rigor, deve-se considerar a realização de estudo de caso. O nível de dificuldade do experimento, replicação, custo, risco e tempo, são outros fatores determinantes na escolha entre estes tipos de experimentos.

### 3.2.1 Processo de experimentação

Experimentação não é uma tarefa trivial, pois esforço é necessário para preparar, conduzir e analisar os resultados. Para que os objetivos do experimento sejam atendidos e o experimento seja conduzido adequadamente, um processo é necessário. Antes de apresentar o processo de experimentação definido em Wohlin et al. (2000), algumas definições importantes são descritas.

Durante a execução de um experimento controlado é estudada a alteração das variáveis que o compõe. Toda variável que pode ser manipulada ou controlada nesse processo é denominada *variável independente*<sup>2</sup>. O efeito dessa manipulação é observado nas *variáveis dependentes*<sup>3</sup>. As variáveis independentes que são manipuladas, são denominadas de *fatores*. As demais são fixadas para não confundirem a causa do efeito gerado. O valor próprio de um *fator* é denominado *tratamento* e pode ser um método, ferramenta, técnica ou condição que resulte um determinado efeito. O valor próprio de uma variável dependente é denominado *resultado*. Os *participantes* são os indivíduos selecionados dentro da população de interesse, para conduzir o experimento. O *objeto* (ou unidade experimental) por sua vez, será aquilo sobre o qual será aplicado o experimento. O conjunto *objeto, sistema de medição e diretrizes de execução* do experimento compõe a instru-

---

<sup>2</sup>Também conhecida por variável de entrada, ou de estado

<sup>3</sup>Também conhecida por variável de saída, ou de resposta

mentação do experimento (Travassos, 2002). Cada conjunto *tratamento, objeto e participante* por sua vez, constitui um *teste* (ou tentativa) (Höhn, 2007).

Para ilustrar os conceitos apresentados, considere o seguinte exemplo: Suponha que um experimento será preparado para avaliar o custo de aplicação de uma nova técnica de teste comparada a outra já existente. Nesse caso temos:

*Variáveis independentes*: Técnicas de teste, ferramenta e experiência.

*Variáveis dependentes*: Tamanho dos conjuntos de teste adequados, tempo e esforço de aplicação das técnicas.

*Fatores*: As técnicas de teste investigadas.

*Tratamento*: Técnica nova e técnica antiga.

*Resultado*: Valor em quantidade do tamanho dos conjuntos de casos de teste adequados e número de elementos requeridos por cada técnica.

*Participantes*: Pessoas que aplicarão o teste com as técnicas investigadas, sobre os programas em teste.

*Objeto*: Programas em teste.

Em Wohlin et al. (2000) o processo de experimentação é dividido em 5 etapas: definição, planejamento, operação, análise e interpretação, apresentação e empacotamento. É válido lembrar que apesar de serem apresentadas seqüencialmente, durante a condução do experimento as mesmas não se dão necessariamente em cascata mas podem ocorrer iterativamente.

## 1. Definição do Experimento

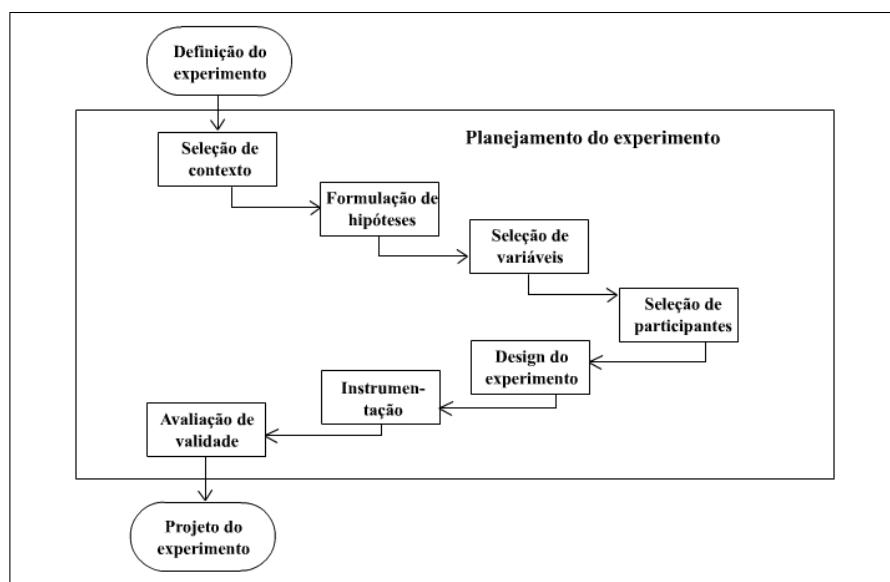
A definição do experimento compõe uma parte vital à experimentação, na qual são definidos o problema, os objetivos, as metas e o contexto do experimento. Isso requer uma visão clara do que se deseja alcançar e da forma como o experimento deve ser conduzido. Desta forma, a hipótese (possível relacionamento causa-efeito a ser verificado) já deve estar estabelecida, permitindo que os objetivos e as metas sejam traçados. Em Wohlin et al. (2000) é apresentado um modelo para descrever as metas:

- Analisar <objeto de estudo>
- Para o propósito de <propósito>
- Com respeito a <enfoque>
- Do ponto de vista da <perspectiva>
- No contexto <contexto>

O objeto do estudo, conforme já definido, é aquilo que será analisado, ou seja, onde será aplicado o tratamento. Exemplo do mesmo são produtos, programas, processos, teorias, modelos etc. O propósito representa o objetivo do experimento existir, ou seja, sua finalidade. O enfoque representa as propriedades alvo, por exemplo, custo, efetividade, eficácia, complexidade, etc. A perspectiva determina o ponto de vista sob o qual a análise e interpretação dos resultados será feita (exemplos: pesquisador, desenvolvedor, consumidor, gerente). Por último, o contexto revela ambiente da condução do experimento, perfil dos participantes e artefatos de software usados para a extração dos dados.

## 2. Planejamento do Experimento

Após definição do experimento, onde as razões e limites do mesmo são determinados, a fase de planejamento caracteriza-se por elaborar a forma como os objetivos traçados serão alcançados. Além disso, os riscos envolvidos com a validação dos resultados são considerados. Compõe-se também como um refinamento da Definição do Experimento. A Figura 3.1 ilustra uma visão geral da seqüência de etapas do planejamento.



**Figura 3.1:** Visão geral da fase de planejamento. Fonte: (Wohlin et al., 2000)

1. **Seleção de contexto** - A seleção de contexto, é um detalhamento do contexto indicado na definição. Desta forma, indica-se o ambiente onde o experimento será realizado e preferencialmente as condições do mesmo, ou seja, explicitação de suas características.
2. **Formulação de hipóteses** - Nesta subfase, definem-se formalmente a hipótese nula e a(s) hipótese(s) alternativa(s). A hipótese nula assume que não há diferença entre os dois tratamentos, com respeito à variável de resposta. A hipótese alternativa apresenta uma posição em que há uma diferença significativa entre os dois tratamentos.

3. **Seleção de variáveis** - Determinam-se as variáveis dependentes e independentes, assim como a escala de medida e os valores que as variáveis podem receber.
4. **Seleção dos participantes** - Os participantes do experimento são escolhidos e identificados. As características dos mesmos devem ser claramente definidas, para que possa ser possível identificar e avaliar o efeito gerado e as diferenças entre eles, em termos dos resultados observados.
5. **Projeto do experimento** - No projeto é elaborado um plano completo sobre as condições experimentais distintas a serem aplicadas pelos participantes, para que seja possível determinar como as condições afetam o resultado obtido na execução da atividade. A elaboração do projeto do experimento depende das variáveis e hipóteses selecionadas. O mesmo pode ser do tipo aleatório, em blocos, balanceado, ou alguma combinação desses.
  - Aleatório: é uma forma de se impedir que fatores indesejáveis e desconhecidos associem-se a determinadas combinações e possam interferir no resultado (Barros Neto et al., 2001). O alvo da aleatoriedade pode ser o próprio objeto, participantes ou ordem em que os mesmos são executados (Wohlin et al., 2000).
  - Em Blocos: é empregado quando um efeito indesejável é conhecido e controlável no resultado, porém não existe interesse no mesmo. Logo, sua finalidade é eliminar sistematicamente este efeito na comparação entre os tratamento. Em um bloco o efeito indesejado é o mesmo, e pode-se estudar o efeito do tratamento nesse bloco (Wohlin et al., 2000). Para tanto, os fatores são distribuídos de maneira a minimizar os efeitos indesejáveis (Neto et al., 2001).
  - Balanceado: os tratamentos são determinados com igual número de participantes. Apesar de não ser imprescindível, o balanceamento simplifica e fortalece as análises estatísticas dos dados (Wohlin et al., 2000).
6. **Instrumentação** - Nesse passo é preparada a implementação prática do experimento como objetos, diretrizes e instrumentos de medida.
7. **Avaliação de Validade** - Na avaliação de validade são levados em conta os riscos que podem comprometer a validade do experimento. Os riscos que podem comprometer a validade de um experimento são fatores além do controle dos experimentadores que podem afetar as variáveis dependentes. Logo, riscos podem ser entendidos como variáveis independentes desconhecidas que geram em adição às hipóteses que estão sendo pesquisadas, hipóteses antagônicas não controladas. Minimizar o impacto desse risco é um passo fundamental no processo de experimentação (Basili et al., 1996). As questões de validade de um experimento podem ser classificadas em cinco tipos: interna, externa, de construção e de conclusão.

- Validade interna: depende da existência de uma relação causal entre tratamento e resultado. Logo, para garantir a mesma, deve ser verificado se não houve a interferência de algum outro fator que não tenha sido medido ou controlado, nessa relação.
- Validade externa: diz respeito à generalização do experimento. Logo, se existe a relação causal entre construção da causa e efeito, deve-se avaliar se o resultado pode ser generalizado para contextos mais amplos ou não.
- Validade de construção: refere-se à relação entre teoria e observação, ou seja, se existe a relação causal entre causa e efeito, deve-se assegurar que o tratamento reflete a construção da causa, e o resultado a construção do efeito.
- Validade de conclusão: corresponde à habilidade de chegar a uma conclusão correta sobre os relacionamentos entre o tratamento e os resultados obtidos no experimento. Essa validade depende de alguns requisitos como a confiabilidade das medidas, da implementação dos tratamentos e o tamanho do conjunto de participantes.

### **3. Operação**

A operação corresponde à fase de aplicação do experimento. É composta de três passos: preparação, execução e validação de dados. No primeiro passo ambiente e material são preparados e os participantes escolhidos. Os participantes são informados da intenção do experimento e seu consentimento é obtido. A execução é o passo em que os participantes realizam as tarefas e os dados são coletados. O último passo consiste em o experimentador validar os dados, determinando se os mesmos são válidos e se foram coletados adequadamente.

### **4. Análise e Interpretação**

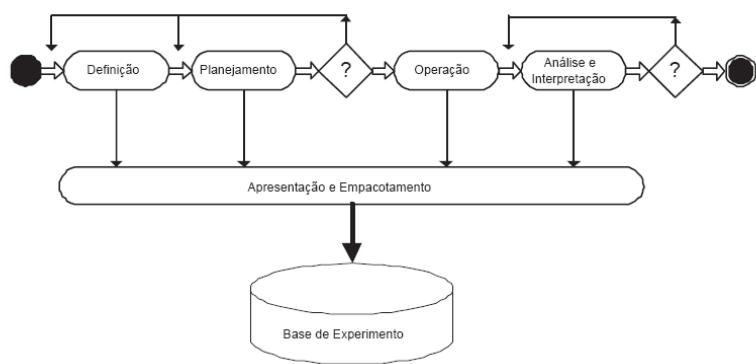
Os dados coletados na etapa “operação” servem de subsídio para execução dessa fase. A primeira parte dessa fase é tentar compreender (vizualizar) os dados, usando para tanto estatísticas descritivas. Na segunda parte é feita a redução do conjunto de dados, eliminando-se dados falsos ou anormais. No terceiro e último passo os dados são usados para avaliar estatisticamente as hipóteses do experimento, tentando-se refutar as hipóteses nulas

### **5. Apresentação e Empacotamento**

O empacotamento corresponde à última atividade do processo de experimentação. A preocupação nessa fase é relacionada à documentação dos resultados e com a repetibilidade. Um pacote de laboratório consiste em uma infra-estrutura experimental que serve para dar suporte a futuras replicações. Dentre as suas funções destacam-se: descrever o experimento em termos específicos, fornecer material para replicação, destacar oportunidades para variação e estabelecer contexto para combinação de resultados de diferentes tipos de tratamentos experimentais (Shull et al., 2004).

Sendo assim, pode-se dizer que os pacotes de laboratório servem como base para confirmar ou rejeitar os resultados do experimento original, complementá-lo ou adaptar o objeto de estudo para um contexto específico.

A fase de apresentação e empacotamento é tradicionalmente vista como a última etapa no processo de experimentação. Entretanto, essa atividade pode ser iterativa, ou seja, desenvolvida ao longo do processo de experimentação. A Figura 3.2 ilustra o modelo de processo decorrente dessa proposta. Conforme pode ser observado, cada etapa do processo gera informações que podem ser aproveitadas para o empacotamento do experimento, sem a necessidade de que o processo seja completado para que as mesmas sejam coletadas.



**Figura 3.2:** Empacotamento ao longo do processo de experimentação (Amaral, 2003).

Visando auxiliar as atividades de empacotamento e replicação de experimento, uma ferramenta denominada COTEST foi desenvolvida Felizardo (2003). Essa ferramenta fornece algumas funcionalidades como cadastramento, seleção dos participantes e elaboração do projeto experimental. Além disso, acompanha o participante passo-a-passo durante a condução do experimento. O registro dos resultados é feito *on-line* e todos os formulários e documentos necessários são disponibilizados pela própria ferramenta.

Apesar da evidente necessidade de empacotamento do experimento para sua replicação, possibilitando o aumento do conhecimento acerca dos conceitos investigados e a calibração das características do experimento, o maior gargalo dessa atividade consiste na falta de normas para padronização das informações. Isso leva a uma série de discordâncias como: a interpretação dos conceitos relevantes ao empacotamento, o seu conteúdo ótimo e os meios da apresentação do pacote (Travassos, 2002). A atividade de documentação (registro) do experimento por exemplo, apresenta uma grande heterogeneidade de estilos de relato, o que leva ao problema de encontrar informações relevantes, ausência de informações importantes e, por consequência, a dificuldade de integração dos resultados de vários estudos em um mesmo corpo de conhecimento. Jedlitschka (2005) apresentam uma proposta preliminar de padronização para relatos de experimentos controlados, baseada na unificação das diversas diretrizes publicadas na literatura. Para tanto, um modelo de relatório para experimentos controlados também é apresentado, orientando detalhadamente como as diversas seções e subseções devem ser preenchidas.

### 3.3 Trabalhos Relacionados

Um dos desafios da área de teste de software é determinar, frente à diversidade de técnicas e critérios de teste existentes, qual a melhor forma de utilizá-los de forma a reduzir custos e maximizar os resultados obtidos.

A comparação entre os critérios pode ser feita sob duas perspectivas: a teórica e a experimental. Na primeira, o estudo é analítico e portanto, baseia-se em característica e propriedades como relação de inclusão entre critérios. Na segunda, o estudo é probabilístico e portanto, dados e estatísticas sobre a aplicação desses critérios são coletados, dos quais são inferidas conclusões, como a freqüência na qual distintas estratégias de teste revelam erros ou o custo de aplicação e relação entre as técnicas. Essas conclusões fornecem subsídios para a escolha do critério a ser utilizado em um determinado contexto (Howden, 1978; Weyuker, 1990).

De acordo com Weyuker et al. (1991), o uso de estudos teóricos para escolha de técnicas e critérios de teste, apesar de ser uma das formas mais usuais, tem sido questionada por prover informações de valor limitado em algumas situações práticas. O uso de informações probabilísticas obtidas a partir de experimentos por sua vez, contornaria algumas dessas limitações. Já em Bertolino (2004) é apontada a complementaridade entre esses estudos. Os estudos analíticos serviriam para levantar algumas evidências sobre em quais condições uma técnica de teste seria melhor do que outra e os resultados dos experimentos serviriam para mostrar quando (e em qual contexto) tais condições são satisfeitas. Briand (2007) fornece uma visão completa sobre o assunto. O mesmo confirma as afirmações anteriores na qual uma comparação analítica não é suficiente para obter ou comparar o custo-efetividade de várias técnicas de teste, sendo a realização de estudos experimentais a saída para o mesmo. Esses estudos, entretanto, devem responder algumas questões, tais como (Briand, 2007):

- Quais taxas de custo e de detecção de defeitos podem ser esperados com o uso da técnica de teste em um determinado contexto?
- Como comparar técnicas de teste alternativas em termos de custo e eficácia em revelar erros?
- É benéfico combinar duas ou mais técnicas de teste? Essas técnicas são complementares em relação a eficácia em revelar erros.

Em Mathur e Wong (1994), o estudo experimental sobre critérios de teste é motivado pela necessidade prática de se decidir se um programa foi suficientemente testado, ou seja, “dado um conjunto de teste  $C_1$ -adequado para um programa  $P$ , é possível melhorar esse conjunto de teste usando um outro critério  $C_2$ ?” Além dessa, outras questões que incentivam a condução de estudos experimentais são: decidir, dados dois critérios, qual é mais difícil de ser satisfeito; determinar o custo ou eficácia de um critério; reduzir custos de um critério sem comprometer a capacidade do mesmo em revelar erros (Souza et al., 2007).

Tratando-se de estudos experimentais, três características principais são usadas para comparação dos critérios: custo, eficácia e dificuldade de satisfação (*strength*). O custo corresponde ao esforço para aplicar um critério. Em geral é medido pelo tamanho do conjunto de teste ou por outras métricas relacionadas ao critério como número de elementos requeridos gerados e tempo gasto para identificar mutantes equivalentes e caminhos não-executáveis. A eficácia equivale à capacidade de um critério detectar maior número de erros em relação a outro. A dificuldade de satisfação diz respeito à probabilidade de um conjunto de teste adequado a um critério satisfazer outro (Wong, 1993).

Nos estudos experimentais os custos de teste geralmente são medidos de acordo com o tamanho do conjunto de teste. Briand (2007) faz uma ressalva a esse respeito. Quando os custos não se relacionam somente à avaliação dos critério mas à atividade de teste como um todo, devem ser observados dois pontos de vista: esforço humano e tempo de execução. Seguindo essa ótica, os custos que a princípio parecem claramente estarem envolvidos apenas com a execução dos testes e possível construção de *stubs* e *drivers*, estendem-se à derivação de um modelo ou dos requisitos de teste, a qual raramente é totalmente automatizada. Weyuker et al. (1991) faz uma observação a esse respeito, na qual durante a realização de um estudo experimental pôde-se observar que seres humanos são freqüentemente muito bons em reconhecer a não-executabilidade, uma tarefa que pode se tornar pouco trivial quando repassada a algoritmos que prestem-se a este fim. Briand (2007) discute que a implementação de oráculos e a instrumentação sobre código fonte ou binário, adicionam custos à atividade de teste como um todo e que por mais que a medida do tamanho do conjunto de teste sirva como parâmetro de comparação dentro de uma família de critérios (já que os outros fatores de custo apontados seriam em suma equivalentes ou reaproveitáveis) isso não é válido comparando-se técnicas, já que os esforços em se derivar um modelo de *statecharts*, por exemplo, seriam muito maiores do que para um grafo de fluxo de controle, por exemplo.

Além da agregação de custos de automatização e de derivação de modelos que a atividade de teste pode implicar, deve-se considerar também as propriedades que são inerentes ao software como a presença de concorrência, o fato de ser ou não distribuído e a manipulação complexa de exceções. “Essas propriedades definem não só os tipos de falha que o software pode ter, mas o quanto difícil seriam de serem detectadas”(Briand, 2007).

Uma outra maneira, menos formal, porém mais abrangente de se comparar critérios baseada nos custos, seria avaliando a dificuldade de se determinar quando um determinado critério é satisfeito. Weyuker (1989) discute medidas quanto à “facilidade de uso” de um critério. Nesta abordagem são consideradas a quantidade de teste requerido e a quantidade de trabalho necessário para decidir se um critério foi satisfeito, no qual o último seria o mais dispendioso.

Weyuker et al. (1991) apresentam algumas definições importantes acerca do que é a avaliação de efetividade. “A efetividade de um critério não está relacionado com a quantidade de defeitos encontrados, mas à quantidade de defeitos remanescentes”. Isso é uma forma de se evitar uma interpretação deturpada acerca do critério em um contexto onde um programa cheio de falhas ainda não testado, serviria de indicador para a efetividade do critério aplicado. De certa forma isso se

relaciona a outra afirmação feita de que “na atividade de teste o interesse é aumentar a confiabilidade do software descobrindo-se defeitos que tenham efeito significante sobre a confiabilidade”. Afinal, se os defeitos presentes são simples e não afetam a confiabilidade do software, certamente não justificaria esforços em se descobrir a quantidade restante dos mesmos (não tanto quanto em um cenário oposto).

Outra definição importante a ser considerada é a de efetividade para critérios de seleção ou de adequação. Weyuker et al. (1991) afirmam que “um critério de seleção só é efetivo se ele detecta a maioria dos defeitos possíveis” enquanto que um “critério de adequação só é efetivo se ele permite que o teste encerre somente quando poucas falhas remanescerem não detectadas”. Isso condiz com a própria classificação de cada tipo de critério. Se é um critério de seleção espera-se que o mesmo selecione um bom conjunto de teste de forma a maximizar o número de falhas encontradas. Um critério de adequação por sua vez não resolve tendo sido satisfeito caso exista uma grande quantidade de falhas no objeto sob teste, ou o que é pior, se as mesmas são falhas graves.

### **Estudos Experimentais Envolvendo Teste Estrutural e Mutantes**

Weyuker (1990) faz uma análise comparativa de critérios estruturais de fluxo de dados (Todos-Usos, Todos-P-Usos, Todos-C-Usos, Todos-DU-Usos). Para o mesmo, experimentos sobre programas reais foram adotados. Quatro métricas para avaliação de custo de adequação aos critérios em relação à expectativa teórica foram usadas. A avaliação dos quatro critérios sob essas métricas permitiu verificar algumas propriedades como: a relação entre o tamanho do conjunto de teste é linearmente proporcional ao tamanho do programa; para o critério Todos-C-Usos, o número de casos de teste suficiente para cobrá-lo é em geral, metade do número de decisões contidas no programa; e por mais que exista uma discrepância teórica grande entre a complexidade de dois critérios como Todos-C-Usos e Todos-Caminhos-DU, os tamanhos dos conjuntos de teste para satisfazer os mesmos em média não são tão discrepantes. Entre o critério menos complexo Todos-C-Usos e o critério mais complexo Todos-Caminhos-DU por exemplo, as métrica indicam que o último requer em média o dobro, enquanto no pior caso o seu limite superior de complexidade é exponencial e o do primeiro é apenas quadrático.

Weyuker et al. (1991) apresentam um estudo comparando variações dos critérios de fluxo de controle Todos-Nós e Todas-Arestas. Neste, uma relação denominada PROBBETER é usada para mostrar quando um conjunto de teste aleatório adequado a um critério  $C_1$  é mais apto a detectar falhas do que um conjunto de teste aleatório adequado a um critério  $C_2$ . Dessa forma, admite-se  $C_1$  como um critério variante do critério Todos-Nós, e  $C_2$  um variante do critério Todas-Arestas. A diferença dos critérios originais com os variantes é que neste último caso os critérios possuem uma variável  $K$  que determina um valor fixo para o tamanho dos conjuntos de teste admissíveis. A relação de inclusão entre os critérios é mantida ( $C_2$  inclui  $C_1$ ). O intuito da demonstração é mostrar que é possível descobrir um programa  $P$  onde o número proporcional de falhas encontradas

pelo critério inferior na relação de compreensão, ou seja,  $C1$  é maior do que o do critério superior ou seja,  $C2$ . Para tanto, define-se um programa simples em linguagem C; delimita-se o domínio de entrada ao intervalo  $[1, 4m]$ , onde  $m$  é um valor inteiro qualquer; define-se uma fórmula geral para determinar o número de casos de teste adequados ao critério  $C1$  em função do tamanho do domínio de entrada (em função de  $m$ ); define-se uma fórmula geral para determinar o número de casos de teste adequados ao critério  $C2$  em função do tamanho do domínio de entrada (em função de  $m$ ); define-se uma fórmula geral para determinar o número de casos de teste adequados ao critério  $C1$  que irão expor falhas e uma fórmula geral para determinar o número de casos de teste adequados ao critério  $C2$  que irão expor falhas. Por fim, uma fórmula é definida para calcular a probabilidade de se encontrar falhas a partir de um conjunto de teste aleatório adequado ao critério Todos-Nós e outra para calcular a probabilidade de se encontrar falhas a partir de um conjunto de teste aleatório adequado ao critério Todas-Arestas. Os resultados da aplicação das fórmulas para três domínios de entradas de tamanhos diferentes mostraram que nos três casos, a probabilidade de se encontrar defeitos usando o critério  $C1$  era maior do que a probabilidade de encontrar falhas usando o critério  $C2$ . Isso se deve, segundo a autora à incapacidade de os casos de teste adicionais requeridos pelo critério  $C2$  não falharem. Isso ilustra claramente a sensibilidade do contexto na comparação entre critérios, o que demonstra que as comparações probabilísticas são capazes de “enxergar” lacunas imperceptíveis do ponto de vista analítico.

Alguns resultados com a aplicação de experimentos avaliando critérios de teste estruturais, baseados em erro foram identificados em Maldonado et al. (2007):

Com o *benchmark* de programas estabelecidos em Weyuker (1990) para avaliação de critérios da família fluxo de dados, outros trabalhos experimentais foram desenvolvidos avaliando os critérios Potenciais-Usos (Maldonado et al., 1992; Vergílio et al., 1992). Nesses trabalhos alguns resultados relevantes puderam ser obtidos como a viabilidade de aplicação prática dos critérios dessa família com conjuntos de teste relativamente pequenos.

Em Mathur e Wong (1993) um estudo empírico é feito comparando os critérios de mutação aleatória (10% de cada operador de mutação selecionado) e mutação restrita (um subconjunto de operadores de mutação selecionados). O objetivo do estudo era comparar custo-benefício. Os resultados obtidos mostraram que ambos os critérios mostraram-se igualmente eficazes e com um custo baixo (sem perdas de eficácia consideráveis).

Mathur e Wong (1994) compararam o *strength* e custo entre os critérios análises de mutantes e Todos-Usos. Os resultados obtidos revelaram que os conjuntos de teste adequados ao critério análise de mutantes foram também adequados ao critério de fluxo de dados. Contudo, o inverso não mostrou-se verdadeiro em muitos casos, de onde conclui-se que na prática, o critério análise de mutantes inclui Todos-Usos.

Wong et al. (1995) realizaram um estudo comparando custo e eficácia da mutação restrita e mutação aleatória (10%) em relação ao critério Todos-Usos. Os resultados obtidos forneceram evidências de que o custo de aplicação é decrescente na ordem: Todos-Usos, mutação aleatória e mutação restrita. Em relação à eficácia observou-se que a ordem do mais eficaz para o menos eficaz

foi: mutação restrita, Todos-Usos e mutação aleatória. Esses resultados mostraram que do ponto de vista prático, pode ser útil para o testador, restringido pelo prazo de entrega do produto, aplicar o critério análise de mutantes somente em partes mais críticas do software, usando para o restante opções mais econômicas como mutação restrita ou o critério Todos-Usos, sem comprometer a qualidade da atividade.

Os resultados obtidos no trabalho Offutt et al. (1996b), confirmam as conclusões obtidas em Wong et al. (1995) e Mathur e Wong (1994). O objetivo do experimento foi comparar o critério análise de mutantes com Todos-Usos. Da mesma forma, o critério análise de mutantes mostrou-se mais eficaz em revelar erros do que o Todos-Usos e o *strength* para satisfação do critério análise de mutantes foi maior que o do critério Todos-Usos.

Em Wong et al. (1994) e Souza (1996) experimentos foram realizados usando a ferramenta *Proteum*, para comparar seis classes de mutação restrita, quanto à eficácia. Os resultados permitiram detectar as classes com a menor relação custo-eficácia. Isso permitiu estabelecer uma estratégia incremental, para cenários onde o prazo para realização de teste fosse restrito, no qual os conjuntos de casos de teste fossem inicialmente adequados à classe de mutação mais econômica e à medida que as restrições de custo permitissem, fossem melhorados para atender às classes de mutação com maior relação custo-eficácia.

Souza et al. (1997), compararam o *strength* e custo dos critérios Análises de mutantes e Potenciais-Usos de onde obtiveram-se os seguintes resultados: o custo (em termos do número de casos de teste requeridos) para satisfazer o critério análise de mutantes foi maior do que para satisfazer os critérios Pontenciais-Usos. Em relação ao *strength*, os critérios Análise de Mutantes e Todos-Pontenciais-Usos mostraram-se incomparáveis, mesmo do ponto de vista prático.

Barbosa et al. (2000) mostram o resultado da condução de uma série de experimentos realizado com a ferramenta *Proteum* para determinar um conjunto de operadores essenciais para aplicação do critério Análise de Mutantes em linguagem C. Esses operadores essenciais são um subconjunto de operadores que têm uma eficácia quase tão boa quanto o conjunto total de operadores disponíveis para a linguagem, mas que geram um custo menor para a aplicação dos testes. O resultado mostrou que dos 71 operadores disponíveis, 8 são suficientes para garantir um escore de mutação próximo a 1.

No contexto de mutação de interface, os estudos experimentais apresentados em Delamaro (1997) e Vincenzi et al. (1999) investigaram a relação entre os critérios análise de mutantes e mutação de interface. Os resultados revelaram que o critério mutação de interface apresenta alta eficácia em revelar erros, porém com um alto custo de aplicação, em decorrência do número de mutantes gerados (à semelhança do critério análise de mutantes). Vincenzi et al. (1999) investigaram uma estratégia de teste para aplicar de forma complementar os critérios Análise de Mutantes e Mutação de Interface visando obter um baixo custo e alto grau de adequação em relação aos critérios. Os resultados obtidos mostraram que é possível obter conjuntos de teste adequados ou muito próximos da adequação para ambos os critérios, mesmo com um número reduzido de operadores.

