LEONARDO BRODBECK CHAVES

UMA AVALIAÇÃO EMPÍRICA DE MÉTRICAS PARA PROGRAMAS ORIENTADOS A OBJETO NO CONTEXTO DE TESTE DE SOFTWARE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio

CURITIBA
2001

UFPR

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática -

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática do aluno ***Leonardo Brodbeck Chaves***, avaliamos o trabalho intitulado ***"Uma Avaliação Empírica de Métricas para Programas Orientados a Objeto no Contexto de Teste de Software"***, cuja defesa foi realizada no dia 21 de fevereiro de 2001. Após a avaliação, decidimos pela aprovação do candidato.

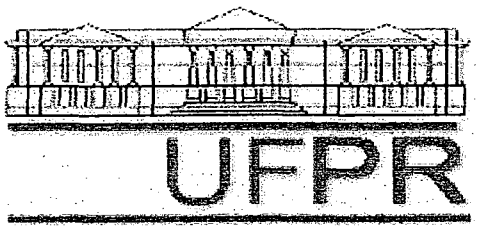
Curitiba, 21 de fevereiro de 2001.

Profª. Dra. Silvia Regina Vergilio
Presidente

Prof. Dr. Edmundo Sérgio Spoto
Membro Externo – UEM

Prof. Dr. Martin Alejandro Musicante
DINF/UFPR

LEONARDO BRODBECK CHAVES



Uma Avaliação Empírica de Métricas para Programas Orientados a Objeto no Contexto de Teste de Software

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre.
Curso de pós-graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profª. Drª. Silvia Regina Vergilio

CURITIBA – PR
2001

Abstract

The increasing complexity of software products, used in most areas of the human activity, has created many challenges to the development. In order to be able to deal with this complexity and assure the quality of software production, the object-oriented paradigm was created, and it is currently divulged and used, in academic and industrial environments.

With the purpose of monitoring efficiently the development of object-oriented systems, software metrics has been proposed these last years, as a way of supporting the search for a high quality software. The testing of software is an important activity, which can consume most cases 40% of the development efforts, and which can be more effective if conducted in a more organized way, following techniques and testing criteria and also if supported by the information provided by metrics.

The correlation between software metrics and the development aspects, as the difficulty of testing, can be determined many times, only through empirical studies.

This work presents the results of two experiments, accomplished in the context described above. The obtained results indicate a relation between the metric set selected for the experiment and the difficulty of testing. The difficulty of testing, in this case, is characterized as the difficulty to reveal faults through test cases. The metrics were collected from UML (Unified Modeling Language) artifacts and from the source code of the system developed elaborated for the experiments.

The obtained empirical results indicate the software characteristics, given by metrics, which influence the difficulty of testing. The obtained knowledge can by used, for example, for remodeling software and for generating more efficient test cases. Therefore, we can reduce the efforts spent on testing.

Resumo

A crescente complexidade dos produtos de software, utilizados em praticamente todas as áreas da atividade humana, tem gerado muitos desafios ao desenvolvimento. Para lidar com esta complexidade e garantir a qualidade na produção de software, o paradigma de orientação a objetos foi criado, e atualmente é largamente difundido e utilizado, nos ambientes acadêmicos e industriais.

Com o intuito de se monitorar de forma eficaz o desenvolvimento de sistemas orientados a objetos, métricas de software têm sido propostas nos últimos anos, de forma a apoiar a busca de um software de qualidade. O teste de software é uma atividade importante, que pode consumir até 40% dos esforços de desenvolvimento, e pode ser mais eficaz se for conduzido de forma organizada, segundo técnicas e critérios de teste, e se também for apoiado nas informações que as métricas podem dispor.

A correlação entre as métricas de software e aspectos do desenvolvimento, como a dificuldade de teste, podem ser determinados muitas vezes apenas através de estudos empíricos.

Este trabalho apresenta os resultados de dois experimentos realizados no contexto acima descrito. Os resultados encontrados apontam uma relação entre o conjunto de métricas selecionadas para o experimento e a dificuldade de teste. A dificuldade de teste, nesse caso, pode ser caracterizada como a dificuldade de se revelar defeitos através de casos de teste. As métricas foram coletadas a partir dos artefatos da UML (*Unified Modeling Language*) e do código fonte dos projetos elaborados para os experimentos.

Os resultados empíricos obtidos apontam as características do software, dadas pelas métricas, que mais influenciam na dificuldade de teste e podem ser utilizado, por exemplo, no remodelamento de um projeto de software e na geração de casos de teste mais eficazes, reduzindo os gastos das atividade de teste.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

1.1 Contexto

O desenvolvimento de sistemas orientados a objeto (OO) têm se tornado muito popular nos ambientes atuais de desenvolvimento. A partir da década de 80, houve uma explosão dos métodos orientados a objeto. Várias metodologias experimentaram diferentes abordagens para análise e projeto de software de acordo com o novo paradigma. Inicialmente alguns poucos métodos se firmaram. No meio da década de 90, alguns métodos de segunda geração começaram a surgir, mais notadamente o método de Booch, como uma evolução dos métodos OMT e Fusion [1]. Em 1995, como resultado da união dos métodos de Booch, OMT e Objectory, surgiu a UML (*Unifield Modeling Language*) [2], que engloba um conjunto de modelos para documentação e visualização dos diversos artefatos que compõem um sistema OO, proporcionando uma abordagem compatível com os novos conceitos do paradigma.

O desenvolvimento OO requer não apenas abordagem diferente para a análise, projeto e implementação, requer também nova abordagem em relação às métricas de software, haja visto que a tecnologia OO utiliza o *objeto*, e não o *algoritmo* ou os dados como bloco fundamental de construção.