Em Maldonado et al. (2000a), um experimento aplicando mutação seletiva no teste de programas em C tanto em nível de unidade quanto de integração mostrou, para o conjunto de programas utilizados, que mesmo com uma redução em torno de 80% do número de mutantes gerados, ainda é possível obter escores de mutação próximos a 1, o que promove o uso da mutação seletiva no teste de mutação.

### **Estudos Experimentais Envolvendo Teste Funcional**

Em Basili e Selby (1987) um experimento é conduzido comparando-se três técnicas (leitura de código, teste funcional e estrutural) quanto à efetividade e eficiência em revelar falhas. O objeto do experimento foi composto por três programas escritos em Fortran e envolveu como participantes 42 estudantes com nível avançado de conhecimento e 32 desenvolvedores profissionais, sendo realizado duas vezes no primeiro grupo e uma vez no segundo. Na última replicação, os participantes receberam um treinamento na forma de tutorial de quatro horas sobre as técnicas. Os resultados obtidos demonstraram que a técnica de leitura de código obteve o maior percentual de inconsistência detectada. Além disso, os participantes revelaram mais erros usando a técnica de teste funcional do que a estrutural. Em média, foram observadas apenas 50% das falhas.

Em Kamsties e Lott (1995) outro experimento foi conduzido comparando-se leitura de código, teste funcional e estrutural quanto à efetividade e eficiência em observar inconsistências/falhas e isoler defeitos em programas em C. O experimento foi realizado duas vezes, com 27 participantes na primeira vez e 23 na segunda. O experimento utilizou a variação aleatória sobre técnicas, programas, participantes e ordem de aplicação das técnicas. Além disso, os defeitos deveriam ser isolados após observação das falhas, com a finalidade de localizar no código fonte o local exato responsável pela falha. Os resultados obtidos revelaram que o percentual de falhas detectadas nas técnicas de leitura de código e teste funcional foi o mesmo e que este foi superior ao percentual de falhas detectadas com a técnica estrutural. Além disso, o emprego do teste funcional mostrou-se mais eficiente no isolamento de erros do que as outras técnicas. Novamente, em média 50% das falhas foram detectadas.

Em cima dos trabalhos Basili e Selby (1987) e Kamsties e Lott (1995), o estudo experimental Wood et al. (1997) foi conduzido, inclusive aproveitando os programas disponíveis no pacote de replicação do experimento realizado em (Kamsties e Lott, 1995). Na técnica de teste estrutural, entretanto, foi aplicado somente o critério que cobre 100% das instruções do programa. O objetivo do estudo era comparar as mesmas três técnicas (leitura de código, funcional e estrutural) quanto à efetividade e eficiência em revelar falhas. Os resultados mostraram que analisadas isoladamente, as três técnicas apresentavam basicamente as mesmas taxas de efetividade quanto à observação de falhas e isolamento de defeitos. As combinações das técnicas, todavia, permitiu observar até nos piores casos taxas de efetividade superiores àquelas obtidas por cada uma das técnicas isoladamente. Em média, a taxa de efetividade das técnicas combinadas foi 25% maior do que das técnicas isoladas.

No trabalho Dória (2001), as mesmas técnicas (Leitura de Código, Teste Funcional e Teste estrutural) e programas empregados em Basili e Selby (1987) Kamsties e Lott (1995) e Wood et al. (1997) foram usados. Neste contudo, a estratégia incremental foi empregada. O objetivo do experimento foi comparar efetividade e eficiência em observar inconsistências/falhas e isolar defeitos. Para tanto, um conjunto de 12 estudantes foram escolhidos como participantes. A condução do teste incremental compreendeu os critérios Todos-Nós, Todas-Arestas, Todos-Usos, Todos-Potenciais-Usos e Análise de Mutantes. Os participantes inicialmente construiram conjuntos de teste adequados ao critério Todos-Nós, usando a ferramenta *Poke-Tool*, ou seja, construíram um conjunto de teste, executavam-no na ferramenta e enquanto a cobertura não atingisse 100% ou acreditasse-se que o mesmo não pudesse ser melhorado em decorrência da presença de caminhos não executáveis, novos casos de teste eram adicionados. Após isso, as falhas foram identificadas comparando-se as informações da especificação com os resultados de teste obtidos. Desta forma, os participantes puderam isolar no código fonte os defeitos que causaram tais falhas. Após isso, o conjunto de teste adequado ao critério Todos-Nós foi reaplicado no critério Todas-Arestas, no qual novos casos de teste eram inseridos caso necessário, para adequação a esse critério. Esse processo foi repetido sucessivamente para os critérios Todos-Usos e Todos-Potenciais-Usos. Após um tempo pré-estabelecido no planejamento do experimento, os caminhos e elementos não-executáveis foram entregues aos participantes, já que estes poderiam não identificá-los e ficar tentando indefinidamente atingir sem sucesso a taxa de cobertura de 100%. Por fim, o último conjunto de teste gerado foi reutilizado para aplicação do critério análise de mutantes (usando-se a ferramenta *Proteum*) no qual foram inseridos novos casos de teste até ser obtido escore de mutação igual a 1. Novamente, as falhas detectadas usando-se a especificação e o resultado do teste de mutação, permitiram o isolamento dos defeitos responsáveis pelas mesmas no código. A relação de mutantes equivalentes foi igualmente fornecidas aos participantes, já que a presença dos mesmos inviabiliza a obtenção de escore de mutação igual a 1. Alguns resultados relevantes obtidos foram: a técnica de teste funcional obteve melhor resultado em revelar falhas e isolar defeitos do que as outras técnicas. Os resultados obtidos nos piores casos, por qualquer combinação de técnica, foram sempre superiores do que os resultados obtidos nos piores casos, para qualquer técnica isolada.

Um estudo experimental comparando teste aleatório, funcional e análise de mutantes, foi realizado em Linkman et al. (2003). O objeto do experimento foi o programa Cal do Unix. O experimento contou com seis participantes, os quais já possuíam conhecimento sobre os critérios e programa usado. O objetivo do experimento foi avaliar a adequação do teste aleatório e funcional em relação ao critério análise de mutantes. Para o teste aleatório, foram selecionados os sete conjuntos de teste gerados em Wong (1993). Para o teste funcional sistemático, um dos participantes gerou, com base na especificação, 76 casos de teste. Com base nos critérios funcionais particionamento de equivalência e análise do valor limite, mais quatro conjuntos de teste foram gerados. O critério análise de mutantes foi usado para avaliar a eficácia dos conjuntos de teste gerados em revelar defeitos. Com base em todos os operadores de mutação para programas em C, foram gerados

4.624 mutantes, dos quais 335 foram identificados equivalentes. Os resultados obtidos no estudo revelaram que apenas o conjunto de teste derivados para o teste funcional sistemático, foi capaz de matar todos os mutantes. Apenas um conjunto de teste gerado para o teste funcional e um conjunto de teste selecionado para o teste aleatório, obtiveram escore de mutação acima de 0,98. Em geral, os escores de mutação atingidos pelos conjuntos gerados para o critério funcional, mostraram-se maiores do que os gerados para o teste aleatório. Os resultados permitiram concluir que, ainda que sejam necessários estudos de casos para obter-se validade estatística, os dados obtidos fornecem indícios de que os testes funcionais podem obter um alto grau de cobertura, se aplicados corretamente.

### 3.3.1 Comparação de Estudos Experimentais Correlacionados

Um trabalho que reuniu e avaliou resultados obtidos em 25 anos de estudos experimentais em teste de software, foi realizado por Juristo et al. (2004). Para avaliar os estudos, as autoras compararam a forma com que os experimentos foram conduzidos, seus objetivos e resultados obtidos. Para tanto, foram selecionados estudos que abordam teste aleatório, teste funcional, teste estrutural, teste de mutação, teste de regressão ou que apliquem alguma otimização sobre a construção de conjuntos de teste. Alguns resultados obtidos com essas comparações e que são relevantes ao tema deste trabalho são considerados:

A primeira comparação realizada foi entre os trabalhos Weyuker (1990) e Bieman e Schultz (1992), comparando critérios de fluxo de dados entre si, dos quais algumas conclusões relevantes feitas pelas autoras são: o critério Todos-P-Usos deveria ter sido usado nesses experimentos no lugar do critério Todos-Usos e o critério Todos-Usos no lugar do critério Todos-DU-Caminhos, uma vez que esses critérios geram menos casos de teste e em geral cobrem os casos de teste gerados pelos outros critérios. Tanto em Weyuker (1990) quanto em Bieman e Schultz (1992) existe uma concordância (contrária à teoria) quanto à aplicabilidade prática do critério Todos-DU-Caminhos. Algumas críticas importantes a respeito desses estudos foram: a variável de resposta dos experimentos consistiu basicamente do número de caso de teste gerados, o qual segundo as autoras é insuficiente para os propósitos de uma técnica de teste e deveria ter sido complementado por um estudo de efetividade; a relação do número de casos de teste gerados para o critério Todos-DU-Caminhos com a estrutura do programa deve ser examinada com mais detalhes, uma vez que em um dos estudo é dito que o número de casos de teste está relacionado ao número de comandos de decisão existentes no programa e no outro estudo, ao número de linhas de código, sem que nenhum dos estudos entretanto, estabeleça uma relação entre essas propriedades.

A segunda comparação foi realizada entre os trabalhos Offutt e Lee (1994), Offutt et al. (1996a) e Wong e Mathur (1995). Algumas conclusões relevantes feitas pelas autoras são: o teste de mutação tradicional parece ser mais efetivo em encontrar erros, porém mais caro. Para sistemas não-críticos, as variantes da mutação tradicional (mais baratas e um pouco menos efetivas) poderiam ser usadas. Uma crítica relevante aos trabalhos considerados foi a de que seria interessante

comparar as variantes da mutação tradicional entre si e não apenas em relação à mutação padrão, para verificar quais são as mais vantajosas quanto ao custo e performance.

A terceira comparação feita, foi em relação ao teste de regressão, abordado nos trabalhos: Rothermel e Harrold (1998), Bible et al. (1999), Vokolos e Frankl (1998), Kim et al. (2000), Graves et al. (1998). Algumas observações importantes acerca dos resultados obtidos por esses estudos são: se os conjuntos de teste são pequenos, então é recomendado aplicar o reteste total (todos os casos de teste são retestados). Se os conjuntos de teste são grandes, então é recomendado aplicar técnicas seguras como *deja-vu* ou *test-tube*<sup>4</sup>. Uma crítica importante sobre a comparação desses estudos foi de que apesar de algum experimentos coincidirem quanto às técnicas e variáveis de respostas usadas, isso não é válido para todos, o que torna difícil obter conclusões comparáveis sobre esses estudos.

A quarta comparação de estudos empíricos é feita sobre teste estrutural baseado em fluxo de controle, fluxo de dados e teste aleatório. Os trabalhos analisados foram: Frankl e Weiss (1993), Hutchins et al. (1994) e Frankl e Iakounenko (1998). Algumas conclusões importantes a respeito dessa comparação foram: não houve, estatisticamente, diferenças significativas quanto à eficácia obtida entre essas técnicas. Em situações de restrição de tempo, o teste aleatório pode ser aplicado com uma expectativa de ser similar ao critério Todos-Usos em 50% dos casos. Até mesmo quando a cobertura máxima é atingida, não há garantias de que o defeito será encontrado, o que indica que as técnicas podem ser sensíveis a determinados tipos de falhas. Uma crítica importante a respeito desses trabalhos foram: a avaliação da efetividade quanto à probabilidade de se encontrar pelo menos um defeito no programa não é muito útil do ponto de vista prático. O número de defeitos detectados em relação ao total presente seriam medidas mais atrativas.

A quinta comparação é realizada sobre os trabalhos Frankl et al. (1997) e Wong e Mathur (1995) os quais comparam análise de mutantes com critérios de fluxo de dados. Algumas conclusões relevantes sobre esses trabalhos são: de forma geral, o critério análise de mutantes mostra-se tanto quanto ou mais efetivo em encontrar falhas que o critério Todos-Usos, porém mais caro. Uma crítica importante quanto a esses estudos é que a porcentagem de conjuntos de teste que revelam pelo menos um defeito, não é uma variável de resposta significativa do ponto de vista prático.

A sexta e última comparação relevante ao escopo deste trabalho é realizada sobre os trabalhos Myers (1978), Basili e Selby (1987), Kamsties e Lott (1995) e Wood et al. (1997) que comparam critérios da técnica estrutural com critérios da técnica funcional. Algumas conclusões relevantes sobre esses trabalhos são: diferenças quanto à efetividade foram encontradas entre as técnicas funcional e estrutural, dependendo do tipo de programa em que eram aplicadas, nos experimentos realizados por Basili e Selby (1987), Kamsties e Lott (1995) e Wood et al. (1997). Além disso, Wood et al. (1997), em uma análise mais profunda, fornece indícios de que a revelação do erro é influenciada pelo tipo de falha que esses defeitos causam no programa; mais defeitos são detectados pela mesma técnica quando os participantes trabalham em grupos ao invés de isoladamente.

---

<sup>4</sup>mais informações sobre as técnicas *deja-vu* e *test-tube* podem ser encontradas em Rothermel e Harrold (1997) e Rosenblum e Weyuker. (1997)

Uma crítica importante realizada sobre os mesmos trabalhos é a de que apesar de serem similares, é necessário ter cuidado ao comparar diretamente os resultados, já que as técnicas empregadas não são absolutamente idênticas.

### 3.3.2 Aspectos Relacionados à Validade Externa

A validade externa diz respeito à representatividade dos resultados obtidos com um experimento que permite a sua generalização. Bertolino (2004) identifica três fatores que são empecilhos na validade externa dos experimentos. O primeiro deles seria a escalabilidade já que os experimentos não podem representar toda a classe de artefatos testáveis dentro de sua categoria (programas, especificação etc.). O segundo fator seria o contexto, vital, pois as condições controladas e ideais propiciadas no ambiente de laboratório certamente não condizem com a aleatoriedade, pressão, restrições e limites de tempo impostos pelo mundo real. O terceiro e último fator seria a bagagem e cultura do testador, uma vez que fatores humanos são extremamente relevantes para os resultados obtidos com o experimento. A disponibilidade de profissionais com tempo livre e aptos a se submeterem a realização de experimentos é escassa.

A variação aleatória (ou randômica) envolvida em experimentação é outra questão que pode interferir nos resultados obtidos. Isso porque um mesmo critério pode ser coberto por vários conjuntos de teste distintos, o que implica que a taxa de custo e efetividade na revelação de erros desse critério está sujeita a algum nível de variação aleatória. Em se tratando de experimentos controlados, essa questão é tratada usando-se uma quantidade suficientemente grande de participantes ou grupos de participantes. O gargalo dessa proposta está entretanto em determinar a quantidade de participantes ou grupos necessários e o que fazer quando alguém se encontra limitado pela disponibilidade de recursos (Briand, 2007).

A geração aleatória de casos de teste é um recurso bastante utilizado por facilitar a automatização e gerar conjuntos de teste grandes a custos reduzidos. Além disso, é uma forma de se evitar que o conhecimento prévio do testador acerca dos programas a serem testados, influencie na geração dos casos de teste e com isso favoreça a adequação a algum critério específico, mascarando o objetivo principal, ou seja, revelar erros. Desta forma, o testador fica incumbido de apenas complementar manualmente o conjunto de teste, caso o conjunto gerado não esteja adequado a algum critério analisado.

## 3.4 Considerações Finais

Neste capítulo foram apresentados os fundamentos e conceitos básicos da área de engenharia de software experimental. Assim, os principais tipos de estudos considerados pela literatura foram descritos, assim como a motivação para aplicação dos mesmos. Após a apresentação dos tipos de estudos, descreveu-se cada etapa do processo de experimentação, adotando-se por referência

o processo de experimentos controlados. Para tanto, os termos próprios envolvidos foram previamente explicados. O processo de experimentação descrito resulta ao final, na construção de um pacote de laboratório, o qual consiste em um dos objetivos deste trabalho.

Além da teoria sobre engenharia de software experimental, apresentou-se neste capítulo os principais trabalhos encontrados sobre experimentos relacionados a técnicas e critérios de teste. Assim foram abordados desde trabalhos que fornecessem um panorama da aplicação de estudos experimentais na área de teste, até trabalhos mais específicos, que apresentassem resultados de estudos experimentais realizados, comparando custo ou eficácia entre técnicas ou critérios de teste.

# Definição e Planejamento do Estudo

## 4.1 Considerações Iniciais

Este capítulo descreve a definição e o planejamento do estudo. O planejamento é composto por um plano de trabalho que define as etapas das tarefas a serem realizadas, a maneira com que cada tarefa deve ser executada, os artefatos envolvidos e os subprodutos a serem gerados. Por se tratar de um estudo de caracterização e avaliação, procurou-se seguir na elaboração do planejamento desta investigação, práticas que outrora tenham sido bem sucedidos e aplicados em estudos do mesmo tipo dentro da computação. Para tanto, alguns conceitos e partes do processo aplicado a experimentos controlados foram importados e adequados ao escopo de pesquisa, na medida em que mostraram-se apropriados para suprir as necessidade levantadas.

## 4.2 Definição do Estudo

A definição do estudo seguiu o *template Goal Question Metric* (Briand et al., 1996) e é apresentado a seguir.

### 4.2.1 Definição de Metas

- **Objeto de Estudo:** Os objetos de estudo são os critérios de teste funcional e estrutural aplicados no paradigma OO e os critérios de teste estrutural e funcional aplicados no paradigma procedural.

- **Propósito:** O propósito é caracterizar e avaliar os critérios, em particular com respeito às diferenças entre os paradigmas.
- **Foco de Qualidade:** O foco de qualidade avaliado é o custo e a dificuldade de satisfação dos critérios nos dois paradigmas.
- **Perspectiva:** A perspectiva do experimento é do ponto de vista de pesquisadores.
- **Contexto:** A investigação será conduzida pelo próprio pesquisador que definiu o estudo, sobre um conjunto de programas de estrutura de dados. O estudo é conduzido na forma de um estudo de variação Multi-Objeto, ou seja, um participante operando sobre um conjunto de objetos<sup>1</sup>.

A sumarização da definição é apresentada a seguir:

**Analisar** “critérios de teste funcional e estrutural aplicados no paradigma OO” e “critérios de teste funcional e estrutural aplicados no paradigma procedural”

**Para o propósito de** caracterização e avaliação

**Do ponto de vista do** pesquisador

**Com respeito a** custo e dificuldade de satisfação strength

**No contexto de** estudante de mestrado testando programas de estrutura de dados

É importante ressaltar que o “custo” definido no **Foco de Qualidade** da definição de metas, refere-se ao custo dos critérios de teste e não da atividade de teste como um todo. A diferenciação é importante por ter impacto sobre outros pontos do estudo como as métricas usadas e a forma de avaliar a validade do estudo. Essas questões são tratadas em detalhes na seção seguinte.

## 4.3 Planejamento

Uma vez descrita a definição do estudo, o plano pode ser elaborado. O plano estabelece as hipóteses e variáveis do estudo e serve como roteiro para a condução e análise do assunto investigado.

---

<sup>1</sup>Os objetos manipulados pelo participante nesse caso são artefatos do experimento e portanto diferem-se do conceito de “Objeto de Estudo do experimento”

### 4.3.1 Seleção de Contexto

Conforme estabelecido na definição, o maior objetivo do estudo conduzido é avaliar se existe impacto no custo e *strength* dos critérios de teste, em razão dos paradigmas no qual o programa é escrito. Para isso, programas do domínio de estrutura de dados são considerados. Os programas foram extraídos de (Ziviani, 2005b) e (Ziviani, 2005a). Esses programas, implementados nas linguagens C e Java respectivamente em cada uma das obras, foram concebidos para exemplificar, em cada paradigma, as mesmas soluções em estrutura de dados apresentada nos dois livros. A amostra considerada para o estudo é composta de 32 programas em cada um dos paradigmas.

Sendo assim, o contexto do estudo é caracterizado como sendo um estudo *off-line*, conduzido por um estudante de mestrado, abordando um problema *real* (identificação e comparação de custos e *strength* de critérios de teste), em um contexto *específico*.

### 4.3.2 Formulação das Hipóteses

O estudo proposto busca delimitar características acerca de um assunto ainda desconhecido, em busca de evidências que possam ser usadas na definição de futuras hipóteses. Sendo assim, as hipóteses propostas não pretendem ser definitivas, mas serão estabelecidas em caráter pragmático com o intuito de viabilizar a continuidade do trabalho dentro do processo adotado.

Admitindo-se que os custos dos critérios de teste sobre um conjunto de programas sejam medidos em termos do *tamanho do conjunto de teste* e *do número de elementos requeridos* e que esses sejam influenciados principalmente: (i) pelo paradigma de desenvolvimento em que são implementados; (ii) pelas ferramentas de teste usadas; (iii) pelos tipo de critérios empregados; (iv) pela habilidade do(s) testador(es) em testar adequadamente esses programas e; (v) pelo tamanho e complexidade dos programas testados. Fixando-se as propriedades (iii) e (iv) como sendo as mesmas para dois conjuntos de programas A e B implementados respectivamente nos paradigmas procedural e OO, sabendo-se que os paradigmas de programação representam formas distintas de se entender e estruturar uma mesma solução, acredita-se que exista uma diferença de custo entre os conjuntos A e B, em razão das diferenças de paradigmas.

Da mesma forma que os custos, admitindo-se que o *strength* de cada critério de teste funcional e estrutural entre dois paradigmas seja medido em termos da *porcentagem de adequação do conjunto de teste adequado a esse critério em um paradigma sobre o mesmo critério no paradigma oposto e vice-versa*, e sabendo-se que os paradigmas de programação procedural e OO representam formas distintas de se entender e estruturar uma mesma solução, acredita-se que exista uma diferença de *strength* entre os conjunto A e B (mesmos da hipótese anterior), em razão da diferença de paradigma.

Baseado nas descrição informal das hipóteses, as seguintes hipóteses nulas e alternativas podem ser estabelecidas:

## 1<sup>a</sup> Hipótese

**Hipótese Nula:** Não existe diferença de custo dos critérios de teste, em razão do paradigma, entre o conjunto A (procedimental) e B (OO).

$$H_0: \text{Custo}(A) = \text{Custo}(B)$$

**Hipótese Alternativa:** Hipótese Alternativa: Existe uma diferença de custo dos critérios de teste, em razão do paradigma, entre o conjunto A (procedimental) e B (OO).

$$H_1: \text{Custo}(A) \neq \text{Custo}(B)$$

## 2<sup>a</sup> Hipótese

**Hipótese Nula:** Não existe diferença de *strength* dos critérios de teste, em razão do paradigma, entre o conjunto A (procedimental) e B (OO).

$$H_0: \text{Strength}(A) = \text{Strength}(B)$$

**Hipótese Alternativa:** Existe uma diferença de *strength* dos critérios de teste, em razão do paradigma, entre o conjunto A (procedimental) e B (OO).

$$H_1: \text{Strength}(A) \neq \text{Strength}(B)$$

### 4.3.3 Seleção de Variáveis

Com base no contexto e nas hipóteses estabelecidas para o estudo, as variáveis independentes e dependentes são definidas.

#### Variáveis Independentes

Retomando o que foi apresentado na Seção 3.2.1, variável independente é toda variável que pode ser manipulada ou controlada no processo de experimentação. Seguindo essa perspectiva, as principais variáveis independentes desse estudo são:

- Paradigma em que os programas estão implementados;
- Ferramentas de teste usadas;
- Critérios de teste empregados;
- Tamanho e complexidade dos programas sob teste;
- habilidade do testador em aplicar corretamente os critérios.

Apesar de todas essas variáveis independentes, a única considerada de interesse para essa investigação, é o paradigma em que os programas estão implementados, constituindo-se como o único fator de interesse do experimento. Esse fator, possui dois tratamentos: paradigma de implementação OO e paradigma de implementação procedural. As demais variáveis serão fixadas no experimento para não interferirem nos efeitos obtidos.

Além dessas, outras variáveis adversas (ambiente em que o testador aplicará os testes, ganho de experiência e cansaço durante a execução da atividade, por exemplo) podem ter alguma interferência nos resultados. Contudo, admite-se que esses efeitos adversos são pouco relevantes e desta forma, tais variáveis não serão consideradas. Ao invés disso, alguns princípios de *design* do experimento deverão ser seguidos, visando minimizar tais efeitos (Seção 3.2.1).

## Variáveis Dependentes

As variáveis dependentes são aquelas nas quais é observado o resultado da manipulação das variáveis independentes. No caso desse estudo, as variáveis dependentes definidas para descrever os custos dos critérios são:

- Número de Elementos Requeridos/Programa: Medida do total de elementos requeridos gerados para **cada** programa, durante a aplicação de um critério em um determinado paradigma.
- Número de Casos de Teste/Programa: Medida do tamanho do conjunto de teste adequado gerado para **cada** programa, durante a aplicação de um critério em um determinado paradigma.
- Número de Elementos Requeridos Não-Executáveis/Programa: Medida do total de elementos requeridos não-executáveis detectados para **cada** programa, durante a aplicação de um critério em um determinado paradigma.
- Número de Elementos Requeridos/Critério: Medida da média e desvio padrão do total de elementos requeridos gerados para **todos** os programas, durante a aplicação de um critério em um determinado paradigma.
- Número de Casos de Teste/Critério: Medida da média e desvio padrão do total dos tamanhos dos conjuntos de teste adequados gerados para **todos** os programas, durante a aplicação de um critério em um determinado paradigma.
- Número de Elementos Requeridos Não-Executáveis/Critério: Medida da média e desvio padrão do total de elementos requeridos não-executáveis detectados para **todos** os programas, durante a aplicação de um critério em um determinado paradigma.

As variáveis dependentes definidas para descrever o *strength* dos critérios são:

- Porcentagem de Cobertura/Programa no paradigma oposto: Esta é uma medida da porcentagem de cobertura do conjunto de teste adequado a cada programa em um determinado paradigma, sobre o seu correspondente no paradigma oposto, durante a aplicação de um critério.
- Porcentagem de Cobertura/Critério: Esta é uma medida da média e desvio padrão das porcentagens de cobertura dos conjuntos de teste adequados a todos os programas em um determinado paradigma, sobre os seus correspondentes no paradigma oposto, durante a aplicação de um critério.

Em adição às variáveis dependentes citadas, algumas outras métricas foram definidas e serão coletadas concomitantemente no estudo. O objetivo dessa coleta é registrar propriedades que possam ser relevantes e que estejam correlacionadas ao comportamento das variáveis dependentes, descrevendo mais detalhes do cenário dos testes funcionais e estruturais, com relação à diferença de paradigma. Essas métricas podem ser ainda reaproveitadas no contexto de futuros estudos experimentais, tanto para a derivação de novas hipóteses a partir das informações fornecidas, quanto para comparação de dados de teste provenientes de programas cujas medidas se assemelhem às dos programas que compõe este estudo. Desta forma, um total de doze métricas de implementação foram definidas. A saber: total de unidades (procedimentos/métodos e construtores); total LOC (linhas de código s/ comentários); total de comandos de desvio; total de comandos recursivos; total unidades sem parâmetros; total de unidades c/ parâmetros apenas do tipo primitivo; total de unidades c/ parâmetros apenas do tipo abstrato (valor ou referência); total de unidades c/ parâmetros mistos (primitivos e abstratos); total de unidades sem retorno; total de unidades c/ retorno do tipo primitivo; total de unidades c/ retorno do tipo abstrato(valor ou referência) e complexidade ciclomática (McCabe).

Além das métricas de implementação, foram medidos nos testes de cada programa: o número de casos de teste que passaram nos critérios funcionais, ou seja, a quantidade de casos de teste cujas assertivas avaliaram como verdadeira a igualdade entre o resultado gerado e o resultado esperado; a cobertura dos critérios estruturais fluxo de controle pelo conjunto de teste funcional adequado; a cobertura dos critérios estruturais fluxo de dados pelo conjunto de teste fluxo de controle adequado e o número de casos de teste extras necessários para adequação dos critérios estruturais nessas duas etapas.

#### 4.3.4 Design do Experimento

O design do experimento tem impacto direto sobre a forma de analisar os resultados. Um design experimental adequado também serve para minimizar a influência de fatores adversos sobre os resultados obtidos.

Conforme estabelecido anteriormente, o estudo contém apenas um fator de interesse, ou seja, o paradigma das implementações testadas, o qual pode assumir dois tratamentos: procedural ou orientado a objeto.

Com relação às demais variáveis independentes fixas, foram aplicados os seguintes princípios de *design*, visando a redução da probabilidade de erro experimental dentro do contexto definido:

- Aninhamento dos critério de teste: Apesar de não ser um fator de interesse, já que a comparação entre critérios dentro de um mesmo paradigma não é relevante para os objetivos dessa investigação, diferentes critérios de teste implicam em maneiras distintas de se determinar os custos e strengths em cada paradigma. Para resolver essa questão os critérios de teste foram agrupados em três níveis (funcional, estrutural fluxo de controle e estrutural fluxo de dados) dentro de cada paradigma, seguindo o princípio de *aninhamento*. Os critérios que compõem o nível funcional são *critério Particionamento por Classe de Equivalência* e *critério Análise Valor Limite*. Os critério escolhidos para compor o nível Estrutural Fluxo de Controle são *critério Todos-Nós* e *critério Todas-Arestas*. Os critério escolhidos para compor o nível Estrutural Fluxo de Dados são *critério Todos-Usos* e *critério Todos-Potenciais-Usos*. Esse princípio permite que cada nível seja avaliado separadamente dentro de cada tratamento, possibilitando uma comparação efetiva de condições (níveis) similares entre os dois paradigmas. Ao fator “Paradigma” denomina-se fator aninhador e ao fator “Critério de teste” fator aninhado. Esse design foi preferido em relação ao cruzado (*crossed design*), já que ele permite a comparação dos paradigmas em relação a um mesmo tipo de critério, sem implicar na análise inversa (característica do design cruzado), ou seja, também comparar os critérios em relação a um mesmo tipo de paradigma.
- Ordem dos testes: Com a finalidade de evitar que o testador ganhe habilidade pela ordem com que os programas são testados em um determinado paradigma e que a rotina neste lhe favoreça ou prejudique no outro paradigma, seja pelo tédio, ganho de experiência ou cansaço acumulado durante a execução dos testes, o princípio da aleatoriedade deverá ser usado para definir a ordem em que os programas serão testados e para cada programa, a ordem do paradigma a ser considerado primeiro. A ordem dos testes é aninhada a cada nível do critério, já que a estratégia de teste definida para geração dos conjuntos de teste é incremental e, portanto, existe uma relação de interdependência com relação ao conjunto de teste entre os critérios.

Para as demais variáveis independentes fixas não foi aplicado nenhum princípio de *design*.

A variável “ferramenta de teste” terá um valor fixo e distinto para cada um dos paradigmas. No paradigma procedural, o valor é determinado pelo par CUTE e Poke-Tool para aplicação dos critérios funcionais e estruturais respectivamente (Sommerlad e Graf, 2007; Chaim, 1991). No paradigma OO, JUnit e JaBUTi representam na mesma ordem, as ferramentas para aplicação dos

critérios de teste (Riehle, 2008; Vincenzi et al., 2003). Apesar de não serem as mesmas ferramentas aplicadas nos dois paradigmas, existe uma grande semelhança quanto à forma de operar e às assertivas empregadas. Além disso, ambas as ferramentas de teste estrutural, apóiam os critérios de interesse para o estudo.

A “especificação” dos programas que compõem o conjunto procedural é a mesma dos programas que compõem o conjunto OO. Além disso, ambos os conjuntos estão balanceados (mesma quantidade de programas nos dois paradigmas).

Mais detalhes sobre as questões de validade do experimento são tratadas na Seção 4.3.6.

O *design* experimental resultante definido para esse estudo experimental é ilustrado na Figura 4.1.

Paradigma					
Procedimental			OO		
	Critérios			Critérios	
Fun.	E.F.C.	E.F.D.	Fun.	E.F.C.	E.F.D.
Todos os Programas Proced.	Todos os Programas Proced.	Todos os Programas Proced.	Todos os Programas OO	Todos os Programas OO	Todos os Programas OO

**Figura 4.1:** Design Aninhado em dois estágios: No primeiro o fator de interesse **paradigma**, com dois tratamentos: OO e procedural; No segundo o fator bloqueante **critério** com três níveis: Funcional, Estrutural Fluxo de Controle e Estrutural Fluxo de Dados.

Neste estudo, os casos de testes serão definidos para cada programa e em cada um dos níveis de critérios de forma incremental, ou seja, o conjunto de teste gerado para o nível funcional será reutilizado na construção do conjunto de teste estrutural fluxo de controle e assim sucessivamente. Com isso, foi necessário organizar uma estratégia de teste que viabilizasse essa condução de forma sistemática, obedecendo o *design* experimental definido.

A estratégia de teste definida foi dividida em quatro seções principais. Na primeira seção (i), definiu-se que será gerado o conjunto de teste adequado aos critérios funcionais. Este conjunto de teste deverá ser derivado da especificação e portanto, deverá adequar-se aos critérios de teste funcionais tanto para a implementação procedural quanto para a implementação OO do programa. Este será o conjunto base para definição do conjuntos de teste na seção seguinte e o número de elementos requeridos nessa fase será determinado em termos do número de classes de equivalência e valores limites gerados.

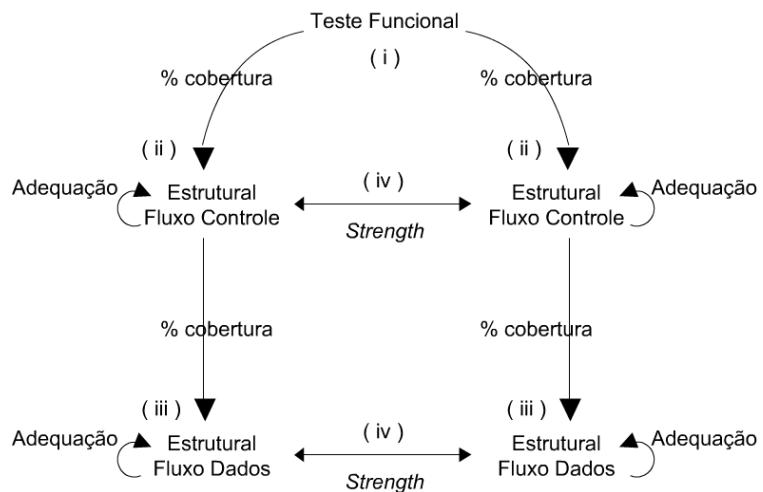
Obtido o conjunto base, na seção seguinte (ii) deverá ser avaliada a cobertura do mesmo para o segundo nível de critérios, ou seja, os critérios estruturais fluxo de controle. As porcentagens

de cobertura atingidas nos dois paradigmas deverão ser anotadas no formulário de testes de cada implementação e em seguida deverá ser feita a adequação dos conjuntos de testes e a determinação dos elementos requeridos não-executáveis para os critérios estruturais deste nível.

O conjunto de teste adequado obtido na segunda seção realimentará os teste estruturais fluxo de dados na terceira seção (iii). Da mesma maneira, as porcentagens de cobertura atingidas nos dois paradigmas deverão ser anotadas no formulário de testes de cada implementação e em seguida deverá ser feita a adequação dos conjuntos de testes e a determinação dos elementos requeridos não-executáveis para os critérios estruturais deste nível.

Na quarta e última seção(iv), os conjuntos de teste adequados gerados na segunda e terceira seção deverão ser convertidos na linguagem do paradigma oposto para verificação do *strength* dos dois tipos de teste estrutural, por meio da anotação da porcentagem de cobertura obtida.

O diagrama exibido na Figura 4.2 ilustra a descrição da estratégia de teste apresentada.



**Figura 4.2:** Diagrama das quatro etapas da estratégia de teste definida.

### 4.3.5 Instrumentação

A instrumentação do estudo foi elaborada em várias etapas, de forma a suprir as três categorias de artefatos requeridos para operação do experimento: objetos do experimento, *guidelines* e instrumentos de medida.

A primeira categoria consiste do *benchmark* de programas aos quais serão aplicados os tratamentos do experimento. Conforme explicado na Seção 4.3.1, os programas que compõem o *benchmark* foram obtidos de (Ziviani, 2005b) (programas C, representantes do paradigma Procedimental) e (Ziviani, 2005a) (programas Java, representantes do paradigma OO). Contudo, por terem sido projetados com propósito acadêmico, esses programas precisaram de alguns ajustes para aproveitamento no estudo. Um dos motivos foi a incompatibilidade e/ou ausência de especificações em um formato apropriado para os testes e independente para cada programa. O outro motivo, foi a

eventual incompatibilidade das implementações entre os dois paradigmas, quanto às funções: algumas operações implementadas na linguagem OO simplesmente não existiam ou eram diferentes daquelas implementadas na linguagem procedural, apesar de se referirem à mesma especificação. Devido a isso, houve a necessidade de preparar previamente esses programas seguindo alguns procedimentos padrões. Primeiramente foi preciso separar as implementações referentes a cada especificação. Em muitos casos, uma mesma especificação possuía um número diferente de classes implementadas em Java com relação ao número de programas em C. O relacionamento especificação/classes e especificação/programa C foi feito por meio de um trabalho seqüencial de análise e correspondência entre cada solução comum estabelecida nos dois livros com seus referidos programas. Após o estabelecimento dessa correspondência, as especificações foram descritas em um formato padrão, pré-estabelecido. Esse formato de especificação foi derivado e adaptado do empregado no pacote de laboratório de Basili et al. (2008).

Um *template* auto-descritivo do documento de especificação empregado no experimento é mostrado nas Figuras 4.3, 4.4, 4.5, 4.6.

## Capítulo 1

# Introdução

### 1.1 Propósito

<Descrever o propósito da especificação>

*“ex.: Este documento descreve a especificação de um programa de pilha por arranjo.”*

### 1.2 Escopo

< Descrever o escopo do programa especificado (por exemplo, complexidade de uso, finalidade e uma breve explicação do que faz)>

*“ex.: O programa Pilha Arranjo é um programa simples, com finalidade acadêmica de exemplificar o tipo abstrato de dados pilha implementado por arranjo e suas principais operações. Existem aplicações para listas lineares nas quais inserções, retiradas e acessos a itens ocorrem sempre em um dos extremos da lista. Uma pilha é uma lista linear em que todas as inserções, retiradas e geralmente todos os acessos são feitos em apenas um extremo da lista.”*

### 1.3 Visão Geral

< Descrever a organização do conteúdo no restante do documento >

*“ex.: O restante desse documento está organizado da seguinte forma: Algumas definições de termos importantes e observações são apresentadas nas subseções seguintes. A seção 2 contém uma descrição geral sobre pilhas e em específico, pilhas por arranjo. A seção 3 apresenta os requisitos funcionais para o programa correspondente.”*

### 1.4 Definições

< Descrever a definição de termos e conceitos usados no texto do documento (um conceito por linha >

*“ex.:*

*Item – Elemento que constitui a pilha (Um item pode ser inserido ou removido de uma pilha).*

*Topo – referência para o último item no topo da pilha.”*

### 1.5 Observações

< Colocar observações pertinentes ao conteúdo do texto, como referências bibliográficas, adaptações e considerações. Cada classe de observação deve ser separada por subseção. Os textos de todas as seções devem ser preenchidos com a mesma formatação dessa meta-descrição (a saber: fonte Times New Roman, tamanho 12, formatação normal) >

*“ex.:*

*1.5.1 Referências*

*A especificação descrita nesse documento baseia-se nos programas e textos contidos em:*

*Ziviani, N. Projeto de Algoritmos: com implementações em Pascal e C. 2 ed. rev. E ampl. São Paulo: Thomson, 2004.*

*Ziviani, N. Projeto de Algoritmos: com implementações em Java e C++. São Paulo: Thomson, 2007. (...)”*

**Figura 4.3:** Template do documento de especificação. Primeira página.

## Capítulo 2

# Descrição Geral

### 2.1 Perspectiva do Produto

< Uma descrição geral sobre a motivação ou o propósito de uso do mesmo >

*"Existem aplicações para listas lineares nas quais inserções, retiradas e acessos a itens ocorrem sempre em um dos extremos da lista. Uma pilha é uma lista linear em que todas as inserções, retiradas e geralmente todos os acessos são feitos em apenas um extremo da lista..."*

*Os itens em uma pilha estão colocados um sobre o outro, com o item inserido mais recentemente no topo e o item inserido menos recentemente no fundo. O modelo intuitivo de uma pilha é o de um monte de pratos em uma prateleira, sendo conveniente retirar pratos ou adicionar novos pratos na parte superior. Esta imagem está freqüentemente associada com a teoria de autômatos, onde o topo de uma pilha é considerado como o receptáculo de uma cabeça de leitura/gravação que pode empilhar e desempilhar itens da pilha (Hooper e Ullman, 1969).*

*As pilhas possuem a seguinte propriedade: o último item 'inserido' é o primeiro item que pode ser retirado da lista. Por esta razão as pilhas são chamadas de listas lifo, termo formado a partir de "last-in, first-out". Existe uma ordem linear para pilhas, que é a ordem do "mais recente para o menos recente". Esta propriedade torna a pilha uma ferramenta ideal para processamento de estruturas aninhadas de profundidade imprevisível, situação em que é necessário garantir que subestruturas mais internas sejam processadas antes da estrutura que as contenham. A qualquer instante uma pilha contém uma seqüência de obrigações adiadas, cuja ordem de remoção da pilha garante que as estruturas mais internas serão processadas antes das estruturas mais externas.*

*Estruturas aninhadas ocorrem freqüentemente na prática. Um exemplo simples é a situação em que é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente. O controle de seqüências de chamadas de subprogramas e sintaxe de expressões aritméticas são exemplos de estruturas aninhadas. As pilhas ocorrem também em estruturas de natureza recursiva, tais como árvores. [...]*

*Em uma implementação através de arranjos os itens da pilha são armazenados em posições contíguas de memória, conforme ilustra a Figura 1.1. Devido às características da pilha as operações de inserção e de retirada de itens devem ser implementadas de forma diferente das implementações usadas anteriormente para listas. Como as inserções e as retiradas ocorrem no topo da pilha, um cursor chamado topo é utilizado para controlar a posição do item no topo da pilha.*

*(...)"*

**Figura 4.4:** Template do documento de especificação. Segunda página.

## 2.2 Funções do Produto

< Uma descrição alto nível das funções (operações) que o programa deve implementar >

“ex.:

*O programa deve permitir as seguintes operações sobre a estrutura de dados Pilha:*

*Criar uma pilha vazia de tamanho máximo 1000;*

*Criar uma pilha vazia de tamanho máximo definido pelo usuário;*

*Verificar se a pilha está vazia;*

*Fazer a pilha ficar vazia;*

*Empilhar um item no topo da pilha;*

*Desempilhar um item que está no topo da pilha, retirando-o da pilha;*

*Verificar o tamanho atual da pilha*

.”

## 2.3 Características do Usuário

< Uma descrição de características do usuário, como a forma que o mesmo interage com o programa e as restrições ou considerações feitas sobre o mesmo >

“ex.:

*“O usuário interage com o programa por meio da invocação de seus métodos/procedimentos. Para tanto, assume-se que o mesmo conheça a interface dessas unidades, assim como a estrutura de eventuais Tipos Abstratos de Dados fornecidos na entrada ou recebidos no resultado da operação.”*

(...)"

## 2.4 Abreviações

< Uma listagem das abreviações usadas durante o texto do documento e seus respectivos significados, ordenados por linha >

“ex.:

*Linf: limite inferior*

*Lsup: limite superior”*

**Figura 4.5:** Template do documento de especificação. Terceira página.

## Capítulo 3

# Requisitos Específicos

### 3.1 Requisitos Funcionais de <nome do programa>

<Uma descrição inicial de como os requisitos funcionais são apresentados e a descrição própria de cada requisito funcional dentro de uma subseção própria.>

“Ex.:

*Nessa seção, cada requisito funcional é descrito em termos das operações executadas pelo programa. Cada operação é descrita por meio dos seguintes campos:*

*Descrição: Descrição geral sobre a função a ser executada pela referida operação.*

*Entrada: Representa os dados que devem ser fornecidos como entrada para que a operação execute suas funcionalidades.*

*Pré-Condição: Uma ou mais pré-condições que são assumidas como verdadeiras, para a execução da funcionalidade da operação.*

*Resultado Esperado: Descreve a resposta do programa após o funcionamento da operação na(s) situação(ões) considerada(s) válida(s).*

*Resultado Adverso: Descreve a resposta do programa após o funcionamento da operação na(s) situação(ões) considerada(s) inválida(s).*

*Os campos Entrada, Resultado Esperado e Resultado Adverso podem ou não estarem preenchidos. Para o último caso, estarão grifados com um hífen e subentende-se que não se aplica nenhum valor aos mesmos. O campo pré-condição só será informado quando necessário.*

#### 3.1.1 Operação 1\*

*Descrição: Criar Pilha vazia com tamanho máximo igual a 1000.*

*Entrada: -*

*Resultado Esperado: Uma pilha vazia de tamanho máximo igual a 1000 é criada com sucesso.*

*Resultado Adverso: -*

#### 3.1.2 Operação 2(J)

*Descrição: Criar pilha vazia com tamanho máximo definido pelo usuário.*

*Entrada: Valor inteiro de tamanho máximo da pilha.*

*\*Pré-Condição: Tamanho máximo da pilha fornecido é menor do que valor máximo de inteiros.*

*Resultado Esperado: Uma pilha vazia de tamanho máximo igual ao valor definido pelo usuário é criada com sucesso.*

*Resultado Adverso: Valor de tamanho máximo fornecido na entrada é negativo e pilha não é criada.*

(...)"

**Figura 4.6:** Template do documento de especificação. Quarta página.

Alguns exemplos de especificações usadas no experimento foram listadas no Apêndice A, página 127.

Grande parte dos textos das especificações basearam-se no texto original dos livros (Ziviani, 2005b) e (Ziviani, 2005a). Esses textos foram modificados ou adaptados, conforme cada caso, para suprir estritamente as necessidades de operação do estudo (como desvinculamento da descrição de uma especificação com algum código-fonte exemplo e remoção de dependências entre os textos de especificações distintas). Esta medida foi tomada com o intuito de interferir o mínimo possível sobre o conteúdo do texto original e ao mesmo tempo atender às necessidades impostas pelo princípio de aleatoriedade dos testes, conforme definido no *design* experimental.

Para evitar a redundância de referências para um mesmo autor nas especificações e pelo fato deste trabalho estar baseado na obra de terceiros, os créditos ao autor original foram atribuídos logo no início das especificações. Os trechos alterados/adaptados foram colocados entre colchetes, com a finalidade de distinção sobre o texto original.

Após a descrição das especificações no formato estabelecido, um trabalho de revisão sobre as especificações foi feito, buscando possíveis erros de preenchimento e formatação em cada uma. Finalizada a revisão, as especificações foram avaliadas pelos orientadores do trabalho e consideradas aptas para condução do experimento.

Além das especificações, a aplicação dos tratamentos no experimento requer outros documentos que auxiliem na execução dos testes. Para tanto foi concebido um documento de implementação, de forma a suprir o testador das informações necessárias para a construção dos casos de testes funcionais sem que fosse necessária a leitura do código-fonte das implementações (como parâmetros de entrada e tipos de retorno esperados por operação). Esse documento serve ainda para registrar métricas de implementação. Apesar de serem essencialmente iguais em formato, duas variações do documento de implementação foram definidas: uma para os programas em C e outra para as implementações Java. A separação foi feita com a finalidade de melhorar a organização e o reaproveitamento das informações obtidas para futuros estudos que venham a utilizar esse material. A figura 4.7, 4.8, 4.9 ilustram um *template* auto-descritivo do documento de implementação em C. O *template* é composto de três partes: “Dados da Implementação” que consiste de um cabeçalho com algumas informações gerais sobre a implementação. “Interface métodos/procedimentos e Estruturas Globais” onde as interfaces dos construtores e métodos ou procedimentos (dependendo da linguagem) são exibidos, assim como eventuais estruturas globais que sejam usadas ou retornadas pelos mesmos e que sejam necessárias para que o testador possa realizar os testes funcionais.

## Dados da Implementação

Autores da implementação:

"ex.: Nívio Ziviani"

Documento de especificação relativo:

"ex.: Pilha por Arranjo"

Nome do arquivo contendo o código fonte:

"ex.: Pilha.c"

Fonte:

"ex.: Livro: Ziviani, N.; Projeto de Algoritmos: com implementações em PASCAL e C; 2a edição; editora Thomson, 2004."

Paradigma da linguagem:

"ex.: Procedimental"

Número de operações implementadas:

"ex.: 5"

Comentários e Observações

"ex.:

1. Apenas as funções de interface de cada operação são mostradas.
2. O método `main` (contido na versão original do código) foi excluído por não fazer parte das operações especificadas (sua função era apenas ilustrar o uso dos demais métodos).
3. As funções que não são comuns às operações especificadas nos dois paradigmas, foram removidas (contidas na versão original do código)."

**Figura 4.7:** Template do documento de implementação em C. Primeira página.

## Interface métodos/procedimentos e Estruturas Globais

```
"ex.:
/*pilha-arranjo.c*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define MaxTam 1000

typedef int Apontador;
typedef int TipoChave;

typedef struct {
    TipoChave Chave;
    /* --- outros componentes --- */
} TipoItem;

typedef struct {
    TipoItem Item[MaxTam];
    Apontador Topo;
} TipoPilha;

void FPVazia(TipoPilha *Pilha){}
int Vazia(TipoPilha Pilha){}
void Empilha(TipoItem x, TipoPilha *Pilha){}
void Desempilha(TipoPilha *Pilha, TipoItem *Item){}
int Tamanho(TipoPilha Pilha){}
"
```

**Figura 4.8:** Template do documento de implementação em C. Segunda Página

## Métricas de implementação

<i>Atributos Comuns (Classe/Programa)</i>	<i>Medida</i>
Total de unidades (procedimentos/métodos e construtores)	"ex.: 5"
Total LOC	36
Total de comandos de desvio	4
Total de comandos recursivos	0
Total unidades sem parâmetros	0
Total de unidades c/ parâmetros apenas do tipo primitivo	0
Total de unidades c/ parâmetros apenas do tipo abstrato (valor ou referência)	5
Total de unidades c/ parâmetros mistos (primitivos e abstratos)	0
Total de unidades sem retorno	3
Total de unidades c/ retorno do tipo primitivo	2
Total de unidades c/ retorno do tipo abstrato (valor ou referência)	0
Complexidade de McCabe	7"

**Figura 4.9:** Template do documento de implementação em C. Terceira Página

Além dos documentos de especificação e implementação, dois formulários foram definidos para aplicação dos testes: o primeiro é um formulário auxiliar para definição das classes de equivalência, durante os testes funcionais. A Figura ilustra um template auto-descritivo do formulário de classes de equivalência. Como pode ser observado na ilustração, o formulário é composto de cabeçalho e uma seção de classes de equivalência. O formato do *template* foi concebido de forma que o testador derive as classes de equivalência a partir de cada operação especificada.

## Documento de Classes de Equivalência

### Programa Alvo:

<nome do documento de especificação relativo a este documento>  
 "ex.: Pilha por Arranjo"

**Nome :**

**Local e data:**

<nome do autor das classes de equivalência>  
 "ex.: Jorge Ferreira da Silva"      <local e data de autoria do documento>  
 "ex.: São Carlos, 30 de abril de 2008"

---

### Operação <número da operação na especificação>:

**<Descrição da operação>**

"ex.: Cria uma pilha vazia com um tamanho máximo default"

<i>Condição de entrada</i>	<i>Classes válidas</i>		<i>Classes Inválidas</i>	
	ID	Classe	ID	Classe
"ex.: Tamanho Maximo	1	0<=Tamanho máximo	2	tamanho máximo < 0"

---

### Operação <número da operação seguinte na especificação>

**<Descrição da operação>**

"ex.: Faz a pilha ficar vazia"

<i>Condição de entrada</i>	<i>Classes válidas</i>		<i>Classes Inválidas</i>	
	ID	Classe	ID	Classe
"ex.: Tamanho da pilha ( pilha )"	1	0<= pilha <=tamanho máximo da pilha	-	—"

**Figura 4.10:** Template do formulário de classes de equivalência.

O segundo formulário (Figuras 4.11, 4.12, 4.13 e 4.14, também denominado de “formulário de testes”, serve para registrar os casos de teste para cada critério avaliado e as medidas de teste auferidas. Assim como o documento de implementação, duas variações do formulário foram estabelecidas: Uma para os testes procedimentais e outra para os OO’s.

## Dados Gerais

Autor(es) dos casos de teste:

"ex.: Jorge Ferreira da Silva"

Documento de especificação relativo:

"ex.: pilha por arranjo"

Comentários e Observações

"Ex.:  
Ordem na randomização: 17º  
Ordem de teste por paradigma: OO/Procedimental ."

**Figura 4.11:** Template do Formulário de testes OO. Primeira Página

## Testes Funcionais OO

Formulário de classes de equivalência:

"ex.: CLE.doc"

### Critério Particionamento de Classes de Equivalência

Nº CT	Operação <número da operação no documento de especificação>	Classe de Equivalência <número identificador da classe de equivalência à qual se aplica o CT>	Dados de Teste		Resultado Esperado <resultado esperado pela execução do CT>
			Parâmetros	Estado (pré-condições)	
"ex.:1	1	1	-	-	Pilha criada com o tamanho máximo default
2	2	1	Tamanho máximo da pilha	Tamanho máximo da pilha = 50	Pilha vazia com o tamanho 50 criada com sucesso.
3	2	2	Tamanho máximo da pilha	Tamanho máximo da pilha = -50	Pilha não pode ser criada.
4	4	1	Pilha	pilha  = 0	True
5	4	2	Pilha	pilha  > 10	False"

**Figura 4.12:** Template do formulário de testes OO. Segunda Página

### Critério Análise Valor Limite

Nº CT	Operação	Classe de Equivalência	Limite <S para superior, I para inferior>	Dados de Teste		Resultado Esperado
				Parâmetros	Estado (pré-condições)	
"ex.: 1	3	2	I	TipoPilha da pilha a ser verificada	Tamanho atual da pilha = 1	Valor 0 é retornado indicando que a pilha não está vazia
2	3	2	S	TipoPilha da pilha a ser verificada	Tamanho atual da pilha = 1000	Valor 0 é retornado indicando que a pilha não está vazia
3	4	1	I	TipoItem do item a ser empilhado/ Ponteiro de TipoPilha da pilha onde o item será empilhado	Tamanho atual da pilha = 0	Item é empilhado no topo da pilha com sucesso.
4	4	1	S	TipoItem do item a ser empilhado/ Ponteiro de TipoPilha da pilha onde o item será empilhado	Tamanho atual da pilha = 999	Item é empilhado no topo da pilha com sucesso.
5	5	1	I	Ponteiro de TipoPilha da pilha onde ocorrerá o desempilhamento	Tamanho atual da pilha = 1	Item do topo da pilha é desempilhado com sucesso
6	5	1	S	Ponteiro de TipoPilha da pilha onde ocorrerá o desempilhamento	Tamanho atual da pilha = 1000	Item do topo da pilha é desempilhado com sucesso
7	6	1	I	Ponteiro de TipoPilha da pilha a ser verificada	Tamanho atual da pilha = 0	Número de itens da pilha é retornado.
1	3	2	I	TipoPilha da pilha a ser verificada	Tamanho atual da pilha = 1	Valor 0 é retornado indicando que a pilha não está vazia
2	3	2	S	TipoPilha da pilha a ser verificada	Tamanho atual da pilha = 1000	Valor 0 é retornado indicando que a pilha não está vazia"

**Figura 4.13:** Template do formulário de testes OO. Terceira Página

### Métricas de teste

<i>Propriedade</i>	<i>Part. Classe Equivalência</i>	<i>Análise Valor Limite</i>	<i>Total</i>
Total de Classes de Equivalência geradas	"ex.: 8	13	13
Total de CT's gerados	8	13	17
Total de CT's que passaram	8	13	17"

<i>Propriedade</i>	<i>Todos-Nós</i>	<i>Todos-Arcos</i>
Total de Elementos Requeridos	"ex.: 13	6
Cobertura Funcionais Adequados	100%	100%
Nº CT's extras para adequação	0	0
Total de CT's	17	17
Nº elementos não executáveis	0	0"

<i>Propriedade</i>	<i>Todos-Usos</i>	<i>Todos-Potenciais-Usos</i>
Total de Elementos Requeridos	"ex.: 8	9
Cobertura Estruturais-FC Adequados	100%	100%
Nº CT's extras para adequação	0	0
Total de CT's	17	17"

**Figura 4.14:** Template do formulário de testes OO. Quarta Página

Além desses formulário e documentos, a instrumentação do estudo-experimental conta com os códigos-fontes dos programas em linguagem C e Java, as ferramentas de teste e um documento de *guidelines*. Nessas *guidelines* são encontradas informações específicas sobre a condução do experimento como a forma de se operar a máquina virtual e as ferramentas de teste envolvidas.

#### 4.3.6 Avaliação de Validade

Em razão da maneira como o estudo está sendo proposto, grande parte das questões de validade não se aplicam diretamente, uma vez que foram definidas para serem verificadas no contexto de experimentos controlados ou *quasi*-experimentos. Desta forma, as questões de validade serão tratadas conforme cada caso, sendo adaptadas para o contexto adotado sempre que possível.

##### Validade Interna

As questões de validade interna dizem respeito às influências desconhecidas que ameaçam a causalidade de um experimento sem o conhecimento do projetista (Wohlin et al., 2000).

**História e Maturação:** As ameaças do tipo história e maturação do experimento serão contingenciadas, com a aplicação da randomização sobre a ordem das especificações e programas testados, assim como de seus respectivos paradigmas. A condução dos testes também será feita em horário dedicado (matutino predominantemente), em dias úteis e seqüenciais (sem previsão de interrupções).

**Instrumentação:** A instrumentação é um dos principais subprodutos do estudo. Sendo assim, todos os cuidados necessários com formulação adequada dos *templates* de formulários e documentos, preparação dos programas e especificações usados, assim como ferramentas de teste e ambiente de aplicação estão amparados por um trabalho cauteloso de revisão bibliográfica, análise e validação dos artefatos além do reúso de diversos conceitos já aplicados com sucesso em outros pacotes de laboratório.

**Mortalidade:** Não pode ser verificada, já que não existem outros participantes para abandonar o estudo.

**Ambigüidade sobre direção da influência causal:** Se existe a relação causal, ela deve ser unidirecional no sentido investigado, pois a variável independente de interesse do estudo depende somente da decisão do desenvolvedor. Além disso, o custo e o *strength* de teste só podem ser definidos depois que os programas foram desenvolvidos e testados em um determinado paradigma.

##### Validade de Construção

As questões de validade de construção dizem respeito à generalização dos resultados para o conceito ou teoria subjacente ao experimento. Em geral, estão relacionadas ao *design* do estudo

experimental e à sua habilidade de refletir a construção da causa e efeito estudados (Wohlin et al., 2000).

**Explicação pré operacional de conceitos inadequada:** Todos os conceitos envolvidos com as medidas e tratamentos do experimento foram definidos em detalhes no plano do projeto. Um exemplo disso é a diferenciação de custos dos critérios de teste em relação aos custos da atividade de teste. O risco deste tipo de ameaça é considerado pequeno para a investigação.

**Viés mono-operacional:** Considerada uma das maiores ameaças de validade deste estudo experimental, em razão da aplicação dos tratamento por apenas um participante e em apenas um domínio de programas, o que torna a construção da causa sub-representada frente ao universo de todos os diferentes tipos de participantes e domínios de programas OO's e procedimentais possíveis.

**Viés mono-método:** É uma ameaça considerada contingenciada, uma vez que os custos dos critérios não são medidos por apenas um tipo de métrica mas oito, das quais nenhuma é baseada em julgamento pessoal, o que diminui a chance de viés em razão das medidas.

**Confusão de construção com níveis de construção:** Para essa ameaça de validade, foram definidos os níveis dos critérios de teste, mesmo os critérios de teste sendo os mesmos nos dois paradigmas.

**Interação de diferentes tratamentos:** A aplicação dos tratamentos nos dois paradigmas pelo mesmo participante é uma característica desejada nesse experimento já que dessa forma existe uma consistência maior entre paradigmas quanto à forma com que os casos de teste foram concebidos, melhorando a comparação. Contudo, também é uma questão de validade que não pode ser verificada ou contingenciada em razão da restrição de número de participantes desse estudo.

**Interação do teste com o tratamento:** As medidas coletadas não geram nenhuma expectativa quanto ao desempenho em quem aplica o tratamento, já que caracteriza atributos de critérios de teste e não qualidades humanas. Logo, o risco deste tipo de ameaça é considerado pequeno para a investigação.

**Restrição de generalização sobre as construções:** Esse tipo de ameaça diz respeito à possibilidade dos tratamentos do experimento serem favoráveis positivamente à hipótese alternativa com relação aos atributos de qualidade verificados porém comprometer indesejavelmente outros atributos de qualidade relacionados que não foram observados, por efeito colateral. No contexto desse experimento, os atributos de qualidade desconsiderados e que poderiam estar relacionados, são dependentes de desempenho humano (por exemplo, produtividade na derivação e execução dos casos de teste em razão do paradigma). Contudo, também é uma questão de validade que não pode ser verificada ou contingenciada em razão da restrição do tipo de estudo realizado.

**Expectativa do experimentador:** Como o projetista do experimento é o próprio aplicador dos tratamentos, e tem ciência de que os resultados deste não são definitivos em qualquer uma das hipóteses, não existe nenhuma expectativa ou motivação em favorecer qualquer uma das hipóteses.

### **Validade de Conclusão**

As questões de validade de conclusão dizem respeito aos detalhes que afetam a habilidade de inferir conclusões corretas sobre relações entre o tratamento e a saída do experimento.

**Fishing e taxa de erro:** O risco de *Fishing*, ou seja, o favorecimento durante a análise dos dados de um resultado esperado pelo experimentador não é considerada uma ameaça, já que como ressaltado anteriormente, não há interesse de concluir em definitivo sobre nenhuma das hipóteses, mas caracterizar informações a partir dos dados levantados na investigação das hipóteses.

**Confiabilidade das medidas:** As métricas aplicadas usam medidas direta, o que aumenta a precisão e facilidade de coleta dos dados. Além disso, acredita-se que a conjunção dos tipos de métricas usadas com a estratégia de teste empregada, contribuirá para a obtenção de uma quantidade de casos de teste menos suscetível a influência de fatores humanos.

**Confiabilidade da implementação do tratamento:** Essa ameaça é relativa à incapacidade de implementar a mesma configuração de tratamento entre todos os participantes ou em diferentes ocasiões. Não se aplica a este estudo experimental da maneira como está definido. Mesmo assim, a forma com que a instrumentação do experimento foi preparada colabora para que esta seja uma das menores ameaças em futuras replicações/redefinições do estudo. Isso porque as ferramentas e demais artefatos do tratamento serão instalados e configurados em uma máquina virtual, que deverá ser gravada em mídia digital. Dessa forma, para reproduzir grande parte da instrumentação do trabalho, o projetista do experimento deverá apenas dispor de computadores e espaço em disco onde possa copiar a máquina virtual e utilizar os objetos do experimento já preparados.

**Heterogeneidade aleatória dos participantes:** Não se aplica a este estudo experimental da maneira como está definido e portanto é uma ameaça que não pode ser determinada ou contingenciada.

### **Validade Externa**

A validade externa refere-se às condições que limitam a habilidade de generalizar os resultados do experimento para práticas da indústria.

As ameaças desse tipo poderiam ser consideradas grandes ao resultado, não fosse a restrição de escopo imposta pelo próprio tipo de estudo investigado.

Ainda que este fosse um experimento controlado, seria considerado um “teste de teoria” e não uma “pesquisa aplicada”. Desta forma, de acordo com Wohlin et al. (2000), esta seria em ordem de prioridade decrescente a última questão de validade, uma vez que “testes de teoria” são raramente relacionados a “condições específicas, populações e números de vezes para os quais o resultado deva ser generalizado”.

## 4.4 Considerações Finais

Neste capítulo foram discutidos todos os elementos de definição e planejamento do estudo experimental deste trabalho. Inicialmente, os termos e processo de experimentos controlados foram reaproveitados para construção da definição e planejamento do estudo proposto. Na definição foram estabelecidos os conceitos da investigação e no planejamento questões como hipóteses, variáveis dependentes, variáveis independentes e instrumentação. Por fim, uma seção abordando os quatro tipos de ameaças de validade foi descrita, considerando-se as limitações e restrições impostas pela forma com que o estudo experimental foi concebido.

No próximo capítulo, serão abordados os assuntos relativos à forma como o estudo experimental foi conduzido e a análise dos resultados obtidos.



# Condução do Estudo e Análise dos Resultados

## 5.1 Considerações Iniciais

A condução do estudo descreve detalhes da forma como o planejamento foi executado, assim como o emprego dos artefatos envolvidos e os procedimentos de coleta e verificação dos dados. Após coletados, os dados foram submetidos à análise, da qual foram derivadas informações relevantes ao estudo e que serviram de embasamento para as conclusões acerca do tema investigado.

## 5.2 Operação

A operação é a parte da condução do estudo que engloba a preparação dos artefatos e ferramentas, a execução das atividades do estudo definidas no plano e a verificação dos dados coletados durante a execução. A seguir são descritos em detalhes, a maneira como cada uma dessas fases foi realizada.

### 5.2.1 Preparação

A preparação da operação do estudo se deu a partir da própria definição do plano. Tendo-se definidos os documentos e formulários para execução dos testes o primeiro passo consistiu em preparar as ferramentas de uma maneira robusta para que pudessem ser usadas ao longo do

experimento, na expectativa de prever-se futuros problemas que pudessem acarretar em atrasos na condução dos testes.

A primeira preocupação com a preparação das ferramentas estava relacionada ao bom funcionamento das mesmas. A ferramenta de teste estrutural Poke-Tool, desenvolvida em âmbito acadêmico, encontrava-se descontinuada em razão da evolução das versões do compilador C, ao qual a ferramenta apresenta-se fortemente relacionada. Isso exigiu o exercício prévio da mesma, em busca de uma configuração de sistema que viabilizasse o funcionamento correto da mesma. A partir desse exercício, foi possível identificar que a última versão disponível da ferramenta encontrava-se compatível com a versão 4.1.2 do compilador GCC, rodando sobre o sistema operacional Linux.

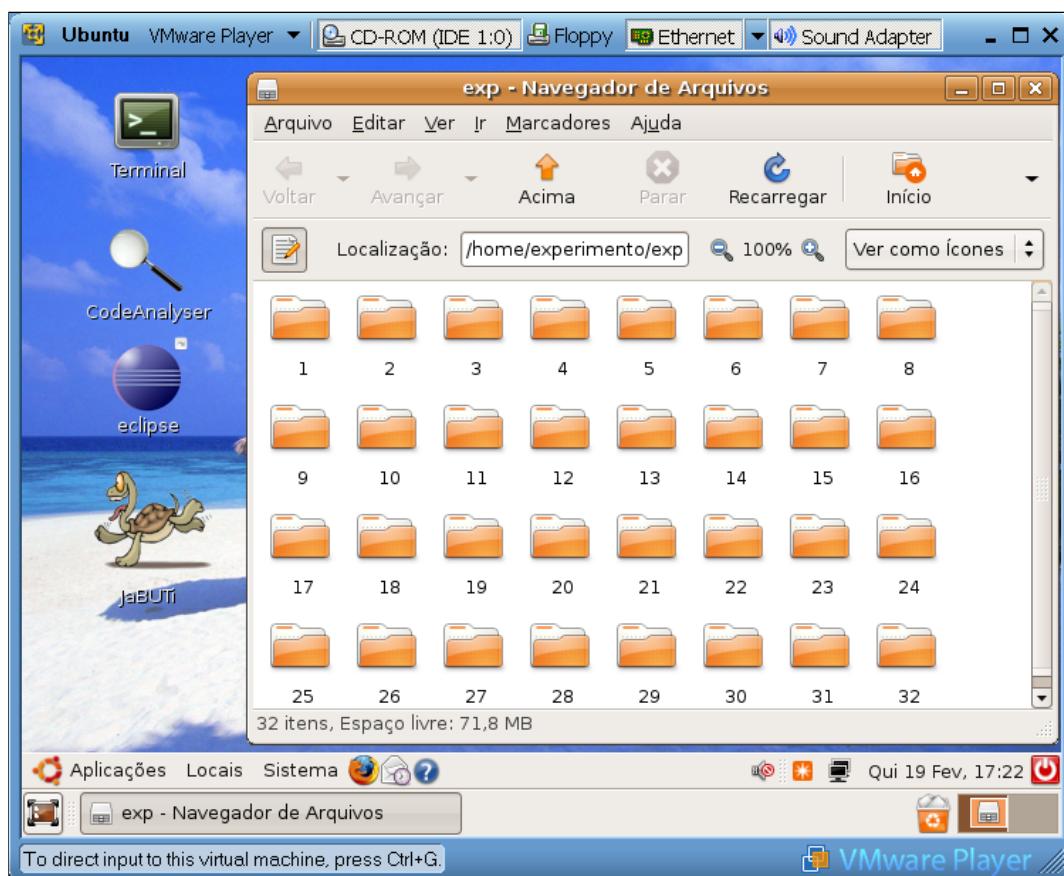
Como uma das finalidades do estudo é viabilizar a definição de futuros estudos experimentais e treinamento em teste, julgou-se adequado instalar e configurar as ferramentas de forma que elas pudessem ser facilmente reaproveitadas em novos contextos sem a necessidade do retribalho e mantendo-se alguma garantia de um funcionamento estável. Usualmente, para este fim, especifica-se as versões e os requisitos de sistema para replicação de estudos com ferramentas. Contudo, muitas vezes isso não é o suficiente para eliminar dificuldades com sua instalação/configuração e não diminui a chance de interferência por alguma configuração extra além dos requisitos exigidos. A partir dessa necessidade, surgiu-se a idéia de definir uma máquina virtual (Figura 5.1), sobre a qual essas ferramentas pudessem ser agrupadas e configuradas para uso direto. Dentre os benefícios encontrados com essa prática podem-se citar: a portabilidade não só das ferramentas e do sistema operacional mas de toda uma configuração de sistema específica, a possibilidade de replicação e geração de *backup* de forma prática e relativamente rápida, a viabilidade de evolução de configurações de ferramentas sem o comprometimento de uma configuração já estável, a possibilidade de interrupção e continuação de uma tarefa em datas diferentes mantendo-se o mesmo estado do sistema operacional e dos programas utilizados e a persistência direta dos resultados gerados internamente pela cópia em uso.

A máquina virtual do estudo foi definida com as seguintes configurações de hardware:

- Tamanho total de disco rígido virtual de 4GB, alocados sob demanda.
- 512MB de memória principal reservada.
- Detecção automática de dispositivos de áudio e vídeo do sistema hospedeiro.
- Interface de rede no modo *bridged*, no qual a máquina virtual conecta-se diretamente à rede física utilizada pelo sistema hospedeiro.

Para as configurações de software, foram escolhidas as seguintes definições:

- Sistema operacional Linux, distribuição Ubuntu - versão 7.04 (Feisty Fawn), o qual já possui nativamente a versão 4.1.2 do compilador GCC instalada e configurada corretamente.



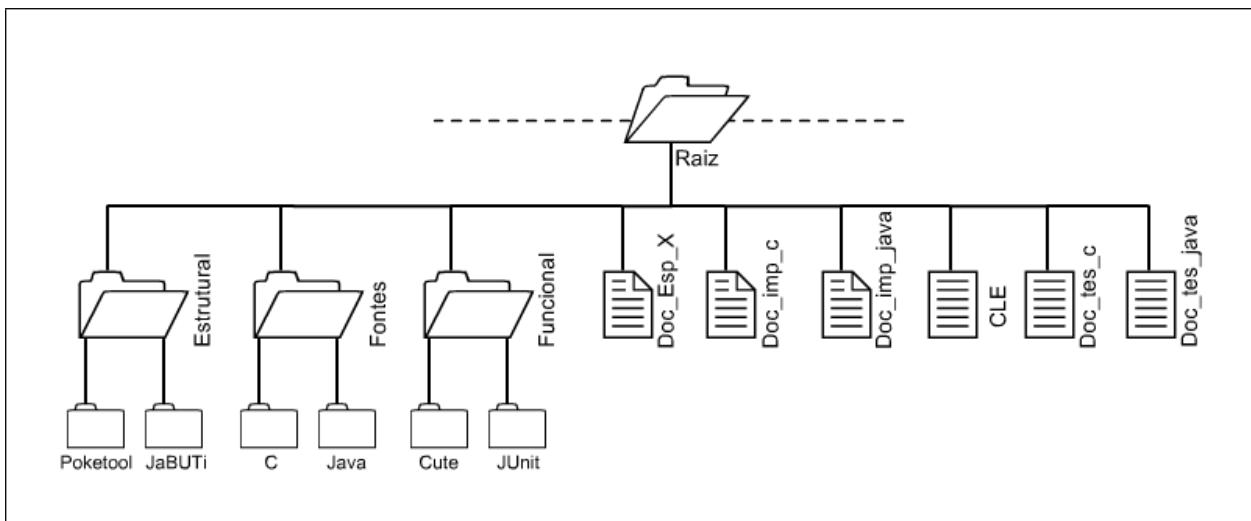
**Figura 5.1:** Máquina virtual definida para o estudo, em execução.

- IDE Eclipse para desenvolvedores C/C++ - versão 3.4.0 (Ganymede), o qual já possui o plug-in da ferramenta JUnit versão 3.8.2 instalado nativamente, para aplicação dos testes funcionais em Java.
- Plug-in da ferramenta “C/C++ Unit Testing Easier”(Cute) versão 1.6.1 instalado sobre a plataforma Eclipse, para aplicação dos testes funcionais em C.
- Ferramenta JaBUTi versão 1.0, para aplicação dos testes estruturais em Java.
- Ferramenta Poke-Tool, com variáveis de ambiente carregadas durante a inicialização do Sistema Operacional, para aplicação dos testes estruturais em C.
- Ferramenta CodeAnalyserPro versão 1.1, para coleta das medidas de implementação dos programas nas duas linguagens.

Após a instalação e configuração das ferramentas, definiu-se uma estrutura de diretórios e arquivos (Figura 5.2) para disponibilização/armazenamento dos artefatos de cada programa utilizado no experimento. Na raiz do diretório, estão disponíveis todos os documentos e formulários de teste, um subdiretório para os códigos fontes do programa (Fontes), outro para as implementações dos testes funcionais (Funcionais) e um último para os testes estruturais (Estruturais). Dentro de cada um desses diretórios foram definidos dois novos subdiretórios correspondentes a cada um dos paradigmas. Em “Fontes” nomeou-se os subdiretórios a partir das linguagens correspondentes ao paradigma: “C” e “Java”; em “Funcional” e “Estrutural” a partir das ferramentas: “Cute” e “Junit”, “Poke-Tool” e “JaBUTi”, respectivamente. Optou-se por esses nomes ao invés de simplesmente usar os paradigmas, para evitar que os artefatos de um tipo de teste fossem sobreescritos por engano por outro do mesmo programa, durante o seu armazenamento. Nos subdiretórios de cada ferramenta estrutural, convencionou-se criar um subdiretório para cada nova sessão de teste definida (relacionada a cada etapa da estratégia de teste), a menos que o conjunto de teste da sessão anterior fosse exatamente igual, caso em que seria redundante e portanto desnecessário.

Para os documentos e formulários, foram atribuídas algumas convenções aos nomes dos arquivos:

- Os documentos de especificação foram nomeados com o prefixo “Doc\_Esp”+ “nome do programa”;
- Os documentos de implementação foram nomeados “Doc\_imp\_c” e “Doc\_imp\_java”, referentes a cada uma das linguagens de implementação do programa;
- Os formulários de teste foram nomeados “Doc\_tes\_c” e “Doc\_tes\_java”, referentes a cada uma das linguagens de implementação do programa;
- O formulário de classes de equivalência foi nomeado “CLE”.



**Figura 5.2:** Estrutura de diretórios e arquivos para cada programa do estudo.

De acordo com o planejamento, a ordem de teste seria definida aleatoriamente por sorteio. Desta forma, por se tratarem de 32 especificações, um valor distinto na faixa de 1 a 32 foi atribuído a cada um dos programas do *benchmark*. O número atribuído serviu também para identificar cada programa durante a análise das medidas coletadas e para nomear o diretório raiz correspondente a cada programa na máquina virtual. A Tabela 5.1 exibe o identificador de cada programa, associado ao seu nome e descrição de funcionalidade.

Para auxiliar no sorteio da ordem das especificações e do paradigma a ser testado primeiro em cada programa, definiu-se um pequeno aplicativo (Figura 5.3) em uma linguagem de *script* para gerar seqüências aleatórias, no qual o usuário informa o título das seqüências, o número de seqüências a serem geradas e o número de elementos por seqüência desejado. Desta forma, para sortear a ordem das especificações, gerou-se uma única seqüência aleatória com “32” elementos. Para a ordem dos paradigmas, bastou-se convencionar que os valores “1” e “2” representavam o paradigma Procedimental e OO respectivamente e gerar 32 seqüências aleatórias com 2 elementos por seqüência.

### 5.2.2 Execução

A execução foi a etapa que demandou mais tempo na condução do estudo. Após todos os artefatos estarem prontos e preparados para condução, incluindo-se escrita do plano, preparação de todas as especificações, implementações e definição da máquina virtual e coleta de métricas de implementação, gastaram-se 3 meses para realização de todos os teste e coleta das medidas.

Os teste funcionais, em particular, responderam pela maior parte do tempo de condução, uma vez que todo o seu processo é manual, sendo apenas a verificação das assertivas auxiliada pelas ferramentas. Dentre as tarefas exigidas para realização dos testes funcionais podem-se citar: A leitura e correto entendimento da especificação, a definição de classes de equivalência e dos valores limites para cada operação especificada, a escrita dos casos de teste em linguagem natural,

Identificador	Nome	Descrição
1	Max	Programa para obter o valor máximo de um conjunto.
2	MaxMin1	Programa para obter os valores máximo e mínimo de um conjunto de forma não eficiente.
3	MaxMin2	Programa para obter os valores máximo e mínimo de um conjunto de forma mais eficiente que o programa MaxMin1.
4	MaxMin3	Programa para obter os valores máximo e mínimo de um conjunto de forma mais eficiente que o programa MaxMin2.
5	Ordenação1	Programa de ordenação de vetor de inteiros.
6	FibRec	Programa para calcular sequência de Fibonacci (recursivo).
7	FibIte	Programa para calcular sequência de Fibonacci (iterativo).
8	MaxMinRec	O programa para encontrar o valor máximo e mínimo de um conjunto MaxMin de forma recursiva.
9	Mergesort	Programa de ordenação pelo método Mergesort.
10	AvaliaMultMatrizes	Obtém o custo mínimo para multiplicação de "n"matrizes.
11	ListaArranjo	Programa da estrutura de dados lista com suas respectivas operações, implementada por meio de arranjos.
12	ListaAutoRef	Programa da estrutura de dados lista com suas respectivas operações, implementada por estrutura de auto-referência.
13	PilhaArranjo	Programa da estrutura de dados pilha com suas respectivas operações, implementada por meio de arranjos.
14	PilhaAutoRef	Programa da estrutura de dados pilha com suas respectivas operações, implementada por meio de auto-referência.
15	FilaArranjo	Programa da estrutura de dados fila com suas respectivas operações, implementada por meio de arranjos.
16	FilaAutoRef	Programa da estrutura de dados fila com suas respectivas operações, implementada por estrutura de auto-referência.
17	Ordenação2	Programa de ordenação pelos métodos de seleção, inserção, shellsort, quicksort e heapsort.
18	HeapSort	Programa de fila de prioridade implementado por meio de arranjos.
19	OrdenaçãoParcial	Programa de ordenação para obter os k primeiros elementos ordenados de um conjunto de tamanho n.
20	PesquisaSeqBin	Programa que implementa os métodos de pesquisa seqüencial e binária para recuperação de informação.
21	Arvore Binária	Programa que implementa árvore binária e operações de inserção, remoção e pesquisa de registros.
22	Hashing1	Programa que implementa o método de Hashing usando Lista Encadeada para tratamento de colisões.
23	Hashing2	Programa que implementa o método de Hashing usando Endereçamento Aberto para tratamento de colisões.
24	GrafoMatAdj	Programa da estrutura de dados grafo com suas respectivas operações, implementado com matriz de adjacência.
25	GrafoListaAdj1	Programa da estrutura de dados grafo com suas respectivas operações, implementado com lista de adjacência usando estrutura de auto-referência.
26	GrafoListaAdj2	Programa da estrutura de dados grafo com suas respectivas operações, implementado com lista de adjacência usando arranjos.
27	BuscaEmProfundidade	Programa que implementa a busca em profundidade em um Grafo.
28	BuscaEmLargura	Programa que implementa a busca em largura em um Grafo.
29	CFC	Programa para obtenção de componentes fortemente conectados de um grafo.
30	Prim	Programa que implementa o algoritmo de Prim para descoberta da árvore geradora mínima em um grafo ponderado.
31	CasamentoExato	Programa que implementa algoritmo de casamento exato para busca em cadeias de caracteres.
32	CasamentoAproximado	Programa que implementa algoritmo de casamento aproximado para busca em cadeias de caracteres.

**Tabela 5.1:** Tabela de programas usados no estudo com identificador, nome e descrição de funcionalidade.

**GeRandom**

**Sequências de especificações**

**Sequência 1:**  
21,26,13,17,28,29,5,9,25,27,4,1,20,19,12,11,15,8,31,6,10,22,24,2,18,23,3,30,7,14,16,32

<b>Título das sequências</b>	<b>especificações</b>
<b>Nº's de sequências</b>	1
<b>Nº de elementos/seq.</b>	32

**Figura 5.3:** Interface de aplicativo para geração de seqüências aleatórias.

definindo-se as entradas, saídas e resultados esperados, a implementação dos casos de teste nas linguagens sob estudo, a verificação das assertivas e a coleta das medidas geradas.

A condução dos teste estruturais, consumiram menos tempo que os funcionais. Nesta etapa, os conjuntos de teste iniciais já estavam estabelecidos. Dentre as tarefas exigidas nessa etapa, podem-se citar: Medição do número de elementos requeridos gerados pelos critérios, análise da cobertura dos critérios pelo conjunto de teste funcional-adequado, adequação do conjunto de teste aos critérios verificados e identificação de elementos requeridos não-executáveis. Os testes estruturais de fluxo de dados, em particular, demandaram mais tempo que os de fluxo de controle pois o número de elementos requeridos gerados eram em média superiores ao primeiro e a identificação das causas de não cobertura são menos diretas.

A análise do *strength* foi a última etapa de teste realizada. Consumiu mais tempo que os testes estruturais fluxo de controle e menos tempo que os estruturais fluxo de dados. O tempo gasto, todavia, foi considerável em relação a verificação de *strength* tradicional, já que em razão da mudança de linguagem e de paradigma, fez-se necessário a reescrita dos conjuntos de teste de cada linguagem na linguagem do paradigma oposto.

A conversão de casos de teste muitas vezes não era direta, chegando a ser impraticável dependendo do caso de teste em questão. Para resolver essa questão e viabilizar de alguma forma tal análise, alguns procedimentos heurísticos de conversão dos casos de teste foram estabelecidos e adotados. O primeiro consistiu em preservar a semelhança semântica do caso de teste mesmo quando a sintática não fosse possível, ou seja, mesmo que os casos de teste não pudessem ter uma correspondência de um para um entre cada linha de caso de teste, o propósito da execução do mesmo, em relação ao programa testado, deveria ser mantido. Para isso, o valor das entradas deveriam ser iguais sempre que possível. Caso não fossem, elas deveriam ser minimamente correspondentes, ou seja, deveriam ser suficientes para que executassem o mesmo comportamento esperado da operação. Caso a última opção ainda não fosse possível, este caso de teste deveria ser considerado um caso de teste não implementável na linguagem oposta e computado em uma coluna própria, durante a summarização dos *strengths* de teste. Além da preocupação com a chamada das operações, o testador deveria estar atento para tentar reproduzir uma configuração ou estado semântico semelhante nas estruturas de dados envolvidas com o caso de teste. Caso isso não fosse possível esse caso de teste também deveria ser contabilizado como um caso de teste não implementável na linguagem oposta. Os passos para a realização dos testes em cada ferramenta, são fornecidos no documento de *guidelines*, o qual encontra-se disponível no apêndice A.

### 5.2.3 Validação dos Dados

Um programa pode conter vários procedimentos/métodos e o formulário de teste é destinado às medidas gerais do programa, ou seja, considerando-se todos os procedimentos/métodos juntos. Para que não houvesse confusão na coleta das medidas durante as etapas de teste, foram utilizados rascunhos para registro individual das medidas por procedimento/método. Após todos os dados das

operações de um programa terem sido coletados nesse rascunho, eles foram agrupados e contados (ou calculados) para o contexto geral do mesmo. Essa decisão serviu não só para modularizar o trabalho de coleta mas também para facilitar a verificação das medidas coletadas durante a revisão de uma etapa de teste, a qual se dava sempre após o término da mesma.

## 5.3 Análise e Interpretação dos Resultados

### 5.3.1 Descrição Estatística

Conforme definido no plano, um conjunto de 32 especificações foram utilizadas no estudo, cada uma com sua correspondente implementação no paradigma procedural e OO.

Para caracterização de propriedades estáticas desses programas, 12 métricas de implementação foram avaliadas, conforme definido no plano, as quais são reportadas nas Tabelas 5.2 e 5.3. Esses dados ajudam a descrever melhor os tipos de programas envolvidos no estudo e correlacionar informações de implementação com as informações de teste geradas. Desses dados, pôde-se observar que:

- O número médio de comandos de desvio (*if*, *while*, *for*) se mostrou semelhante em ambos os paradigmas, sendo  $\bar{x} = 8,65$  no conjunto procedural e  $\bar{x} = 8,81$  no conjunto OO. Além disso, a proximidade dos desvios padrões entre os dois conjuntos ( $DP_{proc} = 7,12$  e  $DP_{OO} = 7,541$ ) também indicam uma semelhança na variação dos dados, com relação à média.
- Em 53,13% dos programas, a complexidade ciclomática foi idêntica para os dois conjuntos de programas e a média de complexidade foi aproximadamente a mesma ( $\bar{x} = 13,2$  e  $DP = 10$ ) e ( $\bar{x} = 13,8$  e  $DP = 10,2$ ) para os paradigmas procedural e OO, respectivamente.
- A média de LOC do conjunto procedural é 60,60% maior do que a do conjunto OO.
- A média do “total de parâmetros exclusivamente do tipo primitivo” para o conjunto OO ( $\bar{x} = 1,69$  e  $DP = 2,05$ ) apresentou-se mais de 10 vezes maior do que a mesma medida para o conjunto procedural ( $\bar{x} = 0,13$  e  $DP = 0,34$ ).
- O conjunto OO apresentou média de aproximadamente 2 unidades sem qualquer parâmetro ( $\bar{x} = 1,75$ ,  $DP = 1,90$ ), enquanto que o conjunto procedural não apresentou nenhuma ocorrência de programa com pelo menos uma unidade sem parâmetro.
- A média do “total de unidades com parâmetros exclusivamente do tipo primitivo” para o conjunto OO ( $\bar{x} = 1,69$  e  $DP = 2,05$ ) apresentou-se mais de 10 vezes maior do que a mesma medida para o conjunto procedural ( $\bar{x} = 0,13$  e  $DP = 0,34$ ).

- A média do “total de unidades com parâmetros exclusivamente do tipo Abstrato” para o conjunto Procedimental ( $\bar{x} = 3$  e  $DP = 3,44$ ) apresentou-se quase duas vezes maior do que a mesma medida para o conjunto OO ( $\bar{x} = 1,18$  e  $DP = 2,05$ ).
- A média do “total de unidades com parâmetros mistos” manteve-se abaixo de 1 em ambos os conjuntos.
- A média do “total de unidades sem retorno” manteve-se próxima entre os dois conjuntos de programas, sendo ligeiramente maior no conjunto procedural ( $\bar{x} = 3,18$  e  $DP = 2,69$ ) com relação ao OO ( $\bar{x} = 2,43$  e  $DP = 2,25$ ).
- A média do “total de retornos do tipo primitivo” do conjunto OO ( $\bar{x} = 1,31$  e  $DP = 1,51$ ) é aproximadamente 2 vezes maior do que a do conjunto procedural ( $\bar{x} = 0,59$  e  $DP = 0,76$ ).
- A média do “total de retornos do tipo Abstrato” apresentou uma diferença expressiva do conjunto OO ( $\bar{x} = 1,40$  e  $DP = 1,86$ ) com relação ao procedural ( $\bar{x} = 0,25$  e  $DP = 0,56$ ), sendo o primeiro aproximadamente 5,5 vezes maior do que o segundo.

As informações obtidas, permitem verificar algumas características dos dois conjuntos amostrais. As observações de desvio e complexidade ciclomática, por exemplo, demonstram que apesar da diferença de paradigmas, a estrutura lógica dos programas mantém-se bastante semelhante entre os dois conjuntos.

Com relação aos parâmetros, pode-se notar que no conjunto de programas do paradigma OO existe um uso mais intenso da passagem de dados do tipo primitivo entre os métodos, via parâmetro, do que no conjunto do paradigma procedural. Este último, por sua vez, apresentou uma quantidade de passagem de dados do tipo abstrato maior do que o primeiro. Essas informações levantam indícios de que o paradigma OO, de fato, pode favorecer interações mais simples entre as entidades (objeto ou método) do que o paradigma procedural, possivelmente por um de seus princípios ser a propriedade de coesão das operações realizadas por cada entidade.

As informações sobre retornos dos dois conjuntos também demonstram um comportamento já conhecido dos dois paradigmas. No procedural, por exemplo, é natural que não se retornem tipos abstratos ao final de cada procedimento, já que muitas vezes, quando isso é necessário, o mesmo é associado a um ponteiro que já fôra passado previamente via parâmetro para este fim. No orientado a objetos, por sua vez, a referência, em geral, é criada dentro do próprio método (com a criação de um objeto) e retornada por este ao final da operação.

A definição do conjunto de teste inicial para comparação dos dois paradigmas, foi feita gerando-se um conjunto de teste adequado aos critérios funcionais. Na Tabela 5.4 são apresentadas as medidas “Número de Elementos Requeridos/Programa”, “Número de Casos de Teste Gerados/Programa” e “Número de Casos de Teste Que Passaram/Programa” para os critério Particionamento

Programas C (procedimental)													
Programa	Total. Und.	LOC	Desvios	Recursões	s/ Parâm.	Parâm. Prim.	Parâm. Abstr.	Parâm. Mistos	s/ Retorno	Retorn. Prim.	Retorn. Abstr.	Cplx. Cicl.	
1	1	12	2	0	0	0	1	0	0	1	0	3	
2	1	14	3	0	0	0	0	1	1	0	0	4	
3	1	14	3	0	0	0	0	1	1	0	0	4	
4	1	27	11	0	0	0	0	1	1	0	0	9	
5	1	17	3	0	0	0	1	0	1	0	0	4	
6	1	6	2	1	0	1	0	0	0	1	0	2	
7	1	9	1	0	0	1	0	0	0	1	0	2	
8	1	20	5	2	0	1	0	0	1	0	0	5	
9	2	23	4	2	0	0	0	2	2	0	0	6	
10	1	25	6	0	0	1	0	0	0	1	0	7	
11	5	44	5	0	0	0	5	0	4	1	0	10	
12	4	42	1	0	0	0	4	0	3	1	0	5	
13	5	36	4	0	0	0	5	0	3	2	0	7	
14	5	49	1	0	0	0	5	0	3	2	0	6	
15	5	39	5	0	0	0	5	0	4	1	0	8	
16	5	50	2	0	0	0	5	0	4	1	0	7	
17	9	107	24	2	0	0	9	0	9	0	0	31	
18	7	76	10	0	0	0	6	0	5	0	1	17	
19	9	117	27	3	0	0	0	9	9	0	0	35	
20	4	52	10	0	0	0	4	0	2	0	2	11	
21	7	94	20	9	0	0	6	1	7	0	0	25	
22	12	118	17	0	0	0	12	0	9	1	2	28	
23	8	98	17	0	0	0	7	1	6	2	0	28	
24	9	116	19	0	0	0	9	0	6	2	1	28	
25	2	76	6	1	0	0	0	1	2	0	0	8	
26	9	123	14	0	0	0	8	1	6	2	1	23	
27	2	76	6	1	0	0	0	1	2	0	0	8	
28	2	199	8	0	0	0	1	1	2	0	0	23	
29	3	130	10	1	0	0	2	1	2	0	1	13	
30	2	199	8	0	0	0	1	1	2	0	0	23	
31	4	68	17	0	0	0	0	4	4	0	0	24	
32	1	38	6	0	0	0	0	1	1	0	0	7	
Média	4,0625	66,1	8,656	0,688	0	0,13	3	0,84	3,1875	0,594	0,25	13,2	
dv. padrão	3,182	51,8	7,124	1,712	0	0,34	3,445	1,71	2,6933	0,756	0,568	10	

**Tabela 5.2:** Métricas de implementação para programas procedimentais.

Classe de Equivalência (PCE) e Análise Valor Limite (AVL). As medidas “Número de Elementos Requeridos/Critério”, “Número de Casos de Teste Gerados/Critério” e “Número de Casos de Teste Que Passaram/Critério” são representadas pela média e desvio padrão de cada coluna, localizadas nas últimas linhas da tabela. As colunas “Total” correspondem ao somatório das medidas nos critérios Particionamento Classe de Equivalência e Análise Valor Limite, menos as repetições existentes entre as duas medidas.

Para o conjunto de teste inicial o número de elementos requeridos e o número de casos de teste gerados no paradigma procedural é igual a do orientado a objetos, já que os mesmos são derivados em termos da especificação comum às duas implementações. Os dados coletados com essas medidas fornecem informações importante quanto ao grau de complexidade dos programas sob análise e do conjunto de teste base estabelecido para o mesmo. Desta forma, as seguintes propriedades puderam ser observadas a partir da Tabela 5.4.

- O programa 25 (*GrafoListaAdj1*) apresentou o maior número de “elementos requeridos gerados” para o critério Particionamento Classe de Equivalência (19) enquanto que o programa

Programas Java (OO)												
Programa	Total. Und.	LOC	Desvios	Recursos	s/ Parâm.	Parâm. Prim.	Parâm. Abstr.	Parâm. Mistos	s/ Retorno	Retorn. Prim.	Retorn. Abstr.	Cplx. Cicl.
1	1	8	2	0	0	0	0	1	0	0	1	3
2	1	13	3	0	0	0	1	0	0	1	0	4
3	1	13	3	0	0	0	0	1	0	1	0	4
4	1	25	11	0	0	1	0	0	0	1	0	9
5	1	14	3	0	0	0	0	1	1	0	0	4
6	1	7	2	2	0	1	0	0	0	1	0	2
7	1	11	1	0	0	1	0	0	0	1	0	2
8	1	20	8	2	0	0	1	0	0	1	0	5
9	5	22	6	2	0	0	0	2	2	0	0	8
10	1	31	6	0	0	1	0	0	1	0	0	7
11	5	28	7	0	3	0	2	0	3	1	1	13
12	4	19	1	0	3	0	1	0	2	1	1	5
13	5	23	3	0	4	0	1	0	2	1	2	7
14	5	32	1	0	4	0	1	0	1	2	2	6
15	5	29	3	0	4	0	1	0	3	1	1	8
16	5	38	2	0	4	0	1	0	2	1	2	7
17	10	86	21	2	1	1	8	0	9	0	1	32
18	8	54	10	0	4	2	1	1	5	0	3	19
19	9	88	28	3	1	1	0	7	8	0	1	35
20	4	30	8	0	0	1	3	0	2	2	0	11
21	11	74	30	11	2	0	9	0	5	0	6	29
22	10	64	12	4	3	4	1	2	3	4	3	19
23	11	72	17	0	2	5	2	2	5	4	2	29
24	9	74	14	0	3	6	0	0	2	2	5	27
25	15	73	6	0	6	8	1	0	2	6	7	24
26	13	82	17	0	6	7	0	0	4	5	4	27
27	3	36	6	1	1	1	1	0	1	1	1	9
28	6	51	9	1	1	4	1	0	4	2	0	15
29	5	51	9	1	3	1	1	0	2	1	2	14
30	6	51	9	1	1	4	1	0	4	2	0	27
31	4	47	18	0	0	4	0	0	4	0	0	25
32	1	26	6	0	0	1	0	0	1	0	0	7
Média	5,25	40,4	8,813	0,938	1,75	1,69	1,188	0,53	2,4375	1,313	1,406	13,8
dv. Padrão	3,9919	24,9	7,541	2,109	1,901	2,28	2,055	1,34	2,2567	1,512	1,864	10,2

**Tabela 5.3:** Métricas de implementação para programas orientados a objetos.

31 (*CasamentoExato*) apresentou o maior número de “elementos requeridos gerados” para o critério Análise do Valor Limite (28).

- Os programas 2 (*MaxMin1*), 4 (*MaxMin3*), 6 (*FibRec*), 7 (*FibIte*), 8 (*MaxMinRec*) e 27 (*BuscaEmProfundidade*) apresentaram o menor “número de elementos requeridos gerados” para o critério Particionamento Classe de Equivalência (1) sendo que o programa 27 também apresentou o menor “número de elementos requeridos gerados” para o critério Análise Valor Limite.
- O programa 25 (*GrafoListaAdj1*) apresentou o maior “número casos de teste gerados” para o critério Particionamento Classe de Equivalência (18) enquanto que o programa 31 (*CasamentoExato*) apresentou o maior “número de casos de teste gerados” para o critério Análise do valor limite (28).
- Os programas 1(*Max*), 2 (*MaxMin1*), 3 (*MaxMin2*), 4 (*MaxMin3*), 5 (*Ordenação1*), 6 (*FibRec*), 7 (*FibIte*), 8 (*MaxMinRec*), 9 (*Mergesort*), 10 (*AvaliaMultMatrizes*), 27 (*BuscaEm-*

Programa	Elementos Requeridos			Casos de testes gerados			Casos de teste que passaram		
	PCE	AVL	Total	PCE	AVL	Total	PCE	AVL	Total
1	2	3	3	1	3	3	1	3	3
2	1	5	5	1	5	5	1	5	5
3	3	5	5	1	5	5	1	5	5
4	1	5	5	1	5	5	1	5	5
5	2	4	4	1	3	3	1	3	3
6	1	3	3	1	3	3	1	3	3
7	1	2	2	1	2	3	1	2	3
8	1	7	7	1	7	7	1	7	7
9	4	8	8	1	7	7	1	7	7
10	3	7	7	1	5	5	1	5	5
11	7	11	11	7	11	14	7	11	14
12	7	7	7	7	7	7	7	7	7
13	8	13	13	8	13	17	8	13	17
14	9	10	10	9	10	10	9	10	10
15	7	8	8	7	14	14	3	2	5
16	9	10	10	9	10	10	9	10	10
17	10	20	20	5	15	15	5	15	15
18	13	23	23	11	21	21	11	21	21
19	10	25	25	5	25	25	5	25	25
20	7	9	9	7	9	12	7	9	12
21	12	16	16	10	14	14	10	14	14
22	13	13	13	11	11	11	11	11	11
23	13	13	13	11	11	11	11	11	11
24	13	17	17	17	18	19	15	16	17
25	19	19	19	18	18	18	17	17	17
26	16	19	19	15	18	19	15	18	19
27	1	1	1	1	1	1	1	1	1
28	3	5	5	1	5	5	1	5	5
29	4	5	5	3	4	4	1	1	1
30	4	4	4	4	4	4	4	4	4
31	8	28	28	8	28	32	8	28	32
32	2	9	9	2	9	10	2	9	10
Média	6,6875	10,4375	10,4375	5,8125	10,03125	10,59375	5,53125	9,46875	10,125
dv. Padrão	5,031754	7,102646	7,102646	5,070073	6,836734	7,347852	4,925145	6,932785	7,38678

**Tabela 5.4:** Medidas de teste para critérios funcionais.

*Profundidade*) e 28 (*BuscaEmLargura*) apresentaram o menor “número de casos de teste gerados” para o critério Particionamento Classe de Equivalência (1) sendo que o programa 27 também apresentou o menor “número de casos de teste” para o critério Análise Valor Limite.

- O programa 25 (*GrafoListaAdj1*) apresentou o maior “número casos de teste gerados” para o critério Particionamento Classe de Equivalência (18) enquanto que o programa 31 (*CasamentoExato*) apresentou o maior “número de casos de teste gerados” para o critério Análise Valor Limite (28).
- O programa 15 (*FilaArranjo*) apresentou o maior “número de casos de teste que não passaram” em relação ao total gerado, tanto no critério Particionamento classe de equivalência

(42, 85%), quanto no critério Análise Valor Limite (14, 29%)

- A média de “número de casos de teste que passaram” em relação à média de “número de casos de teste gerados”, tanto no critério Particionamento de Equivalência quanto no Análise Valor Limite, apresentaram-se semelhantes.

As observações sobre as medidas de teste funcionais, permitem algumas constatações. Em primeiro lugar é interessante notar que a ocorrência de maior valor tanto para “número de elementos requeridos gerados” quanto para “número casos de teste gerados” (programa 25) não está relacionada à ocorrência de maior “complexidade ciclomática” ou de “número de desvios”(programa 19) em nenhum dos dois paradigmas. Ampliando-se o escopo para as quatro maiores ocorrências de “número de elementos requeridos gerados” e de “número casos de teste gerados” - programas 31 (*CasamentoExato*), 19 (*OrdenaçãoParcial*), 18 (*HeapSort*) e 26 (*GrafoListaAdj2*)) - nota-se que somente o programa 19 está simultaneamente associado às quatro maiores ocorrências de maior “complexidade ciclomática”. Essa informação, mesmo não sendo inesperada é importante, pois fornece indícios de que na prática, a elevação dos custos de critérios funcionais não estão necessariamente associados aos programas de complexidade estrutural mais elevada, ressaltando a importância da complementaridade no usos das técnicas funcionais e estruturais.

Uma outra observação dos testes funcionais para o conjunto considerado é a de que a média do “número de casos de teste que passaram” ( $\bar{x} = 10,12$  e  $DP = 7,38$ ) é bastante próxima da média do “número de casos de teste gerados” ( $\bar{x} = 10,59$  e  $DP = 7,34$ ). Essa informação demonstra que o conjunto amostral considerado, pelo menos do ponto de vista de testes funcionais, quase não apresenta erros. Essa característica, contudo, já era esperada uma vez que os programas foram extraídos de um material didático para ensino, o qual utiliza originalmente os programas como exemplo aplicado da teoria. Logo, é natural que os mesmos apresentem-se corretos.

A Tabela 5.5 exibe as medidas coletadas para as métricas de teste definidas no plano, com relação aos critérios estruturais fluxo de controle. Com base nos dados coletados as seguintes observações podem ser feitas acerca das diferenças de custos considerando-se a métrica “número de elementos requeridos”:

- O programa 19 (*OrdenaçãoParcial*) gerou o maior número de elementos requeridos em ambos os paradigmas para o critério Todos-Nós, sendo que no paradigma OO o número de elementos requeridos gerados para o mesmo é 23,72% maior do que no conjunto procedural.
- A ocorrência de menor número de elementos requeridos gerados para o critério Todos-Nós para o conjunto OO é 50% maior do que a ocorrência de menor número de elementos requeridos gerados para o conjunto procedural.
- A média de elementos requeridos gerados para o critério Todos-Nós para o conjunto procedural é aproximadamente a mesma do conjunto OO, sendo a do conjunto procedural apenas 0,68% maior do que a do conjunto OO.

- A mediana de elementos requeridos gerados para o critério Todos-Nós para o conjunto procedural é próxima à do conjunto OO, sendo estas respectivamente 16, 5 e 18, 5.
- O programa 19 (*OrdenaçãoParcial*) e 31 (*CasamentoExato*) representam a maior ocorrência de elementos requeridos gerados para o critério Todas-Arestas nos paradigmas procedural e OO respectivamente, sendo que a maior ocorrência do conjunto OO é 85% maior do que a do conjunto procedural.
- A ocorrência de menor número de elementos requeridos gerados para o critério Todas-Arestas para o conjunto OO é 150% maior do que a ocorrência de menor número de elementos requeridos gerados para o conjunto procedural.
- A média de elementos requeridos gerados para o critério Todas-Arestas para o conjunto OO é 121, 78% maior do que a do conjunto procedural.
- A mediana de elementos requeridos gerados para o critério Todas-Arestas para o conjunto procedural é 7, 5 enquanto que no conjunto OO ela é 20.
- A média de elementos requeridos não executáveis foi inferior a 0, 5 nos dois paradigmas, tanto para o critério Todos-Nós quanto para o critério Todas-Arestas.

Das informações levantadas, é possível constatar que o critério Todas-Arestas demonstrou um aumento de aproximadamente 100%, do paradigma procedural para o OO, quanto ao número de elementos requeridos gerados tanto para a menor ocorrência quanto para a maior e a média. Ao investigar as possíveis causas desse motivo, foi detectado que o fator ferramenta pode estar influenciando esses resultados, uma vez que a ferramenta de teste estrutural em C aplica algumas otimizações durante o processo de geração dos elementos requeridos, o que não ocorre na ferramenta em Java. Logo, por ser uma ameaça à validade de construção dos resultados, essa métrica não deve ser usada isoladamente com propósito de comparação entre os paradigmas.

O critério Todos-Nós, por sua vez, não revelou maiores discrepâncias quanto ao número de elementos requeridos gerados, uma vez que as medidas de tendência central (média, mediana) se mostraram próximas em ambos os paradigmas. A “inesperada” semelhança nesse caso, pode estar ocorrendo devido à vantagem das medidas de LOC apresentada pelo conjunto procedural, a qual estaria “compensando” os ganhos de otimização gerados pela ferramenta nesse caso.

Os resultados levantados não permitem maiores conclusões acerca da métrica “número de elementos requeridos não-executáveis” dada a baixa incidência dos mesmos sobre os critérios analisados. Contudo, por estarem relacionados ao número de elementos requeridos gerados, os resultados dessa métrica podem também serem influenciados pelos algoritmos de otimização da ferramenta de teste estrutural em C e a mesma decisão da métrica “número de elementos requeridos gerados” deve ser considerada.

Com relação à métrica total de casos de teste gerados, as seguintes observações podem ser feitas:

Programa	Nº elem. req.	Procedimental						Todos-Nós						Orientado a Objetos						
		Todos-Nós			Todas Arestas			Todos-Nós			Todas Arestas			Todos-Nós			Todas Arestas			
		%Func. adeq.	CT's extras	Total CT's	Nº elem. ñ. exec.	%Func. adeq.	CT's extras	Nº elem. ñ. exec.												
1	7	100,00%	0	3	0	3	100,00%	0	3	0	8	75,00%	1	4	0	8	87,50%	0	4	0
2	8	100,00%	0	5	0	5	100,00%	0	5	0	10	75,00%	1	6	0	11	90,90%	0	6	0
3	9	100,00%	0	5	0	5	100,00%	0	5	0	10	80,00%	1	6	0	11	87,50%	0	6	0
4	21	90,47%	1	6	1	14	84,62%	0	6	0	21	80,95%	2	7	1	27	77,77%	0	7	2
5	9	100,00%	0	3	0	4	100,00%	0	3	0	11	81,81%	1	4	0	12	91,66%	0	4	0
6	4	100,00%	0	3	0	2	100,00%	0	3	0	6	66,66%	1	4	0	5	80,00%	0	4	0
7	5	100,00%	0	3	0	2	100,00%	0	3	0	6	66,66%	1	4	0	5	80,00%	0	4	0
8	12	100,00%	0	7	0	6	100,00%	0	7	0	13	84,61%	0	8	0	15	93,33%	0	8	0
9	10	100,00%	0	7	0	5	100,00%	0	7	0	17	88,23%	1	8	0	20	95,00%	0	8	0
10	9	100,00%	0	5	0	4	100,00%	0	5	0	17	88,23%	1	6	0	20	95,00%	0	6	0
11	11	100,00%	0	14	0	6	100,00%	0	14	0	12	100,00%	0	14	0	9	100,00%	0	14	0
12	8	100,00%	0	7	0	3	100,00%	0	7	0	10	100,00%	0	7	0	7	100,00%	0	7	0
13	13	100,00%	0	17	0	6	100,00%	0	17	0	12	100,00%	0	17	0	7	100,00%	0	17	0
14	10	100,00%	0	10	0	4	100,00%	0	10	0	10	100,00%	0	10	0	5	100,00%	0	10	0
15	11	100,00%	0	14	0	5	100,00%	0	14	0	11	100,00%	0	14	0	7	100,00%	0	14	0
16	12	100,00%	0	10	0	5	100,00%	0	10	0	10	100,00%	0	10	0	9	100,00%	0	10	0
17	48	87,50%	0	15	2	23	72,82%	0	15	2	47	95,74%	1	16	0	56	98,21%	1	16	0
18	37	91,89%	2	23	0	18	83,33%	0	23	0	36	97,22%	0	21	0	36	97,22%	0	21	0
19	73	97,26%	1	25	1	40	92,50%	0	25	1	59	96,00%	1	21	0	71	98,00%	0	21	0
20	22	90,72%	1	13	1	11	81,81%	1	13	1	20	94,45%	1	13	0	20	94,80%	1	13	0
21	33	96,96%	1	15	0	16	93,75%	0	15	0	36	97,22%	1	15	0	37	94,59%	0	15	0
22	33	96,96%	1	12	0	16	93,75%	1	13	0	27	100,00%	0	11	0	27	100,00%	0	11	0
23	40	97,50%	1	12	0	18	94,44%	0	12	0	47	76,66%	97,87	12	0	51	69,56%	94,12	14	0
24	57	100,00%	0	19	0	24	100,00%	0	19	0	47	100,00%	0	20	0	49	100,00%	0	20	0
25	60	83,33%	3	21	1	27	77,78%	0	21	0	38	97,36%	1	19	0	37	100,00%	0	19	0
26	45	95,55%	2	21	0	21	90,47%	0	21	0	47	87,23%	3	22	0	48	87,50%	0	22	0
27	17	100,00%	0	1	0	8	100,00%	0	1	0	21	100,00%	0	1	0	21	100,00%	0	1	0
28	21	100,00%	0	5	0	9	100,00%	0	5	0	29	72,41%	4	9	0	32	78,13%	0	9	0
29	29	96,55%	0	4	1	13	92,30%	0	4	1	16	100,00%	0	4	0	18	100,00%	0	4	0
30	16	100,00%	0	4	0	9	87,50%	1	4	0	23	100,00%	1	2	0	26	96,15%	1	3	0
31	51	100,00%	0	32	0	24	100,00%	0	32	0	57	100,00%	1	33	0	74	100,00%	0	33	0
32	18	100,00%	0	10	0	7	100,00%	0	10	0	20	90,00%	1	11	0	24	95,83%	0	11	0
Média	23,72	97,65%	0,41	10,97	0,22	11,34	95,16%	0,09	11,00	0,16	23,56	90,36%	3,84	11,22	0,03	25,16	93,40%	3,04	11,31	0,06
dv. Padrão	18,57	4,26%	0,76	7,64	0,49	9,09	7,62%	0,30	7,65	0,45	15,85	11,02%	17,18	7,20	0,18	19,26	8,27%	16,62	7,17	0,35

Tabela 5.5: Métricas de teste estrutural fluxo de controle.

- Tanto a média da métrica “total de casos de teste gerados” quanto o seu desvio padrão apresentaram-se bastante semelhantes para o critério Todos-Nós e para o critério Todas-Arestas em ambos os paradigmas - Procedimental:  $\bar{x} = 10,96$  e  $DP = 7,64$  para o critério Todos-Nós e  $\bar{x} = 11$  e  $DP = 7,64$  para critério Todas-Arestas; OO:  $\bar{x} = 11,21$  e  $DP = 7,19$  para o critério Todos-Nós e  $\bar{x} = 11,31$  e  $DP = 7,17$  para o critério Todas-Arestas.
- A ocorrência de maior valor para “total de casos de teste gerados” nos critérios “Todos-Nós” e “Todas-Arestas” no paradigma Procedimental foi de 32 em ambos os casos e no paradigma OO de 33, também em ambos os casos.
- A ocorrência de menor valor para “total de casos de teste gerados” nos critérios “Todos-Nós” e “Todas-Arestas” nos paradigmas Procedimental e OO foi de 1.
- A mediana de “total de casos de teste gerados” para o critério Todos-Nós e Todas-Aresta foi a mesma (10) em ambos os paradigmas.

Com base nas observações iniciais feitas sobre o total de casos de teste gerados não é possível detectar grandes discrepâncias de custo entre os paradigmas com relação a esses critérios, uma vez que as medidas de tendência central se mostraram próximas em ambos os casos. Essa informação é condizente com as observações de complexidade ciclomática e número de desvios, as quais também apresentaram-se próximas em ambos os paradigmas.

A Tabela 5.6 exibe as medidas coletadas para as métricas de teste definidas no plano, com relação aos critérios estruturais fluxo de dados. Com base nos dados coletados as seguintes observações podem ser feitas acerca das diferenças de custo considerando-se a métrica “número de elementos requeridos”:

- O programa 19 (*Ordenação Parcial*) gerou o maior número de elementos requeridos em ambos os paradigmas para o critério Todos-Usos, sendo que no paradigma OO o número de elementos requeridos gerados para o mesmo é 3,27% maior do que no conjunto procedural.
- A ocorrência de menor número de elementos requeridos gerados para o critério Todos-Usos para o conjunto OO é 175% maior do que a ocorrência de menor número de elementos requeridos gerados para o conjunto procedural.
- A média de elementos requeridos gerados para o critério Todos-Usos para o conjunto procedural apresentou uma diferença relevante com relação à do conjunto OO, sendo a do conjunto OO 42,78% maior do que a do conjunto procedural.
- A mediana de elementos requeridos gerados para o critério Todos-Usos apresentou uma diferença relevante com relação aos paradigmas, sendo as mesmas 50 e 77 para os paradigmas procedural e OO, respectivamente.

Programa	Procedimental												Orientado a Objetos											
	Todos-Uso						Todos Pot-Uso						Todos-Uso						Todos Pot-Uso					
	Nº elem. req.	%Flx. adeq.	CT's extras	Total CT's	Nº elem. ñ. exec.	Nº elem. req.	%Flx. Ctrl.	CT's Total	Nº elem. ñ. exec.	%Flx. Ctrl.	CT's Total	Nº elem. req.	%Flx. Ctrl.	CT's Total	Nº elem. ñ. exec.	%Flx. Ctrl.	CT's Total	Nº elem. adeq.	%Flx. Ctrl.	CT's Total	Nº elem. ñ. exec.	%Flx. Ctrl.	CT's Total	
1	21	95,23%	0	3	1	15	93,33%	0	3	1	25	96,00%	0	4	1	41	95,12%	0	4	2	95,12%	0	4	2
2	30	96,67%	0	5	1	24	95,00%	0	5	1	43	97,00%	0	5	1	78	97,43%	0	5	2	97,43%	0	5	2
3	30	93,33%	0	5	2	17	90,00%	0	5	2	43	95,35%	0	6	2	78	96,15%	0	6	2	96,15%	0	6	2
4	91	71,42%	1	6	24	80	70,00%	0	6	22	119	62,00%	2	8	36	303	66,33%	0	8	83	66,33%	0	8	83
5	33	93,95%	0	3	2	40	85,00%	0	3	6	33	93,33%	0	3	2	40	85,00%	0	3	6	85,00%	0	3	6
6	4	100,00%	0	3	0	2	100,00%	0	3	0	11	100,00%	0	4	0	26	100,00%	0	4	0	100,00%	0	4	0
7	14	85,71%	0	3	0	15	83,33%	0	3	0	11	100,00%	0	4	0	26	100,00%	0	4	0	100,00%	0	4	0
8	27	100,00%	0	7	0	18	100,00%	0	7	0	52	100,00%	0	8	0	121	100,00%	0	8	0	100,00%	0	8	0
9	40	95,00%	0	7	2	24	91,66%	0	7	3	77	92,21%	0	8	6	227	88,54%	0	8	26	88,54%	0	8	26
10	33	93,93%	0	5	2	40	85,00%	0	5	6	77	92,20%	0	6	6	237	86,00%	0	6	31	86,00%	0	6	31
11	13	100,00%	0	14	0	12	100,00%	0	14	0	30	100,00%	0	14	0	24	100,00%	0	14	0	100,00%	0	14	0
12	6	100,00%	0	7	0	10	100,00%	0	7	0	15	100,00%	0	7	0	16	100,00%	0	7	0	100,00%	0	7	0
13	8	100,00%	0	17	0	9	100,00%	0	17	0	19	100,00%	0	17	0	17	100,00%	0	17	0	100,00%	0	17	0
14	4	100,00%	0	10	0	7	100,00%	0	10	0	11	100,00%	0	10	0	9	100,00%	0	10	0	100,00%	0	10	0
15	8	100,00%	0	14	0	8	100,00%	0	14	0	25	100,00%	0	14	0	17	100,00%	0	14	0	100,00%	0	14	0
16	10	100,00%	0	10	0	13	100,00%	0	10	0	22	100,00%	0	10	0	24	100,00%	0	10	0	100,00%	0	10	0
17	149	93,29%	4	19	5	180	89,44%	0	19	12	203	93,10%	5	21	10	473	91,97%	0	21	28	91,97%	0	21	28
18	81	87,65%	4	27	3	59	76,27%	0	27	8	135	87,65%	4	27	3	59	76,27%	0	27	8	76,27%	0	27	8
19	275	91,63%	2	25	19	285	81,75%	1	25	40	284	90,84%	0	21	25	695	91,65%	1	22	51	91,65%	1	22	51
20	49	91,81%	3	15	0	43	91,02%	3	15	0	80	85,58%	5	16	0	118	77,00%	5	16	3	77,00%	5	16	3
21	59	100,00%	0	15	0	20	100,00%	0	15	0	106	100,00%	0	15	0	174	100,00%	0	15	0	100,00%	0	15	0
22	54	92,59%	1	14	2	35	91,42%	0	14	2	106	96,22%	2	12	3	134	95,52%	1	12	3	95,52%	1	12	3
23	103	93,20%	2	14	4	83	85,54%	0	14	6	225	95,11%	3	17	8	334	92,21%	0	17	17	92,21%	0	17	17
24	136	88,97%	0	19	15	126	76,98%	0	19	29	192	93,00%	1	21	9	258	86,00%	0	20	28	86,00%	0	20	28
25	123	82,92%	10	31	7	143	77,62%	1	32	14	126	93,65%	2	21	4	207	87,92%	0	21	17	87,92%	0	21	17
26	93	81,72%	4	25	6	89	76,40%	0	25	15	211	93,84%	3	25	5	305	89,51%	0	25	20	89,51%	0	25	20
27	51	94,12%	1	2	1	62	95,16%	0	2	1	80	95,00%	1	2	1	173	93,06%	0	2	1	93,06%	0	2	1
28	51	94,12%	0	5	3	89	96,63%	0	5	3	111	96,40%	0	9	4	273	92,67%	0	9	18	92,67%	0	9	18
29	90	86,66%	1	5	10	114	88,59%	0	5	9	50	96,00%	1	5	1	100	95,00%	0	5	2	95,00%	0	5	2
30	71	92,95%	0	4	4	161	93,17%	0	4	11	90	94,44%	1	8	4	309	93,03%	0	31	18	93,03%	0	31	18
31	187	93,05%	9	40	3	277	94,22%	4	40	4	205	90,24%	9	42	1	779	90,63%	0	42	2	90,63%	0	42	2
32	80	82,50%	4	14	1	144	87,50%	1	15	1	73	90,00%	4	15	1	354	85,87%	1	16	4	85,87%	1	16	4
Média	63,25	92,89%	1,44	12,28	3,66	70,13	90,47%	0,31	12,34	6,13	90,31	94,37%	1,34	12,66	4,16	188,41	92,28%	0,25	13,41	11,63	92,28%	0,25	13,41	11,63
d.v. Padrão	60,496	6,75%	2,54	9,36	5,76	75,12	8,65%	0,90	9,44	9,35	73,97	7,12%	2,12	8,70	7,60	190,55	8,12%	0,92	9,25	18,10	8,12%	0,92	9,25	18,10

**Tabela 5.6:** Métricas de teste estrutural fluxo de dados.

- O programa 19 (*OrdenaçãoParcial*) e 31(*CasamentoExato*) representam a maior ocorrência de elementos requeridos gerados para o critério Todos-Potenciais-Usos nos paradigmas procedural e OO respectivamente, sendo que a maior ocorrência do conjunto OO é 173,33% maior do que a do conjunto procedural.
- A ocorrência de menor número de elementos requeridos gerados para o critério Todos-Potenciais-Usos para o conjunto OO é três vezes e meia (350%) maior do que a ocorrência de menor número de elementos requeridos gerados para o conjunto procedural.
- A média de elementos requeridos gerados para o critério Todos-Potenciais-Usos para o conjunto OO é 168,68% maior do que a do conjunto procedural.
- A mediana de elementos requeridos gerados para o critério Todos-Potenciais-Usos apresentou uma diferença relevante com relação aos paradigmas, sendo as mesmas 40 e 127,5 para o conjunto procedural e OO, respectivamente.
- A média de elementos requeridos não executáveis para o critério Todos-Usos apresentou-se semelhante entre os paradigmas.
- A média de elementos requeridos não executáveis para o critério Todos-Potenciais-Usos apresentou uma diferença expressiva entre os paradigmas, sendo que no paradigma OO a mesma é 89,79% maior do que no paradigma procedural.

Novamente, as discrepâncias expressivas entre os valores observados com relação às métrica “Elementos Requeridos Gerados” e “Elementos Requeridos Não-Executáveis”, do paradigma procedural para o OO, podem estar sendo influenciadas pelas otimizações aplicadas pela ferramenta de teste estrutural do paradigma procedural, favorecendo a redução das medidas nas mesmas.

Com relação à métrica “total de casos de teste gerados” as seguintes observações podem ser feitas:

- Tanto a média da métrica “total de casos de teste gerados” quanto o seu desvio padrão apresentaram-se bastante semelhantes entre o critério Todos-Usos e o critério Todas-Potenciais-Usos em ambos os paradigmas.
- A ocorrência de maior valor para “total de casos de teste gerados” nos critérios “Todos-Usos” e “Todos-Potenciais-Usos” no paradigma Procedimental foi de 40 e no paradigma OO de 42.
- A ocorrência de menor valor para “total de casos de teste gerados” nos critérios “Todos-Usos” e “Todos-Potenciais-Usos” nos paradigmas Procedimental e OO foi de 2.
- A mediana de “total de casos de teste gerados” para o critério “Todos-Usos” foi a mesma (10) em ambos os paradigmas.

- A mediana de “total de casos de teste gerados” para o critério “Todos-Potenciais-Usos” apresentou-se próxima em ambos os paradigmas (10 no paradigma procedural e 11 no OO).

Com base nas observações feitas sobre o total de casos de teste gerados para os critérios de fluxo de dados, não é possível detectar grandes discrepâncias entre os paradigmas com relação a esses critérios, uma vez que as medidas de tendência central e dos valores máximo e mínimo, se mostraram próximos em ambos os paradigmas.

As últimas medidas coletadas, relacionam-se à verificação da segunda hipótese, ou seja, à comparação do *strength* entre paradigmas. Os valores coletados constam na Tabela 5.7. A análise dos dados permite as seguintes observações:

- Nos critérios estruturais fluxo de controle, os conjuntos de teste OO-adequados apresentaram uma cobertura sobre os programas procedimentais ( $\bar{x} = 96\%$  e  $DP = 0,09$ ) relativamente maior do que os conjuntos de teste procedimentais-adequados sobre os programas OO ( $\bar{x} = 93\%$  e  $DP = 0,09$ ).
- Nos critérios estruturais fluxo de dados, ocorreu o inverso. Os conjuntos de teste procedimentais-adequados apresentaram uma cobertura sobre os programas OO ( $\bar{x} = 93\%$  e  $DP = 0,08$ ) relativamente maior do que os conjuntos de teste OO-adequados sobre os programas procedimentais ( $\bar{x} = 89\%$  e  $DP = 0,12$ ).
- A menor ocorrência de cobertura de um critério sobre um programa se deu no paradigma procedural (critério Todos-Potenciais-Usos, 33%) sendo a mesma 50% menor do que no paradigma oposto.
- Todos os critérios apresentaram ocorrências com cobertura igual a 100%, em ambos os paradigmas.

Com base nas observações feitas sobre o *strength* entre paradigmas é possível notar que: os critérios estruturais fluxo de controle apresentaram em média um *strength* ligeiramente maior no paradigma OO enquanto que os critérios estruturais fluxo de dados apresentaram em média um *strength* ligeiramente maior no paradigma procedural.

A verificação dos casos de teste não implementáveis, após conversão dos casos de teste entre paradigmas, fornece possíveis indícios para o aumento do *strength* a favor do paradigma procedural, com relação aos critérios de fluxo de dados. A conversão de casos de teste compostos somente de estruturas OO não pôde ser reproduzida no paradigma procedural, obscurecendo o exercício de associações e potenciais-associações que normalmente estão relacionadas às mesmas no caso de teste original. Um exemplo de caso de teste que não pode ser convertido é apresentado na Figura 5.4. Como pode-se observar, o caso de teste é composto de somente uma instrução, que

corresponde à instanciação do objeto “Max”. Contudo, não existe instrução na linguagem procedural capaz de representar tal funcionalidade, tanto do ponto de vista sintático quanto semântico. Portanto, este caso de teste é considerado não implementável na linguagem do paradigma oposto.

```

64
65     public void testFC1() {
66         Max mx = new Max();
67     }
68

```

**Figura 5.4:** Trecho de código contendo caso de teste não implementável na linguagem do paradigma oposto

### 5.3.2 Teste de Hipóteses

As informações obtidas até esse ponto, com base na análise “crua” dos dados e de suas medidas de tendência central, fornecem apenas indícios pontuais acerca das hipóteses investigadas.

Ainda que este estudo tenha a finalidade de caracterização do tema proposto e não se configure como um experimento controlado, verificou-se que seria prudente a aplicação de métodos de análise que fornecessem maior respaldo estatístico às conclusões obtidas. Para tanto, elegeu-se a aplicação de testes de hipóteses sobre a amostra coletada. É importante ressaltar que alguns requisitos de validade para aplicação desses testes foram pormenorizados e que portanto, a aplicação dos mesmos tem um caráter mais pragmático do que conclusivo assim como a própria definição formal das hipóteses no plano.

A aplicação dos testes de hipóteses se deu em duas etapas. Na primeira etapa, a primeira hipótese do plano (relacionada ao custo de teste) foi verificada com base na métrica de custo “Número de Casos de Teste Gerados/Programa”. Na segunda etapa, foi verificada a segunda hipótese (relacionada ao *strength*), com relação à métrica “Porcentagem de Cobertura/Programa no paradigma oposto”.

A verificação dos testes foi realizada com o auxílio da ferramenta de análise estatística *SPSS Statistics* versão 17.0 (Inc., 2009).

#### Teste de Hipótese 1

O primeiro teste de hipótese tenta refutar a primeira hipótese nula definida no plano, com base nas métricas “Número de Casos de Teste Gerados/Programa”.

Por se tratar de uma amostra pareada, a aplicação do teste estatístico deverá ser realizada em termos das diferenças das medidas pareadas de cada ensaio da amostra. Assim, seja  $X_1$  e  $X_2$  as variáveis que indicam as duas condições de medida (procedimental e OO, respectivamente), a variável dependente  $D$  é definida como:

$$D = X_2 - X_1$$

Programa	OO-adequado -> Procedimental					Procedimental-adequado -> OO				
	Todos Nós	Todas Arestas	Todos Usos	Todos Pot. Usos	Não Implem.	Todos Nós	Todas Arestas	Todos Usos	Todos Pot. Usos	Não Implem.
1	100%	100%	95,23%	93,33%	1	75%	87,50%	100%	97,56%	0
2	100%	100%	96,67%	95,00%	1	80%	90,90%	100%	98,72%	0
3	100%	100%	93,33%	90%	1	80%	90,90%	100%	98,72%	0
4	95,24%	92,31%	75,83%	72,50%	1	85,71%	88,88%	63%	66%	0
5	100%	100%	93,93%	85%	1	81,81%	91,66%	93,94%	92,30%	0
6	100%	100%	100%	100%	1	66,66%	80%	100%	96,15%	0
7	100%	100%	85,71%	83,33%	1	66,66%	80%	100%	96,15%	0
8	100%	100%	100%	100%	1	84,62%	93,33%	100%	97,52%	0
9	100%	100%	95%	91,66%	1	88,23%	95%	92,20%	88,11%	0
10	100%	100%	93,93%	85%	1	88,23%	95%	92,21%	86,50%	0
11	100%	100%	100%	100%	0	100%	100%	100%	100%	0
12	100%	100%	100%	100%	0	100%	100%	100%	100%	0
13	100%	100%	100%	100%	0	100%	100%	100%	100%	0
14	100%	100%	100%	100%	0	100%	100%	100%	100%	0
15	100%	100%	100%	100%	0	100%	100%	100%	100%	0
16	100%	100%	100%	100%	0	100%	100%	100%	100%	0
17	87,50%	72,82%	93,29%	89,44%	1	95,74%	98,22%	93,10%	92,39%	0
18	91,89%	83,33%	87,65%	76,27%	0	97,33%	65,78%	95,55%	95,50%	0
19	100%	100%	86,22%	81,66%	1	96,61%	98,59%	90,84%	91,79%	0
20	95,45%	90,90%	93,87%	81,40%	0	100%	100%	95%	88,16%	0
21	100%	100%	100%	100%	0	89,79%	86,54%	83,12%	91,11%	0
22	96,96%	93,75%	88,88%	85,71%	1	96,77%	93,55%	94,06%	92,50%	0
23	100%	100%	94,59%	91,43%	0	78,33%	72,60%	68,89%	70,25%	0
24	100%	100%	88,97%	76,98%	0	100%	100%	93%	86%	0
25	96,11%	55%	46,06%	33,33%	0	97,36%	100%	94,44%	88,54%	4
26	100%	100%	88,17%	87,64%	0	95,74%	95,83%	96,27%	91,80%	0
27	100%	100%	94,12%	95,16%	1	100%	100%	97,50%	97,11%	0
28	100%	100%	100%	100%	4	72,41%	78,12%	75,67%	84,61%	0
29	96,55%	92,30%	87,77%	90,35%	0	100%	100%	96%	96%	0
30	100%	87,50%	94,37%	93,17%	1	95,65%	100%	95,55%	92,88%	0
31	100%	100%	92,51%	93,86%	0	96,61%	98,66%	97,07%	97,30%	0
32	100%	100%	90%	90,97%	0	90%	95,83%	98,63%	99,43%	0
Média	99%	96%	92%	89%	1,19	91%	93%	94%	93%	0
dv. Padrão	0,03	0,10	0,10	0,13	0,75	0,10	0,09	0,09	0,08	0

**Tabela 5.7:** Strength dos critérios entre paradigmas. Nas colunas da esquerda, a cobertura refere-se ao conjunto OO-adequado sobre os programas procedimentais. Na coluna da direita, a cobertura refere-se ao conjunto procedural-adequado sobre os programas OO.

Como o design experimental encontra-se aninhado com relação aos critérios de teste, dois níveis de amostras pareadas foram considerados: *a* nível fluxo de controle, baseado nas medidas coletadas para o critério Todas-Arestas e *b* nível fluxo de dados, baseado nas medidas coletadas para o critério Todos-Potenciais-Usos. Esses critérios foram eleitos pois são, em seus respectivos níveis, os critérios superiores na hierarquia de inclusão entre critérios, conforme observado na Figura 2.1 do Capítulo 2. Para o nível *a* estabelece-se que a variável dependente  $D_a$  é definida

como:

$D_a = X_2 - X_1$  tal que,  $X_1$  e  $X_2$  representam medidas pareadas para cada ensaio da amostra no nível fluxo de controle.

Para o nível  $b$  estabelece-se que a variável dependente  $D_b$  é definida como:

$D_b = X_2 - X_1$  tal que,  $X_1$  e  $X_2$  representam medidas pareadas para cada ensaio da amostra no nível fluxo de dados.

Antes que fosse determinado o tipo teste de hipótese adequado (paramétrico ou não paramétrico), fez-se necessário conhecer a distribuição dos dados da amostra. Como a amostra consta de 32 valores pareados ( $n = 32$ ), o teste *Shapiro-Wilk* (Conover, 1998) foi utilizado para verificação da condição de normalidade da distribuição amostral.

Os seguintes parâmetros foram definidos para verificação da normalidade da distribuição de  $D_a$ :

- $H_0 = D_a$  tem distribuição normal.
- $H_1 = D_a$  não tem distribuição normal.
- Nível de significância ( $\alpha$ ) = 0,05.

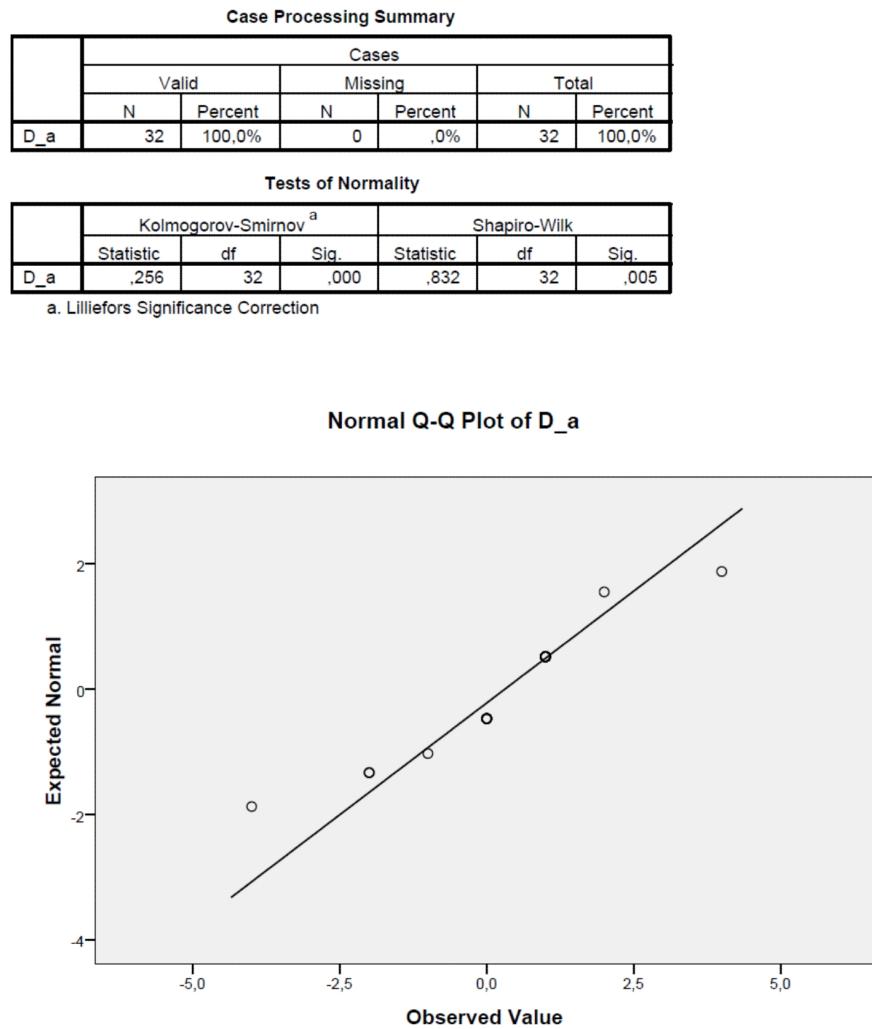
O resultado do teste de normalidade sobre a variável  $D_a$  é exibido na Figura 5.5. Conforme pode-se observar, o resultado gerado para o teste de Shapiro-Wilk, apresentou probabilidade de significância (p-value)  $p = 0,005$ . Como  $p < \alpha$ , rejeita-se a hipótese nula de que  $D_a$  tem distribuição normal, ao nível de 5%.

Com relação à distribuição de  $D_b$ , os seguintes parâmetros foram definidos para verificação da normalidade:

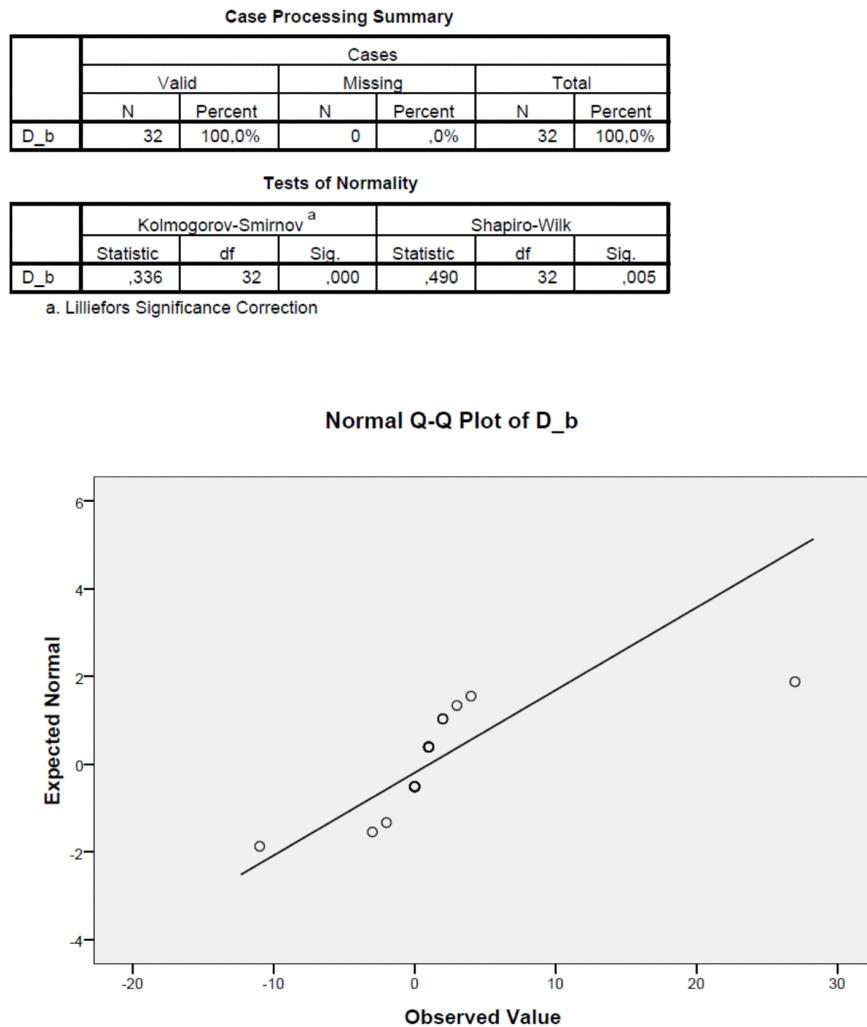
- $H_0 = D_b$  tem distribuição normal.
- $H_1 = D_b$  não tem distribuição normal.
- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste de normalidade sobre a variável  $D_b$  é exibido na Figura 5.6. Conforme pode-se observar, o resultado gerado para o teste de Shapiro-Wilk, apresentou probabilidade de significância (p-value)  $p = 0,005$ . Como  $p < \alpha$ , rejeita-se a hipótese nula de que  $D_b$  tem distribuição normal, ao nível de 5%.

Verificada a não normalidade da distribuição das variáveis, elegeu-se a aplicação do teste não-paramétrico de Wilcoxon (*Signed-Rank*) (Hollander e Wolfe, 1999) para verificação da primeira



**Figura 5.5:** Resultados do teste de normalidade sobre a variável  $D_a$  e gráfico  $q-q$  plot correspondente



**Figura 5.6:** Resultados do teste de normalidade sobre a variável  $D_b$  e gráfico  $q-q$  plot correspondente

hipótese do plano. O teste de Wilcoxon faz uso do sinal e da magnitude dos escores das diferenças entre pares de medidas, provendo uma alternativa ao teste  $t$  pareado quando a distribuição de diferenças de uma população não é normal.

O teste de Wilcoxon requer que a distribuição de diferenças da população seja simétrica sobre o valor desconhecido  $M$  da mediana. Desta forma, seja  $\mu_D = 0$  o valor hipotético especificado de  $M$ . O teste avalia as distâncias das distribuições de diferenças ( $X_2 - X_1$ ) em relação a  $\mu_D$ .

Em razão da separação de níveis de critérios, determinada pelo *design* experimental, a avaliação das hipóteses nula e alternativa foi subdividida em dois níveis também, sendo que a rejeição da hipótese nula do plano só poderá ser dada mediante a rejeição da hipótese nula em ambos os níveis.

Para a aplicação do teste no nível  $a$  (fluxo de controle), os seguintes parâmetros foram definidos:

- $H_{0a}$  = Não existe diferença de custo do critério de teste fluxo de controle, considerando-se a variável “total de casos de teste gerados”, em razão do paradigma. ( $H_{0a} : \mu_{Da} = 0$ )
- $H_{1a}$  = Existe diferença de custo do critério de teste fluxo de controle, considerando-se a variável “total de casos de teste gerados”, em razão do paradigma. ( $H_{1a} : \mu_{Da} \neq 0$ )
- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste é exibido na Figura 5.7. Conforme pode-se observar, o resultado gerado para o teste de Wilcoxon apresentou probabilidade de significância (p-value)  $p = 0,163$ . Como  $p > \alpha$ , não é possível rejeitar a hipótese nula  $H_{0a}$  de que  $\mu_{Da} = 0$ , ao nível de 5%.

Para a aplicação do teste no nível  $b$  (fluxo de dados), os seguintes parâmetros foram definidos:

- $H_{0b}$  = Não existe diferença de custo do critério de teste fluxo de dados, considerando-se a variável “total de casos de teste gerados”, em razão do paradigma. ( $H_{0b} : \mu_{Db} = 0$ )
- $H_{1b}$  = Existe diferença de custo do critério de teste fluxo de dados, considerando-se a variável “total de casos de teste gerados”, em razão do paradigma. ( $H_{1b} : \mu_{Db} \neq 0$ )
- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste sobre a variável  $\mu_{Db}$  é exibido na Figura 5.8. Conforme pode-se observar, o resultado gerado para o teste de Wilcoxon apresentou probabilidade de significância (p-value)  $p = 0,045$ . Como  $p < \alpha$ , rejeita-se a hipótese nula  $H_{0b}$  de que  $\mu_{Db} = 0$ , ao nível de 5%.

Os resultados das duas etapas do teste de hipótese para a primeira hipótese do plano apresentaram-se discrepantes pois no nível de fluxo de controle a hipótese nula não pôde ser rejeitada, o que por sua vez também não fornece evidências para aceitá-la. No nível de fluxo de dados, todavia, a hipótese nula é rejeitada, fornecendo evidências para aceitação da hipótese alternativa. Apesar do

Descriptive Statistics					
	N	Mean	Std. Deviation	Minimum	Maximum
X1	32	11,00	7,650	1	32
X2	32	11,31	7,173	1	33

Wilcoxon Signed Ranks Test					
Ranks					
	N	Mean Rank	Sum of Ranks		
X2 - X1	Negative Ranks	5 <sup>a</sup>	17,10	85,50	
	Positive Ranks	17 <sup>b</sup>	9,85	167,50	
	Ties	10 <sup>c</sup>			
	Total	32			

a. X2 < X1  
 b. X2 > X1  
 c. X2 = X1

Test Statistics <sup>b</sup>					
	X2 - X1				
Z	-1,396 <sup>a</sup>				
Asymp. Sig. (2-tailed)	,163				

a. Based on negative ranks.  
 b. Wilcoxon Signed Ranks Test

**Figura 5.7:** Resultados do teste de Wilcoxon (*Signed-Rank*) para  $H_{0a}$

Descriptive Statistics					
	N	Mean	Std. Deviation	Minimum	Maximum
X1	32	12,34	9,438	2	40
X2	32	13,41	9,245	2	42

Wilcoxon Signed Ranks Test					
Ranks					
	N	Mean Rank	Sum of Ranks		
X2 - X1	Negative Ranks	3 <sup>a</sup>	15,33	46,00	
	Positive Ranks	16 <sup>b</sup>	9,00	144,00	
	Ties	13 <sup>c</sup>			
	Total	32			

a. X2 < X1  
 b. X2 > X1  
 c. X2 = X1

Test Statistics <sup>b</sup>					
	X2 - X1				
Z	-2,008 <sup>a</sup>				
Asymp. Sig. (2-tailed)	,045				

a. Based on negative ranks.  
 b. Wilcoxon Signed Ranks Test

**Figura 5.8:** Resultados do teste de Wilcoxon (*Signed-Rank*) para  $H_{0b}$

teste de hipótese ser um teste rigoroso e não ser impossível a concomitância de resultados distintos para os dois níveis de critérios, ao se analisar os testes de hipóteses em conjunto com a primeira análise pura dos dados, é possível identificar que os resultados, de uma maneira geral, tendem à não rejeição da hipótese nula para ambos os casos. Além disso nota-se que a probabilidade de significância, que permite rejeitar a hipótese nula na segunda etapa da aplicação do teste de hipótese, apesar de ser menor que o nível de significância, apresenta-se bastante próxima ( $0,045 \approx 0,05$ ). Logo, se o nível de significância fosse reduzido de 1% essa etapa do teste também não rejeitaria a hipótese nula.

## Teste de Hipótese 2

O segundo teste de hipótese tenta refutar a segunda hipótese nula definida no plano, relacionada à diferença de *strength* entre paradigmas com base na métrica “Porcentagem de Cobertura-/Programa no paradigma oposto”.

Por se tratar de uma amostra pareada, a aplicação do teste estatístico deverá ser realizada em termos das diferenças das medidas pareadas de cada ensaio da amostra. Assim, seja  $X_1$  e  $X_2$  as variáveis que indicam as duas condições de medida (*strength* do critério procedural em relação ao conjunto OO-adequado e *strength* do critério OO em relação ao conjunto procedural-adequado, respectivamente), a variável dependente  $D$  é definida como:

$$D = X_2 - X_1$$

Como o design experimental encontra-se aninhado com relação aos critérios de teste, dois níveis de amostras pareadas foram considerados: *a* nível fluxo de controle, baseado nas medidas coletadas para o critério Todas-Arestas e *b* nível fluxo de dados, baseado nas medidas coletadas para o critério Todos-Potenciais-Usos. Para o nível *a* estabelece-se que a variável dependente  $D_a$  é definida como:

$D_a = X_2 - X_1$  tal que,  $X_1$  e  $X_2$  representam medidas pareadas para cada ensaio da amostra no nível fluxo de controle.

Para o nível *b* estabelece-se que a variável dependente  $D_b$  é definida como:

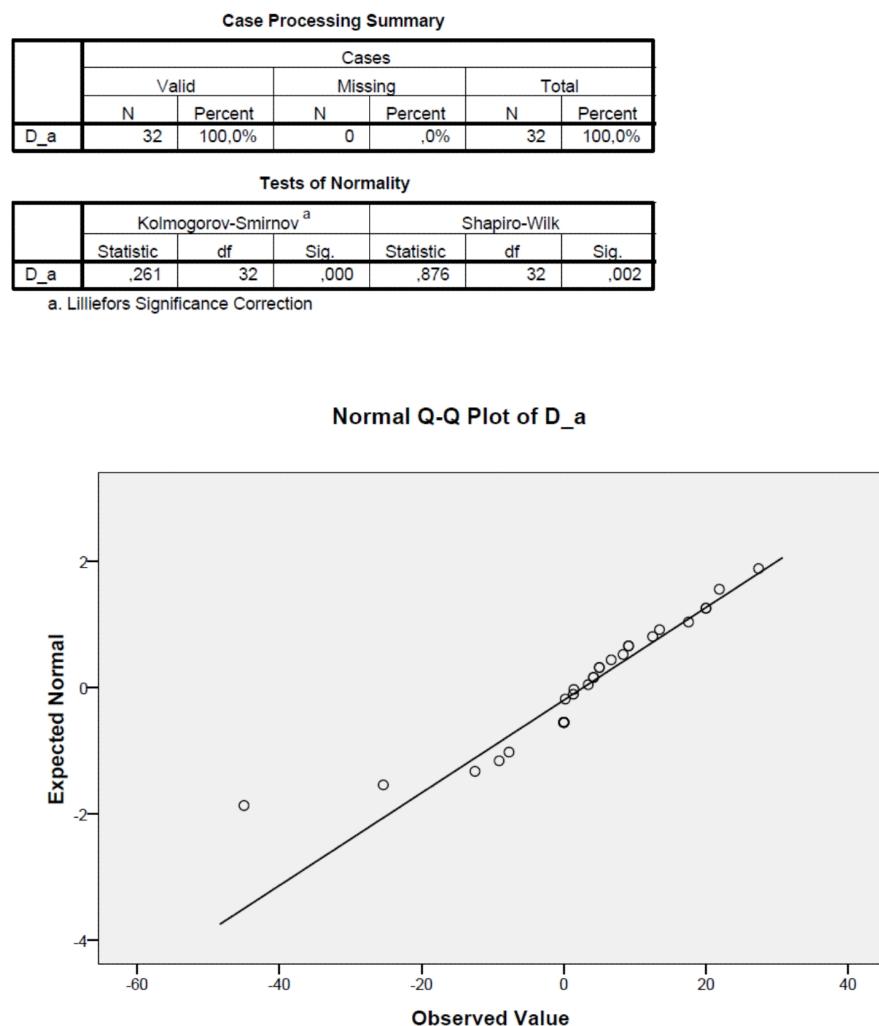
$D_b = X_2 - X_1$  tal que,  $X_1$  e  $X_2$  representam medidas pareadas para cada ensaio da amostra no nível fluxo de dados.

Antes que fosse determinado o tipo teste de hipótese adequado (paramétrico ou não paramétrico), fez-se necessário conhecer a distribuição dos dados da amostra. Como a amostra consta de 32 valores pareados ( $n = 32$ ), o teste *Shapiro-Wilk* foi utilizado para verificação da condição de normalidade da distribuição amostral.

Os seguintes parâmetros foram definidos para verificação da normalidade da distribuição de  $D_a$ :

- $H_0 = D_a$  tem distribuição normal.
- $H_1 = D_a$  não tem distribuição normal.
- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste de normalidade sobre a variável  $D_a$  é exibido na Figura 5.9. Conforme pode-se observar, o resultado gerado para o teste de Shapiro-Wilk, apresentou probabilidade de significância (p-value)  $p = 0,002$ . Como  $p < \alpha$ , rejeita-se a hipótese nula de que  $D_a$  tem distribuição normal, ao nível de 5%.



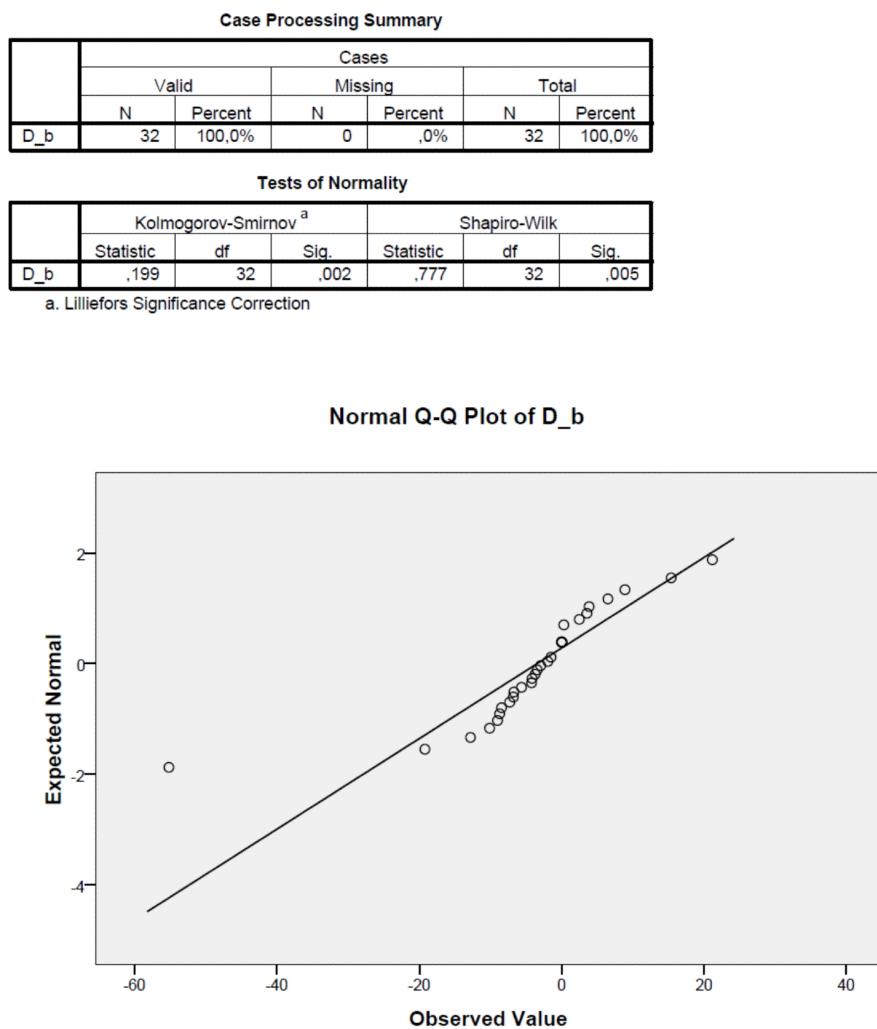
**Figura 5.9:** Resultados do teste de normalidade sobre a variável  $D_a$  e gráfico *q-q plot* correspondente

Com relação à distribuição de  $D_b$ , os seguintes parâmetros foram definidos para verificação da normalidade:

- $H_0 = D_b$  tem distribuição normal.

- $H_1 = D_b$  não tem distribuição normal.
- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste de normalidade sobre a variável  $D_b$  é exibido na Figura 5.10. Conforme pode-se observar, o resultado gerado para o teste de Shapiro-Wilk, apresentou probabilidade de significância (p-value)  $p = 0,005$ . Como  $p < \alpha$ , rejeita-se a hipótese nula de que  $D_b$  tem distribuição normal, ao nível de 5%.



**Figura 5.10:** Resultados do teste de normalidade sobre a variável  $D_b$  e gráfico *q-q plot* correspondente

Em razão da separação de níveis de critérios, determinada pelo *design* experimental, a avaliação das hipóteses nula e alternativa foi subdividida em dois níveis também, sendo que a rejeição da hipótese nula do plano só poderá ser dada mediante a rejeição da hipótese nula em ambos os níveis.

Para a aplicação do teste no nível  $a$  (fluxo de controle), os seguintes parâmetros foram definidos:

- $H_0a$  = Não existe diferença de *strength* do critério de teste fluxo de controle, considerando-se a variável “Porcentagem de Cobertura/Programa no paradigma oposto”, em razão do paradigma. ( $H_0a : \mu_{Da} = 0$ )
- $H_1a$  = Existe diferença de *strength* do critério de teste fluxo de controle, considerando-se a variável “Porcentagem de Cobertura/Programa no paradigma oposto”, em razão do paradigma. ( $H_1a : \mu_{Da} \neq 0$ )
- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste é exibido na Figura 5.11. Conforme pode-se observar, o resultado gerado para o teste de Wilcoxon apresentou probabilidade de significância (p-value)  $p = 0,061$ . Como  $p > \alpha$ , não é possível rejeitar a hipótese nula  $H_0a$  de que  $\mu_{Da} = 0$ , ao nível de 5%.

Descriptive Statistics					
	N	Mean	Std. Deviation	Minimum	Maximum
X1	32	95,8722%	9,66027%	55,00%	100,00%
X2	32	93,0278%	9,01435%	65,78%	100,00%

Wilcoxon Signed Ranks Test				
Ranks				
	N	Mean Rank	Sum of Ranks	
X2 - X1	Negative Ranks	19 <sup>a</sup>	11,34	215,50
	Positive Ranks	5 <sup>b</sup>	16,90	84,50
	Ties	8 <sup>c</sup>		
	Total	32		

a.  $X2 < X1$   
b.  $X2 > X1$   
c.  $X2 = X1$

Test Statistics <sup>b</sup>	
	X2 - X1
Z	-1,872 <sup>a</sup>
Asymp. Sig. (2-tailed)	,061

a. Based on positive ranks.  
b. Wilcoxon Signed Ranks Test

**Figura 5.11:** Resultados do teste de Wilcoxon (*Signed-Rank*) para  $H_0a$

Para a aplicação do teste no nível  $b$  (fluxo de dados), os seguintes parâmetros foram definidos:

- $H_0b$  = Não existe diferença de *strength* do critério de teste fluxo de dados, considerando-se a variável “Porcentagem de Cobertura/Programa no paradigma oposto”, em razão do paradigma. ( $H_0b : \mu_{Db} = 0$ )
- $H_1b$  = Existe diferença de *strength* do critério de teste fluxo de dados, considerando-se a variável “Porcentagem de Cobertura/Programa no paradigma oposto”, em razão do paradigma. ( $H_1b : \mu_{Db} \neq 0$ )

- Nível de significância ( $\alpha$ ) = 0,05.

O resultado do teste sobre a variável  $\mu_D b$  é exibido na Figura 5.12. Conforme pode-se observar, o resultado gerado para o teste de Wilcoxon apresentou probabilidade de significância (p-value)  $p = 0,058$ . Como  $p > \alpha$ , não é possível rejeitar a hipótese nula  $H_0 b$  de que  $\mu_D b = 0$ , ao nível de 5%.

Descriptive Statistics					
	N	Mean	Std. Deviation	Minimum	Maximum
X1	32	89,4759%	12,97904%	33,33%	100,00%
X2	32	92,9094%	8,02266%	66,00%	100,00%

Wilcoxon Signed Ranks Test				
Ranks				
	N	Mean Rank	Sum of Ranks	
X2 - X1				
Negative Ranks	8 <sup>a</sup>	12,63	101,00	
Positive Ranks	18 <sup>b</sup>	13,89	250,00	
Ties	6 <sup>c</sup>			
Total	32			

a.  $X_2 < X_1$   
b.  $X_2 > X_1$   
c.  $X_2 = X_1$

Test Statistics <sup>b</sup>	
	X2 - X1
Z	-1,892 <sup>a</sup>
Asymp. Sig. (2-tailed)	,058

a. Based on negative ranks.  
b. Wilcoxon Signed Ranks Test

**Figura 5.12:** Resultados do teste de Wilcoxon (*Signed-Rank*) para  $H_0 b$

Com relação à segunda hipótese do plano, as duas etapas do teste de hipótese mostraram-se condizentes entre si, não fornecendo indícios para rejeição da hipótese nula nos dois casos. Isto, todavia, não permite aceitar a hipótese nula como verdadeira. O resultado apresentado pelo teste de hipótese, contudo, é distinto das observações iniciais sobre as medidas de *strength* dos dois conjuntos, a qual apontava “ligeiras” diferenças a favor de um dos paradigma em cada um dos dois níveis de critérios considerados. Nesse caso, a análise inicial dos dados deve ser desconsiderada em favor do teste de hipótese, já que este último consiste em um método estatísticos mais confiável e as pequenas diferenças observadas na análise pura dos dados podem ser consideradas como irrelevantes para se sustentar a suspeita da existência de uma diferença significativa de *strength*, com relação aos paradigmas.

## 5.4 Considerações Finais

Neste capítulo discutiu-se todos os detalhes da condução e análise do estudo proposto nesse trabalho. A condução consistiu na implementação das tarefas propostas no plano de estudo previamente definido. Desta forma, todas as informações relativas à forma como a instrumentação do estudo foi utilizada para gerar os dados relevantes à investigação do estudo foram descritas.

A análise dos dados foi realizada em duas fases. A primeira consistiu em apresentar todas as medidas para cada tipo de métrica considerada no estudo e juntamente com as medidas de tendência central relativas, obter informações que pudessem ser analisadas pontualmente, caracterizando o contexto dos programas e dos conjuntos de teste envolvidos.

Na segunda fase uma abordagem mais rigorosa foi empregada, para verificação das hipóteses definidas no plano. A aplicação dos testes para verificação das hipóteses definidas no plano foi precedida de um teste de ajustamento para verificação da normalidade da distribuição amostral das variáveis a serem utilizadas na verificação das hipóteses. Para este fim empregou-se o teste de Shapiro-Wilk, adequado ao tamanho da amostra considerada. O teste revelou uma distribuição não normal dos dados amostrais para as variáveis relacionadas às duas hipóteses investigadas, apontando a necessidade de aplicação de um teste não-paramétrico para a verificação das hipóteses.

O teste não paramétrico adotado foi o teste de Wilcoxon. A aplicação desse teste foi feita em duas etapas para cada uma das duas hipóteses consideradas. Essa medida foi tomada visando atender o *design* experimental que subdividia os critérios em nível fluxo de controle e fluxo de dados.

Na verificação da primeira hipótese, relacionada à diferença de custo dos paradigmas, foi revelada uma discrepância entre os resultados para os critérios no nível de fluxo de controle e no nível de fluxo de dados, não rejeitando para o primeiro nível e rejeitando para o segundo nível. Essa dualidade foi resolvida retomando-se a análise pontual dos dados, feita na primeira parte da seção de análise. Nesta, as observações feitas sobre a média, desvios padrões, valores mínimos e máximos indicavam na direção da não rejeição da hipótese nula em ambos os níveis. Além disso, o valor da probabilidade de significância que rejeitou o teste de hipótese no segundo nível, por estar muito próximo ao nível de significância adotado, também foi considerado para questionamento dessa rejeição.

A verificação da segunda hipótese, relacionada à diferença de *strength* entre os paradigmas, não apresentou discrepancia entre os níveis, não conseguindo rejeitar a hipótese nula tanto no nível fluxo de controle quanto no nível fluxo de dados.

Logo, a análise pontual e estatística dos dados permitiram concluir que não é possível afirmar que existe uma diferença de custo e de *strength*, em razão dos paradigmas, tanto para critérios estruturais de fluxo de controle quanto para critérios de fluxo de dados, considerando-se o domínio de programas investigados e o contexto no qual a investigação foi realizada. Isso significa que ainda que não se tenha conseguido demonstrar a existências dessas diferenças nesta investigação, novos

estudos experimentais futuros devem ser realizado, considerando-se um contexto mais amplo de programas, linguagens e participantes. Esses estudos poderiam fornecer mais evidências a favor da aceitação das hipótese nulas, ou ainda, revelar outros domínios em que os dados forneçam evidências suficientes para refutá-las. De qualquer forma, a realização desses novos experimentos poderá agora, ser amparada pelos artefatos gerados nesse trabalho e seus resultados e conclusões, comparados com os já obtidos.



## Conclusões e Trabalhos Futuros

---

---

Este trabalho realizou um estudo experimental comparando custo e *strength* de critérios de teste estruturais aplicados a programas do paradigma procedural e orientado a objetos, no domínio de estrutura de dados, provendo resultados iniciais na área sobre essa questão, uma vez que nenhum trabalho com tal finalidade havia sido identificado.

Conforme discutido ao longo do trabalho, a atividade e os resultados de teste estão sujeitos à influência de diversos fatores, dentre eles o paradigma de desenvolvimento adotado e o domínio de programas considerado. Para se avaliar essas questões, além do conhecimento de testes, a realização de estudos experimentais considerando-se os termos e processos da engenharia de software experimental se faz necessária. A execução da investigação seguindo esses princípios colabora com a obtenção de resultados mais objetivos e significativos, além de fornecer detalhes sobre a maneira com que o mesmo é conduzido. Isso possibilita uma avaliação mais clara das questões de validade envolvidas e fornece bases para a replicação/ampliação do estudo em outros cenários.

Para esse fim, nos capítulos iniciais dessa dissertação foi apresentada toda a fundamentação teórica de teste de software e de engenharia de software experimental, considerando também os trabalhos relacionados que envolvessem estudos experimentais para comparação entre técnicas e critérios de teste. O levantamento dos trabalhos relacionados foi importante tanto na assimilação prática dos conceitos de experimentação aplicados a teste, quanto na detecção das forças e fraquezas de cada trabalho, com relação às questões de validade.

Para o planejamento, condução e análise do estudo, foram empregados os termos e processos utilizados na experimentação controlada, à medida em que estes se mostraram apropriados. Para tanto, foram utilizados:

- O *template* GQM para estabelecimento do contexto da investigação;

- A escolha das variáveis independentes, dependentes e a definição das hipóteses de interesse para o estudo. Além das variáveis, foi considerada a coleta de 12 métricas de implementação, para caracterização de propriedades estáticas dos programas envolvidos no estudo;
- A elaboração de um *design* experimental que refletisse os propósitos comparativos analisados. O design em questão aplicou os princípios de *aninhamento*, *balanceamento* e *randomização*, sobre os fatores envolvidos no estudo;
- A análise de questões de validade interna, externa, de construção e de conclusão, relacionadas ao tipo de estudo considerado;
- A instrumentação adequada para construção e utilização dos artefatos necessários à condução do estudo. Dentre os artefatos desenvolvidos constam: A definição dos próprios *templates* dos documentos e formulários empregados no estudo; 32 documentos de especificação, relativos a cada programa testado no estudo; 32 programas C e 32 programas Java, correspondentes às implementações procedural e OO de cada especificação; 1 documento de implementação para disponibilização das métricas de implementação e auxilio na realização dos testes funcionais, correspondente a cada uma das implementações envolvidas; 1 formulário para descrição das classes de equivalência de cada especificação, durante a realização dos testes funcionais; 1 formulário de teste para registro textual dos casos de teste e coleta das medidas de teste correspondente a cada uma das implementações;
- A realização de uma análise estatística sobre os dados dividida em duas etapas. A primeira fazendo uma análise pontual sobre as informações obtidas e a segunda considerando a utilização de testes de hipóteses;
- A definição de uma máquina virtual, contendo todos os artefatos descritos anteriormente mais as ferramentas, as quais já se encontram instaladas e configuradas para uso.

Espera-se que o arcabouço de artefatos gerados, a experiência adquirida na execução de cada etapa do processo (plano, condução e análise) e os resultados obtidos ao final da investigação, sejam úteis na definição de futuros estudos experimentais na área de testes.

Os resultados obtidos por meio da análise realizada no estudo não forneceram evidências quanto à existência de diferenças de custos e de *strength* entre os paradigmas procedimentais e OO, considerando o domínio de estrutura de dados. Ainda que esses resultados estejam na direção correta com relação à comparação de custos e *strength* entre os paradigmas procedural e OO, a realização de estudos experimentais futuros considerando um contexto mais amplo de programas, linguagens e participantes se faz necessário. Esses estudos serviriam para aumentar a confiança quanto à não-rejeição das hipóteses nulas verificadas nesse trabalho, fornecendo mais evidências a favor da aceitação das mesmas ou para encontrar outros domínios em que as evidências sejam suficientes para refutá-las.

Além disso, acredita-se que tanto o material gerado quanto os resultados de teste propriamente, serão úteis no contexto de ensino e treinamento de teste de software, uma vez que foram definidos no domínio de programas de estrutura de dados e considerando esta finalidade.

## 6.1 Contribuições

Pode-se destacar como principais contribuições deste trabalho:

1. Os resultados sobre a comparação de custo e *strength* de critérios de teste, no domínio de programas de estrutura de dados.
2. A definição de um arcabouço de artefatos e resultados experimentais para replicação/ampliação de novos experimentos.
3. A elaboração de artefatos reaproveitáveis no contexto de ensino e treinamento de teste de software.

Dentre os artefatos gerados, destacam-se:

- A definição de uma máquina virtual com as ferramentas de teste e programas já instalados e configurados para uso.
- A elaboração de um documento de “guidelines”, contendo a descrição de operação de cada uma das ferramentas de teste envolvidas.
- A documentação das especificações dos programas envolvidos, as quais individualizaram os requisitos de cada programa tornando-os independentes entre si;
- O ajuste de 32 implementações C e 32 implementações Java, para correspondência das funcionalidades de operações equivalentes.
- A disponibilização de implementações e resultados de teste que sirvam de oráculo para comparação com novos resultados de teste sobre os mesmos programas. Os resultados disponíveis são relativos aos critérios Particionamento de Equivalência, Análise Valor Limite, Todos-Nós, Todas-Arestas, Todos-Usos e Todos Potenciais Usos.
- A disponibilização de medidas de implementação e de teste para cada um dos 64 programas envolvidos no estudo.

## 6.2 Dificuldades e Limitações

O presente trabalho apresentou algumas dificuldades e limitações com relação à sua condução.

A primeira delas relaciona-se às próprias ferramentas de teste. Nota-se que apesar das ferramentas escolhidas atenderem os requisitos mínimos para condução do estudo, praticamente não haviam opções alternativas disponíveis. No caso de testes funcionais por exemplo, a ferramenta CUTe, para testes de programas C, foi a única ferramenta cuja operação realmente se mostrou estável e semelhante com a ferramenta JUnit, para testes de programas Java. Mesmo assim, algumas assertivas que já existiam “prontas” na JUnit, tiveram que ser escritas em termos de expressões lógicas na ferramenta CUTe por não existirem na mesma. Além disso, pouca documentação para instalação/configuração dessa última ferramenta se encontrava disponível, o que exigiu um esforço maior na realização da tarefa.

A ferramenta Poke-Tool, para testes estruturais em C, além de dificuldades com o acerto da configuração exata exigida para sua correta instalação e uso, dependeu da definição de alguns *scripts* para sistematização da execução dos teste. A falta de uma interface gráfica que auxiliasse a realização das operações na ferramenta e a necessidade de adaptação do arquivo contendo a implementação dos casos de teste funcionais, também contribuiu para aumento dos esforços na condução dos testes. Para essa ferramenta em especial, não dispunha-se de nenhuma outra alternativa que avaliasse o mesmo conjunto de critérios. A alternativa mais próxima, além de ser comercial, não implementava um dos critérios de interesse para o estudo (Potenciais-Usos).

Uma outra questão que limitou o estudo, foi a questão da ferramenta Poke-Tool, diferentemente da JaBUTi, aplicar otimizações na geração dos elementos requeridos. Essa característica representou uma ameaça sobre a validade de construção dos resultados, eliminando a possibilidade de comparação dos elementos requeridos gerados e elementos requeridos não-executáveis, entre os programas das duas linguagens. De qualquer forma, as informações obtidas a partir desses dados foram legadas para estudos posteriores que possam vir a utilizá-los na comparação dentro do mesmo paradigma.

Com relação à preparação do estudo, ressalta-se a dificuldade de adequar as operações e suas respectivas funcionalidades, entre os dois paradigmas. Esse trabalho teve de ser precedido do ajustamento das próprias especificações, que originalmente apresentavam-se impróprias para realização de um estudo experimental. Isso porque além de interdependentes, as descrições das mesmas, em alguns casos, se dava em cima dos próprios códigos dos programas correspondentes, não fornecendo o isolamento adequado entre especificação e implementação para a aplicação dos testes funcionais.

## 6.3 Trabalhos futuros

Trabalhos futuros que verifiquem as hipóteses deste trabalho considerando outras técnicas e estratégias de teste são oportunos.

Um trabalho de interesse considerando a utilização dos artefatos já gerados, seria a avaliação da eficácia em revelar erros dos critérios estruturais, com relação aos paradigmas procedural e OO.

A própria replicação desse estudo, considerando outros domínios de programas, ou a redefinição do mesmo envolvendo mais linguagens de programação e participantes, colaboraria para enriquecimento da base de informações em teste, além de ampliar a caracterização dos resultados para vários contextos distintos.

No âmbito do grupo de pesquisa em que este trabalho foi desenvolvido, um outro trabalho de mestrado vem sendo realizado considerando o contexto de análise de mutantes para verificação das mesmas hipóteses. Ao final de sua realização, espera-se que os novos resultados gerados sejam integrados à base de informações já estabelecida. Isso colaboraria para a formação de um corpo de conhecimento sólido, no domínio de programas de Estrutura de Dados, envolvendo as três principais técnicas de teste de programas. Os materiais gerados nos dois trabalhos poderá, ainda, ser utilizado em uma proposta de parceria do grupo de pesquisa com o autor do *benchmark* de programas utilizados, para definição de novos materiais didáticos voltados ao ensino de teste de software.



# Referências Bibliográficas

---

---

- ACREE, A. T.; BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. *Mutation analysis.* Relatório Técnico GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, 1979.
- AGRAWAL, H. *Design of mutant operators for the C programming language.* Relatório Técnico SERC-TR-41-P, Software Engineering Research Center/Purdue University, 1989.
- AGRAWAL, H.; ALBERI, J. L.; HORGAN, J. R.; LI, J. J.; LONDON, S.; WONG, W. E.; GHOSH, S.; WILDE, N. Mining system tests to aid software maintenance. *IEEE Computer*, v. 31, n. 7, p. 64–73, 1998.
- AMARAL, E. A. G. G.; TRAVASSOS, G. H. A package model for software engineering experiments. In: *ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2003)*, Roma, Itália, 2003, p. seção de poster.
- BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Towards the determination of sufficient mutant operators for C. In: *First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, (Edição especial do Software Testing Verification and Reliability Journal, 11(2), 2001), 2000.
- BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability Journal*, v. 11, n. 2, p. 113–136, John Wiley & Sons, 2001.
- BASILI, V.; GREEN, S.; LAITENBERGER, O.; LANUBILE, F.; SHULL, F.; SØRUMGÅRD, S.; ZELKOWITZ, M. Lab package for the empirical investigation of perspective-based reading. [On-line], *World Wide Web*.  
Disponível em: [http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr\\_package/manual.html](http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html) (Acessado em 26/08/2008)

- BASILI, V. R.; GREEN, S.; LAITENBERGER, O.; SHULL, F.; SORUMGARD, S.; ZELKOWITZ, M. The empirical investigation of perspective-based reading. *Empirical Software Engineering: An International Journal*, v. 1, n. 2, p. 133,164, 1996.
- BASILI, V. R.; SELBY, R. W. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, v. 13, n. 12, p. 1278–1296, 1987.
- BEIZER, B. *Software testing techniques*. 2 ed. New York: Van Nostrand Reinhold, 1990.
- BERTOLINO, A. The (im)maturity level of software testing. *WERST Proceedings/ACM SIGSOFT*, v. 29, n. 05, p. 1–4, 2004.
- BIBLE, J.; ROTHERMEL, G.; ROSENBLUM, D. *A comparative study of coarse and fine-grained safe regression test selection*. Relatório Técnico 99-60-05, Computer Science Department, Oregon State university, 1999.
- BIEMAN, J. M.; SCHULTZ, J. L. An empirical evaluation (and specification) of the all-du-paths testing criterion. *Software Engineering Journal*, p. 43–51, 1992.
- BINDER, R. V. Testing object-oriented systems: A status report. *American Programmer*, v. 07, n. 04, p. 222–228, 1994.
- BINDER, R. V. *Testing object-oriented systems: Models, patterns, and tools*, v. 1. Addison Wesley Longman, Inc., 1999.
- BRIAND, L. C. A critical analysis of empirical research in software testing. *First International Symposium on ESEM*, p. 1–8, 2007.
- BRIAND, L. C.; DIFFERDING, M. C.; ROMBACH, H. D. Practical guidelines for measurement-based process improvement. *Software Process Improvement and Practice*, v. 2, n. 4, p. 253,280, 1996.
- BUDD, A. T. *Mutation analysis: Ideas, examples, problems and prospects*. Computer Program Testing North-Holland Publishing Company, p. 129–148, 1981.
- CHAIM, M. L. *Poke-Tool — uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados*. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, 1991.
- COLANZI, T. E. *Uma abordagem integrada de desenvolvimento e teste de software baseada na uml*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 1999.
- CONOVER, W. J. *Practical nonparametric statistics*. John Wiley & Sons, 1998.
- DELAMARO, M. E. *Proteum – um ambiente de teste baseado na análise de mutantes*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 1993.

- DELAMARO, M. E. *Mutação de interface: Um critério de adequação interprocedimental para o teste de integração.* Tese de Doutoramento, IFSC/USP, São Carlos, SP, 1997.
- DELAMARO, M. E.; MALDONADO, J. C. *Uma visão sobre a aplicação da análise de mutantes.* Relatório Técnico 133, ICMC, São Carlos, SP, notas do ICMC, Série Computação, 1993.
- DELAMARO, M. E.; MALDONADO, J. C. Proteum: A tool for the assessment of test adequacy for C programs. In: *Conference on Performability in Computing Systems (PCS'96)*, Brunswick, NJ, 1996, p. 79–95.
- DEMILLO, R. A. *Software testing and evaluation.* The Benjamin/Commings Publishing Company, Inc, 1978.
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 4, p. 34–41, 1978.
- DOLINER, M. Cobertura. [On-line], *World Wide Web*.  
Disponível em: <http://cobertura.sourceforge.net/index.html> (Acessado em 23/01/2007)
- DOMINGUES, A. L. S. *Avaliação de critérios e ferramentas de teste para programas oo.* Dissertação de Mestrado, ICMC/USP, São Carlos, SP, (in Portuguese), 2002.
- DOMINGUES, A. L. S.; SIMÃO, A. S.; VINCENZI, A. M. R.; MALDONADO, J. C. Evaltool: Um ambiente de apoio à avaliação e seleção de ferramentas de teste para programas orientados a objetos. In: *Anais da Sessão de Ferramentas do XVI Simpósio Brasileiro de Engenharia de Software*, Gramado, RS, 2002, p. 384–389.
- DÓRIA, E. S. *Replicação de estudos empíricos em engenharia de software.* Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2001.
- FABBRI, S. C. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. *Introdução ao teste de software,* cáp. Teste Funcional. 1st ed Campus / Elsevier, p. 9,25, 2007.
- FELIZARDO, K. R. *COTEST - uma ferramenta de apoio à replicação de um experimento baseado em código fonte.* Dissertação de Mestrado, UFSCar, São Carlos, SP, 2003.
- FRANKL, P.; IAKOUNENKO, O. Further empirical studies of test effectiveness. In: *ACM SIGSOFT International Symposium on Foundations on Software Engineering*, Lake Buena Vista, Florida, USA, 1998, p. 153–162.
- FRANKL, P. G.; WEISS, S. N. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transaction on Software Engineering*, v. 19, n. 8, p. 774–787, 1993.

- FRANKL, P. G.; WEISS, S. N.; HU, C. All-uses vs. mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, v. 38, p. 235–253, 1997.
- FUJIWARA, S.; BOCHMANN, G. V.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, v. 17, n. 6, p. 591–603, 1991.
- GÖNENÇ, G. A method for design of fault-detection experiments. *IEEE Transactions on Computers*, v. 19, n. 6, p. 551–558, 1970.
- GRAVES, T. L.; HARROLD, M. J.; KIM, J. M.; A., P.; ROTHERMEL, G. An empirical study of regression test selection techniques. In: *1998 International Conference on Software Engineering*, Kyoto, Japan, 1998, p. 188–197.
- HARROLD, M. J.; ROTHERMEL, G. Perfoming data flow testing on classes. In: *II ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, p. 154–163.
- HÖHN, E. N. Um framework de estudos experimentais de técnicas de verificação, validação e teste no contexto de orientação a objeto e a aspecto. Exame de Qualificação de Doutorado – Instituto de Ciências Matemáticas e de Computação – USP – São Carlos, SP, 2007.
- HOFFMAN, D.; STROOPER, P. C. A framework for automated class testing. *Software Practice and Experience*, v. 27, n. 5, p. 573–597, 1997.
- HOLLANDER, M.; WOLFE, D. A. *Nonparametric statistical methods, 2nd edition*. Wiley-Interscience, 1999.
- HORGAN, J. R.; LONDON, S. A. Data flow coverage and the C language. In: *Symposium Software Testing, Analysis, and Verification*, 1991, p. 87–97.
- HORGAN, J. R.; MATHUR, A. P. Assessing testing tools in research and education. *IEEE Transactions on Software Engineering*, v. 9, n. 3, p. 61–69, 1992.
- HOWDEN, W. E. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, v. 4, n. 4, p. 293–298, 1978.
- HOWDEN, W. E. *Software engineering and technology: Functional program testing and analysis*. New York: McGraw-Hill Book Co, 1987.
- HUTCHINS, M.; FOSTER, H.; GORADIA, T.; OSTRAND, T. Experiments on the effectiveness of dataflow and controlflow-based test adequacy criteria. In: *16th International Conference on Software Engineering*, Sorrento, Italy, 1994, p. 191–200.
- IEEE *IEEE standard glossary of Software Engineering terminology*. Padrão 620.12, IEEE, 1990.

- INC., S. Spss statistics. Disponível em: <http://www.spss.com/>. Último acesso: 16/03/2009, 2009.
- INTERNATIONAL SOFTWARE AUTOMATION Panorama c/c++. [On-line], World Wide Web. Disponível em: [ftp://ftp.parasoft.com/jtest/jtest\\_win32.exe](ftp://ftp.parasoft.com/jtest/jtest_win32.exe) (Acessado em 24/02/2002)
- IPL Ipl software products group. [On-line], World Wide Web. Disponível em: <http://www.ipl.com> (Acessado em 25/02/2008)
- JEDLITSCHKA, A.; PFAHL, D. *Reporting guidelines for controlled experiments in software engineering*. Relatório Técnico ISERN-05-01, ISERN-REPORT, 2005.
- JURISTO, N.; MORENO, A. M. *Basics of software engineering experimentation*. P.O. Box 17, 3300 AA, Dordrecht, The Netherlands: Kluwer Academic Publishers, 2001.
- JURISTO, N.; MORENO, A. M.; VEGAS, S. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, v. 9, n. 7-44, p. 133,164, 2004.
- KAMSTIES, E.; LOTT, C. M. An empirical evaluation of three defect-detection techniques. In: *V European Software Engineering Conference*, Londres, UK, 1995, p. 362–383.
- KIM, J. M.; PORTER, A.; ROTHERMEL, G. An empirical study of regression test application frequency. In: *22nd International Conference on Software Engineering*, Limerick, Ireland, 2000, p. 126–135.
- KUNG, D.; LU, Y.; VENUGOPALAN, N.; HSIA, P.; TOYOSHIMA, Y.; CHEN, C.; GAO, J. Object state testing and fault analysis for reliable software systems. In: *7th International Symposium on Software Reliability Engineering*, White Plains, NY: IEEE Computer Society Press, 1996, p. 76–85.
- LEMOS, O. A. L. Uma contribuição para o teste estrutural de programas orientados a aspectos. Exame de Qualificação de Doutorado – Instituto de Ciências Matemáticas e de Computação – USP – São Carlos, SP, 2006.
- LINKMAN, S.; VINCENZI, A. M. R.; MALDONADO, J. An evaluation of systematic functional testing using mutation testing. In: *7th International Conference on Empirical Assessment in Software Engineering - EASE*, Keele, UK, 2003, p. 1–15.
- MA, Y.-S.; OFFUTT, J.; KWON, Y.-R. Mujava: a mutation system for java. In: *ICSE '06: Proceeding of the 28th international conference on Software engineering*, New York, NY, USA: ACM, 2006, p. 827–830.

- MAFRÀ, S. N.; TRAVASSOS, G. H. *Estudos primários e secundários apoiando a busca por evidência em engenharia de software.* Relatório Técnico RT-ES 687/06, PESC COPPE/UFRJ, 2005.
- MALDONADO, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software.* Tese de doutoramento, DCA/FEE/UNICAMP, Campinas, SP, 1991.
- MALDONADO, J. C.; AURI, E. F. B.; VINCENZI, M. R.; DELAMARO, M. E.; SOUZA, S.; JINO, M. *Introdução ao teste de software.* Instituto de Ciências Matemáticas e de Computação - ICMC-USP, nota Didática n. 65, 2004.
- MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E. Evaluation N-selective mutation for C programs: Unit and integration testing. In: *Mutation 2000 Symposium*, San Jose, CA: Kluwer Academic Publishers, 2000a, p. 22–33.
- MALDONADO, J. C.; DELAMARO, M. E.; JINO, M. *Introdução ao teste de software.* Editora Elsevier, 1st edition, 2007.
- MALDONADO, J. C.; VERGÍLIO, S. R.; CHAIM, M. L.; JINO, M. Critério potenciais usos: Análise da aplicação de um benchmark. In: *VI Simpósio Brasileiro de Engenharia de Software*, Gramado, RS, 1992, p. 357–374.
- MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E. Proteum IM 2.0: An integrated mutation testing environment. In: *Mutation 2000 Symposium*, San Jose, CA: Kluwer Academic Publishers, 2000b, p. 91–101.
- MATHUR, A. P. On the relative strengths of data flow and mutation testing. In: *Ninth Annual Pacific Northwest Software Quality Conference*, Portland, OR, 1991, p. 165–181.
- MATHUR, A. P.; WONG, W. E. Evaluation of the cost of alternative mutation strategies. In: *VII Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, RJ, Brazil, 1993, p. 320–335.
- MATHUR, A. P.; WONG, W. E. An empirical comparison of data flow and mutation based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, v. 4, n. 1, p. 9–31, 1994.
- MCCABE, T. J., W. A. H. *Structured testing: A testing methodology using the cyclomatic complexity metric.* NIST Special Publication 500-235, Computer Systems Laboratory National Institute of Standards and Technology Gaithersburg, MD. USA, 1996.
- MYERS, G. J. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, v. 21, n. 9, p. 760–768, 1978.
- MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. *The art of software testing.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

- NETO, B. B.; SCARMINIO, I. S.; BRUNS, R. E. *Como fazer experimentos: Pesquisa e desenvolvimento na ciência e na indústria.* Editora da UNICAMP, 2001.
- OFFUT, A. J.; LEE, S. D. Empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, v. 20, n. 5, p. 337–344, 1994.
- OFFUTT, A. J. The coupling effect: Fact or fiction. In: *Proceedings of Symposium on Testing, Analysis, and Verification*, ACM Press, 1989, p. 131–140.
- OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, v. 5, n. 2, p. 99–118, 1996a.
- OFFUTT, A. J.; PAN, J.; TEWARY, K.; ZHANG, T. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, v. 26, n. 2, p. 165–176, 1996b.
- OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An experimental evaluation of selective mutation. In: *15th International Conference on Software Engineering*, Baltimore, MD: IEEE Computer Society Press, 1993, p. 100–107.
- OSTERWEIL, L. Strategic directions in software quality. *ACM Comput. Surv.*, v. 28, n. 4, p. 738–750, 1996.
- PARASOFT CORPORATION C++ Test. [On-line], World Wide Web.  
Disponível em: <ftp://ftp.parasoft.com/cpptest/C++Test-10.exe> (Acessado em 24/02/2002)
- PARASOFT CORPORATION Jtest. [On-line], World Wide Web.  
Disponível em: [ftp://ftp.parasoft.com/jtest/jtest\\_win32.exe](ftp://ftp.parasoft.com/jtest/jtest_win32.exe) (Acessado em 24/02/2002)
- PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. *J. Object Oriented Program.*, v. 2, n. 5, p. 13–19, 1990.
- PRESSMAN, R. S. *Engenharia de software.* 5nd. ed. McGraw-Hill, 2002.
- RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for program test data selection. In: *6th International Conference on Software Engineering*, Tokio, Japan, 1982, p. 272–278.
- RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985.
- RATIONAL SOFTWARE CORPORATION PureCoverage. [On-line], World Wide Web.  
Disponível em: <ftp://ftp.rational.com/exchange/outgoing/devtools/nt/coverage65.exe> (Acessado em 03/03/2000)

- RIEHLE, D. Junit 3.8 documented using collaborations. *SIGSOFT Softw. Eng. Notes*, v. 33, n. 2, p. 1–28, 2008.
- ROSENBLUM, D.; WEYUKER., E. Lessons learned from a regression testing case study. *Empirical Software Engineering Journal*, v. 2, n. 2, 1997.
- ROTHERMEL, G.; HARROLD, M. J. A safe, efficient regression test selection technique. *ACM Transaction on Software Engineering and Methodology*, v. 6, n. 2, p. 173–210, 1997.
- ROTHERMEL, G.; HARROLD, M. J. Empirical studies of a safe regression test selection technique. *IEEE Transaction on Software Engineering*, v. 24, n. 6, p. 401–419, 1998.
- ROTHERMEL, H. D. S. E. G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, v. 10, n. 4, 2005.
- SABNANI, K. K.; DAHBURA, A. Protocol test generation procedure. *Computer Networks and ISDN Systems*, v. 15, n. 4, p. 285–297, 1988.
- SHULL, F.; MENDONÇA, M.; BASILI, V. R.; CARVER, J.; MALDONADO, J. C.; FABBRI, S.; TRAVASSOS, G. H.; FERREIRA, M. C. Knowledge-sharing issues in experimental software engineering. *Empirical Software Engineering: An International Journal*, v. 9, n. 1, p. 111,137, 2004.
- SIMÃO, A. S.; MALDONADO, J. C. Mutation based test sequence generation for Petri nets. In: *Proceedings of III Workshop of Formal Methods*, João Pessoa, PB, 2000, p. 68–79.
- SIMÃO, A. S.; SOUZA, S. R. L.; MALDONADO, J. C. A family of coverage testing criteria for coloured petri nets. In: *XVII Simpósio Brasileiro de Engenharia de Software*, Manaus, AM, 2003, p. 209–224.
- SOMMERLAD, P.; GRAF, E. Cute: C++ unit testing easier. In: *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, New York, NY, USA: ACM, 2007, p. 783–784.
- SOUZA, S. R. S. *Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de software*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC/USP), São Carlos, SP, 1996.
- SOUZA, S. R. S.; FABBRI, S. C. P. F.; BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. *Introdução ao teste de software*, cap. Estudos Teóricos e Experimentais. 1st ed Campus / Elsevier, p. 251,268, 2007.

- SOUZA, S. R. S.; MALDONADO, J. C.; FABBRI, S. C. P. F.; MASIERO, P. C. Statecharts specifications: A family of coverage testing criteria. In: *CLEI2000 - Conferência Latino Americana de Informática*, Cidade do Mexico, Mexico, 2000.
- SOUZA, S. R. S.; MALDONADO, J. C.; VERGILIO, S. R. Análise de mutantes e potenciais-usos: Uma avaliação empírica. In: *XIII Conferência Internacional de Tecnologia de Software - CITS'97*, Curitiba, PR, Brasil, 1997, p. 225–236.
- TAKASHI, T.; LIESENBERG, H. K. E.; XAVIER, D. T. *Programação orientada a objetos – uma visão integrada do paradigma de objetos*. 1st. ed. São Paulo - SP: VII Escola de Computação, 1990.
- TRAVASSOS, G. *Introdução à engenharia de software experimental*. Relatório Técnico RT-ES 590/02, PESC COPPE/UFRJ, 2002.
- VERGÍLIO, S. R.; MALDONADO, J. C.; JINO, M. Caminhos não-executáveis na automação das atividades de teste. In: *Anais do VI Simpósio Brasileiro de Engenharia de Software*, Gramado, RS, 1992, p. 343–356.
- VILELA, P. R.; MALDONADO, J. C.; JINO, M. Program graph visualization. *Software-Practice & Experience*, v. 27, n. 11, p. 1245–1262, 1997.
- VINCENZI, A. M. R. *Subsídios para o estabelecimento de estratégias de teste baseadas na técnica de mutação*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 1998.
- VINCENZI, A. M. R. *Orientação a objeto: Definição e análise de recursos de teste e validação*. Tese de Doutoramento, ICMC/USP, São Carlos, SP, 2004.
- VINCENZI, A. M. R.; DOMINGUES, A. L. S.; DELAMARO, M. E.; MALDONADO, J. C. *Introdução ao teste de software*, cáp. Teste Orientado a Objetos e de Componentes. 1st ed Campus / Elsevier, p. 119,174, 2007.
- VINCENZI, A. M. R.; MALDONADO, J. C.; BARBOSA, E. F.; DELAMARO, M. E. Operadores essenciais de interface: Um estudo de caso. In: *13th Simpósio Brasileiro de Engenharia de Software*, Florianópolis, SC, 1999, p. 373–391.
- VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. JaBUTi: A coverage analysis tool for Java programs. In: *XVII SBES – Simpósio Brasileiro de Engenharia de Software*, Manaus, AM, Brasil, 2003, p. 79–84.
- VOKOLOS, F.; FRANKL, P. G. Empirical evaluation of the textual differencing regression testing technique. In: *International Conferenceon Software Maintenance (ICSM-98)*, Bethesda, Maryland: IEEE Computer Society Press, 1998.

- WEYUKER, E. J. On testing non-testable programs. *Computer Journal*, v. 25, n. 4, p. 465–470, 1982.
- WEYUKER, E. J. Comparing test data adequacy criteria. *Software Engineering Notes*, v. 14, n. 6, p. 42–49, 1989.
- WEYUKER, E. J. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, v. 16, n. 2, p. 121–128, 1990.
- WEYUKER, E. J.; WEISS, S. N.; HAMLET, R. G. Comparison of program testing strategies. In: *4th Symposium on Software Testing, Analysis and Verification*, Victoria, British Columbia, Canada, 1991, p. 1–10.
- WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.
- WONG, E.; MALDONADO, J. C.; DELAMARO, M. E.; MATHUR, A. P. Constrained mutation for C programs. In: *8º Simpósio Brasileiro de Engenharia de Software*, Curitiba, Brasil, 1994, p. 439–452.
- WONG, W. E. *On mutation and data flow*. Tese de Doutoramento, Department of Computer Science, Purdue University, W. Lafayette, IN, 1993.
- WONG, W. E.; MALDONADO, J. C.; SOUZA, M. E.; SOUZA, S. R. S. A comparison of selective mutation in C and Fortran. In: *Anais do Workshop do Projeto Validação e Teste de Sistemas de Operação*, Águas de Lindóia, SP, 1997, p. 71–84.
- WONG, W. E.; MATHUR, A. P. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, v. 4, n. 1, p. 69–83, 1995.
- WONG, W. E.; MATHUR, A. P.; MALDONADO, J. C. *Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness*, cap. 40 London: Chapman & Hall, p. 258–265, 1995.
- WOOD, M.; ROPER, M.; BROOKS, A.; MILLER, J. Comparing and combining software defect detection techniques: a replicated empirical study. In: *VI European Conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, Nova York, NY, EUA, 1997, p. 262–277.
- ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, v. 29, n. 4, p. 366–427, 1997.
- ZIVIANI, N. *Projeto de algoritmos com implementações em java e c++*. Thomson, 2005a.
- ZIVIANI, N. *Projeto de algoritmos com implementações em pascal e c*. 2nd. ed. Thomson, 2005b.

APÊNDICE

A

## Apendice A

---

---

A seguir são apresentadas as *guidelines* do estudo conduzido. O acesso aos demais artefatos gerados no trabalho está disponível no website: <http://www.box.net/shared/a2bo8h8x8z> (Último acesso: 16/03/2009).

## ***Guidelines do estudo experimental***

### **Ambiente de instrumentação**

Durante a condução de um experimento, o ideal é que se disponha uma mesma configuração que possa ser sistematizada entre os participantes e restabelecida durante a replicação/redefinição de novos experimento. Levando em conta que todas as ferramentas usadas no estudo rodam no sistema operacional Linux, resolvemos definir uma máquina virtual com todos os artefatos do experimento já instalados e configurados, de forma que ela possa ser gravada em mídia de DVD e rodada em qualquer computador. Para isso, basta ter o player da máquina virtual instalado e 6 GB de espaço livre em disco rígido, já que dados serão gravados durante o uso das ferramentas e para isso é necessário que a máquina esteja armazenada fora do DVD.

O player da máquina virtual é distribuído gratuitamente para instalação e uso, tanto em Windows quanto em Linux, e pode ser obtido no site do fabricante:

<http://www.vmware.com> (site do fabricante)

<http://www.vmware.com/download/player/> (link direto para página de download do player.  
Último acesso: 01/09/2008)

### **Definição das Randomizações.**

Conforme definido no design do experimento, a ordem de aplicação dos tratamentos (testes) deve ser estabelecida aleatoriamente. Para facilitar essa tarefa, pode-se usar o “GeRandom”, um pequeno aplicativo desenvolvido para essa finalidade. Neste, o usuário informa o título da randomização (ex.: “Seqüência Aleatória das Especificações por Critério”), número de seqüências aleatórias desejadas e a quantidade de elementos para cada seqüência. Por padrão, as seqüências aleatórias são geradas em termos de números inteiros não repetidos, variando de “1” até o tamanho da seqüência informada.

Para o estudo proposto, o número de elementos das seqüências é igual ao total de especificações, ou seja, 32.

Depois de estabelecidas as seqüências aleatórias das especificações, são definidas as ordens dos paradigmas, para cada especificação. Assim, 32 seqüências de dois elementos são definidas, cada uma com dois elementos representando os dois paradigmas (1 para procedural e 2 para OO).

Geradas as seqüências, o testador deve imprimi-las e guardá-las. Os testes devem seguir estritamente a ordem de execução definida na randomização, nos três níveis de critérios.

### Ferramentas de teste funcional

As ferramentas de teste funcional, apesar de não apoiarem a aplicação dos critérios (por não gerarem automaticamente os requisitos de teste) auxiliam o testador em sua execução. Como os programas foram implementados nas linguagens C e Java, duas ferramentas foram escolhidas para aplicação desses critérios (CUTe e JUnit). Essas ferramentas rodam sobre o mesmo ambiente de desenvolvimento (eclipse), dispõem de assertivas em comum para comparação dos resultados de teste obtidos e são bastante similares quanto à forma de operar.

#### 1) JUnit

Para os testes funcionais OO, foi eleita a ferramenta JUnit versão 3.8 . Preferiu-se usar essa versão, apesar de já existir uma versão mais recente, por motivos de reaproveitamento dos casos de teste em Java, durante os testes estruturais, já que a ferramenta de teste estrutural é compatível com essa versão.

Por motivos de organização, o testador deve criar para cada especificação do experimento, um novo projeto no JUnit. Abaixo é descrita a seqüência de passos para criar um novo projeto:

1. Abra o ambiente eclipse (atälho no desktop);
2. Clique em Arquivo/New/Java Project/;
3. Forneça um nome ao Projeto;
4. Adicione o programa Java a ser testado (ou sua pasta correspondente ao pacote no qual esteja declarado) na pasta “src” do projeto criado. Isso pode ser feito arrastando-se o arquivo Java (ou o diretório do pacote) do sistema operacional para a pasta dentro do eclipse;
5. Selecione a pasta “src” (ou pacote do programa dentro da pasta “src”), vá em File/New/JUnit Test Case;
6. Selecione a opção New JUnit 3 test;
7. Nomeie a classe de teste;
8. Selecione Finish/OK.

Para criar um caso de teste no arquivo de testes, defina um método “public” com retorno “void” e cujo nome se inicie com a palavra “test” + sufixo de sua preferência. Um exemplo é fornecido a seguir:

```
public void testFC1() {  
    Ordenacao ord = new Ordenacao();  
}
```

Uma boa prática de teste que deve ser adotada para este estudo é a de se definir somente um caso de teste por método.

Para rodar o caso de teste, selecione o arquivo contendo o conjunto de teste com o botão direito do mouse e clique em “Run As/ JUnit Test”. Os resultados de teste deverão ser exibidos na guia própria do JUnit, localizada na parte inferior do eclipse.

Finalizado os testes funcionais de um programa, abra o workspace do eclipse (localizado em: /home/experimento/workspace) compacte a pasta correspondente ao projeto de teste criado e copie-o para o subdiretório “JUnit” localizado no diretório:

```
"/home/experimento/exp/<nºprograma>/Funcional"
```

## 2) CUTe

Para os testes funcionais procedural, foi eleita a ferramenta CUTe versão 1.6.1.2 .

Por motivos de organização, o testador deve criar para cada especificação do experimento, um novo projeto no CUTe. Abaixo é descrita a seqüência de passos para criar um novo projeto:

9. Abra o ambiente eclipse (atälho no desktop);
10. Clique em Arquivo/New/Project/;
11. Selecione a opção “C++ Project” e clique em “next”;
12. Forneça um nome ao Projeto, selecione a opção “Cute Project” na janela “Project type” e clique em “finish”;
13. Sobre a pasta do projeto clique com o botão direito do mouse, vá até “properties/C/C++ General / Paths and Symbols. Na aba “Includes” selecione em “Languages” a opção “GNU C++”. Clique em “Add” e adicione o diretório:

```
"/home/experimento/eclipse/Boost/boost_1_36_0"
```

Confirme a operação clicando “Ok”.

14. Adicione o programa C a ser testado na pasta “src” do projeto criado. Isso pode ser feito arrastando-se o arquivo C do sistema operacional para a pasta dentro do eclipse;
15. A pasta “src” do projeto CUTe já contém por padrão um arquivo denominado “Test.cpp”. Esse arquivo deve ser usado para adicionar os casos de teste. A renomeação desse arquivo é opcional.
16. Antes de escrever os casos de teste inclua o arquivo a ser testado no arquivo de teste, utilizando para tanto o comando: #include "nomedoprograma"

Para criar um caso de teste no arquivo de testes, defina um procedimento com retorno “void” e cujo nome se inicie com a palavra “test” + sufixo de sua preferência. Um exemplo é fornecido a seguir:

```
void test1_1() {
    Vetor* vet;
    vet = (Vetor*)malloc(sizeof(Vetor[2]));
    ((*vet)[0]) = 9; ((*vet)[1]) = 17;
    MaxMin1 (*vet, &maior, &menor);
    ASSERT(maior == 17 && menor == 9);
}
```

Uma boa prática de teste que deve ser adotada para este estudo é a de se definir somente um caso de teste por método. Além disso, recomenda-se programar o arquivo de teste usando-se o padrão ANSI C.

Após descrição de todos os casos de teste, uma chamada para os mesmos deve ser definida no procedimento “main” do arquivo de teste. Para tanto, após o comando “cute::suite s”, insira o seguinte comando para cada procedimento de teste definido:

```
s.push_back(CUTE(nomeDoProcedimentoDeTeste));
```

Finalizada a contrução do arquivo de teste, compile o projeto de teste, pressionando as teclas “Ctrl+B”.

Para rodar o caso de teste, selecione o arquivo binário criado no diretório “Bináries” no projeto de teste e com o botão direito do mouse clique em “Run As/ CUTE Test”. Os resultados de teste deverão ser exibidos na guia própria do CUTE, localizada na parte inferior do eclipse.

Finalizado os testes funcionais de um programa, abra o workspace do eclipse (localizado em: /home/experimento/workspace) compacte a pasta correspondente ao projeto de teste criado e copie-o para o subdiretório “Cute” localizado no diretório:

```
"/home/experimento/exp/<nºprograma>/Funcional"
```

#### Ferramentas de teste estrutural

##### 3) JaBUTi

Para os testes estruturais OO, foi eleita a ferramenta JaBUTi versão 1.0.

Por motivos de organização, o testador deve criar para cada especificação do experimento, um novo projeto na JaBUTi. Abaixo é descrita a seqüência de passos para criar um novo projeto:

1. Descompacte a pasta do projeto JUnit do programa sob teste no diretório:

```
"/home/experimento/Jabuti"
```

Obs.: Caso já exista uma pasta bin e src nesse diretório, apague-a antes.

2. Abra a ferramenta JaBUTi, dando dois cliques sobre o ícone disponível no Desktop da máquina virtual.
3. Após a ferramenta ter sido carregada, clique em File/Open Classe.
4. Procure pelo arquivo de teste do projeto, dentro da pasta Bin e selecione-o com um clique.
5. Na janela "classpath" insira os diretórios:

"/home/experimento/Jabuti/bin:/home/experimento/Jabuti/src"

Clique em "Open".

6. Na próxima janela, selecione o programa a ser instrumentado (o programa sob teste propriamente) na árvore "User Package" e insira-o na janela "Classes to Instrument".
7. Em "Project Name", clique em "Select" e dê um nome para o projeto. Clique em "Open" para confirmar o nome do projeto.
8. Confirme a criação do projeto clicando em "Ok".

Para inserir o arquivo de casos de teste JUnit na JaBUTi, siga os seguintes passos:

1. Clique em Test Case/Executing JUnit Test Set. Na janela Warning selecione "yes".
2. A janela de importação dos casos de teste deverá ser aberta. Preencha com os seguintes parâmetros:

\*Path to JUnit test suíte source code: /home/experimento/Jabuti/src

\*Path to JUnit test suíte binary code: /home/experimento/Jabuti/bin

\*Test suite full qualified name:<nomeDoPacote.nomeDoArquivoDeTeste>

Obs.: "nomeDoArquivoDeTeste" sem extensão!

\*JaBUTi library: /home/experimento/Jabuti/Jabuti-bin-2007-12-19.zip

3. Clique em "Run Normally (No Trace)".
4. Caso o arquivo de teste não contenha erros, os métodos de teste deverão aparecer na janela "Test Case Execution Status".
5. Confirme a importação dos casos de teste clicando em "Run Collecting Trace Information".
6. Após alguns segundos, o display de alerta deverá ficar vermelho no topo direito da interface gráfica da JaBUTi. Quando isso ocorrer, clique em Update/Update. A avaliação de cobertura será realizada e os resultados serão atualizados.
7. Para acessar os resultados de teste, clique em "Summary/By Method". Os resultados de teste serão exibidos individualmente para cada método do programa testado.
8. Para alternar entre os critérios considerados, selecione uma das opções na barra superior da interface da JaBUTi: "All-Nodes-ei" (Todos-Nós), "All-Edges-ei" (Todas-Arestas), "All-Uses-ei" (Todos-Usos), "All-Pot-Uses-ei" (Todos-Potenciais-Usos).

Finalizado os testes na ferramenta JaBUTi, compacte as pastas “Bin” e “src” localizadas no diretório da JaBUTi, juntamente com os arquivos “.trc”, “.jbt” e “jar” e copie o arquivo compactado para o diret

#### 4) Poke-Tool

Para os testes estruturais procedural, foi eleita a ferramenta Poke-Tool.

Por motivos de organização, o testador deve criar para cada especificação do experimento, uma seção de teste na Poke-Tool. Abaixo é descrita a seqüência de passos para criar uma nova seção:

1. Crie uma pasta para a seção desejada dentro de:  
"/home/experimento/exp/<nºprograma>/Estrutural/PokeTool"

2. Copie o arquivo de teste e o programa a ser testado, criados durante os testes funcionais com a ferramenta CUTe.

3. Copie os 4 arquivos de *script* definidos para o estudo (“1\_Instrumenta.sh”, “2\_Compila.sh”, “3\_ExecutaCT.sh” e “4\_AvaliaCobertura.sh”), localizados no diretório:

"/home/experimento/poke"

para a pasta da seção de teste criada. Nas propriedades de cada arquivo de *script*, habilite a permissão para “Permitir execução do arquivo como programa”.

4. Edite o arquivo de teste para que possa se adequar aos requisitos da Poke-Tool. Para tanto faça as seguintes modificações:

\* Altere a extensão do arquivo de teste de “.cpp” para “.c”.

\* Remova os includes inerentes à ferramenta “CUTe”. A saber: #include "cute.h"  
#include "ide\_listener.h" e #include "cute\_runner.h".

\*Inclua as bibliotecas que possam ser necessárias às chamadas existentes dentro dos procedimentos de teste. Nos testes funcionais, pode não ter sido necessário incluir essas bibliotecas já que o eclipse as provia automaticamente (ex.: “stdlib.h”).

\* Comente as assertivas de cada procedimento de teste.

\*Remova todo o conteúdo do procedimento “void runSuite()”, deixando apenas as chamadas de procedimento, que devem ser feitas normalmente, ao invés de usando as diretivas do CUTe. Exemplo:

<b>Substituir <i>s.push_back(CUTE(test1_1); por test1_1();</i></b>
--

Realizada a adequação do arquivo de teste, retorne ao diretório da seção de teste criada. O próximo passo consiste em adequar os *scripts* para uso com o programa em questão. Para tanto modifique da seguinte forma:

\*No *script 1*, substitua “ConjuntoTeste.c” pelo nome do arquivo de teste criado.

\* O *script 2* não necessita de modificação.

\*No *script 3*, substitua o parâmetro “–fProcedimento” pelo nome dos procedimentos do programa sob teste para os quais deseja-se realizar a avaliação dos critérios de teste. O nome dos procedimentos devem ter “-f” como prefixo e devem ser separadas umas das outras por espaço.

\*No *script 4*, substitua a cada 4 linhas, o parâmetro “-dProcedimento” pelo nome de um novo procedimento do programa sob teste para o qual deseja-se realizar a avaliação dos critérios de teste. Ex.:

(...)
<i>newpokeaval -dProcedimento1 -nos -ne 1 to 1 newpokeaval -dProcedimento1 -arcs -ne 1 to 1 newpokeaval -dProcedimento1 -uses -ne 1 to 1 newpokeaval -dProcedimento1 -pu -ne 1 to 1</i>
<i>newpokeaval -dProcedimento2 -nos -ne 1 to 1 newpokeaval -dProcedimento2 -arcs -ne 1 to 1 newpokeaval -dProcedimento2 -uses -ne 1 to 1 newpokeaval -dProcedimento2 -pu -ne 1 to 1</i>
(...)

Finalizada a adequação dos *script*, execute-os dando dois cliques sobre os mesmos. A execução dos scripts deve ser realizada na ordem de numeração indicada no nome dos mesmos. Os resultados de cobertura da ferramenta Poke-Tool são disponibilizados no diretório de cada procedimento, criados na raiz da pasta da seção durante a execução do primeiro *script*. Os resultados de cada critério são separados em arquivos distintos. A nomenclatura dos arquivos de resultados é dada por “nomeDoCritério”+output.tes (ex.: “nosoutput.tes”, “arcoutput.tes” etc).