Atualmente, encontram-se na literatura corrente métricas tradicionais que podem ser adaptadas ao paradigma [3], e também métricas estritamente OO que avaliam suas estruturas e conceitos [4, 5, 6].

Uma questão em aberto atualmente são os critérios para se escolher um bom conjunto de métricas. Nenhuma métrica pode ser considerada com certeza melhor que

outra, devido às características particulares de cada ambiente de desenvolvimento. Entretanto, algumas características práticas devem ser levadas em consideração [3]:

- Deve haver uma correlação significativa entre os valores das métricas e a taxa de defeitos ou o custo estimado;

- O conjunto de métricas deve permitir uma coleta automatizada, viabilizando a utilização de ferramentas, hoje indispensáveis ao desenvolvimento de software.

Outra questão relacionada ao desenvolvimento OO e às métricas é o teste de sistemas OO. Esta atividade é fundamental na garantia da obtenção da qualidade de tais sistemas. Recentemente, técnicas de teste tradicionais têm sido estendidas para o software OO, e existem três técnicas básicas que podem também ser aplicadas: i) Teste Funcional, que deriva casos de teste baseados apenas nos aspectos funcionais dos programas; ii) Teste Estrutural, que deriva casos de teste baseados no código fonte e geralmente oferecem medidas de cobertura; e iii) Teste Baseado em Erros, que se utiliza dos erros mais comuns em programação para derivar os casos de teste. Um critério baseado em erros bem conhecido é a Análise de Mutantes [7], que gera programas mutantes, diferentes do original apenas por uma modificação simples determinada pelo operador de mutação. A ferramenta denominada Proteum [7] suporta o teste baseado na Análise de Mutantes.

O teste funcional é mais freqüentemente utilizado em sistemas OO [8]. O modelo de casos de uso da UML especifica a funcionalidade dos sistemas OO, e é em geral utilizado para derivar os casos de teste [2].

Teste de software é uma atividade que tem como objetivo revelar a presença de defeitos no programa. Geralmente é uma atividade custosa e fundamental para aumentar a confiabilidade dos programas. Nesse contexto, métricas de software desempenham um papel bastante importante. Elas podem avaliar as principais características de um sistema que dificultam ou facilitam o teste, bem como, aumentam ou diminuem o número de defeitos gerados durante o desenvolvimento. A correlação entre as métricas de software e aspectos do desenvolvimento, como a dificuldade de teste, podem ser determinados muitas vezes apenas através de estudos empíricos.

Os estudos empíricos, concretizados através de experimentos, são importantes em medida que melhoram o entendimento das relações entre as métricas e a qualidade do software sendo desenvolvido.

Um problema encontrado atualmente é a falta de dados empíricos sobre métricas de software orientado a objeto. Os poucos trabalhos existentes na literatura, para software OO, basicamente exploram as relações entre as métricas de complexidade, acoplamento, coesão e herança com a facilidade ou dificuldade de detecção de defeitos e outras características relacionadas ao desenvolvimento [3, 17, 42].

1.2 Motivação

Dado o contexto acima, têm-se os seguintes fatores que constituem a motivação para o presente trabalho:

a) As métricas constituem um ponto importante com respeito à obtenção da qualidade de sistemas OO ou do processo de desenvolvimento;

b) Somente validações empíricas mostram a utilidade prática das métricas OO propostas;

c) Teste de software é uma atividade cara e a utilização de métricas pode reduzir os custos dessa atividade;

d) Na literatura existem poucos trabalhos com validações empíricas das métricas OO no contexto de teste de software;

1.3 Objetivos

O objetivo deste trabalho é apresentar resultados experimentais sobre a utilização de métricas para sistemas OO no contexto da atividade de teste. Para isso foram realizados dois experimentos para verificar a influência das métricas OO na atividade de teste. Essa influência é medida pela:

- Dificuldade para revelar defeitos nos programas durante o teste;
- Propensão dos sistemas a conter defeitos.

Os resultados empíricos obtidos através dos experimentos realizados contribuem para proporcionar uma melhor utilização das métricas no desenvolvimento OO, através de diretrizes para a redução da complexidade, permitindo um desenvolvimento melhor controlado, já que uma correlação mais clara entre as métricas e as características reais do software podem ser inferidas. Também podem contribuir na qualidade do desenvolvimento de sistemas OO, obtendo-se sistemas mais fáceis de testar.

1.4 Organização

Esse capítulo apresentou o contexto no qual este trabalho está inserido, a motivação para realizá-lo e o objetivo a ser atingido.

No Capítulo 2 são apresentados os principais conceitos do paradigma de orientação a objeto, as abordagens existentes de desenvolvimento e os principais modelos da UML (*Unified Modeling Language*). As métricas adaptadas ou específicas para sistemas OO também são apresentadas.

O Capítulo 3 traz uma revisão bibliográfica sobre o teste de software. Apresenta os conceitos gerais de teste de software, as técnicas mais difundidas, os critérios de teste e os aspectos principais do teste OO, bem como a revisão dos trabalhos da literatura que relacionam teste e métricas OO.

O Capítulo 4 apresenta uma avaliação empírica de métricas OO, através de dois experimentos realizados. O conjunto de métricas utilizadas, as hipóteses a serem avaliadas, a metodologia utilizada no desenvolvimento dos sistemas do experimento, a teoria estatística utilizada e uma análise dos resultados são apresentados.

O Capítulo 5 encerra o trabalho, apresentando as considerações finais, contribuições e trabalho futuros.

O trabalho contém 2 apêndices. No Apêndice A os operadores de mutação da ferramenta Proteum são listados. No Apêndice B foram transcritos os valores tabulados para o teste t, que são usados no teste de hipótese da validação das correlações estabelecidas pelos experimentos.

A Figura 2.11 mostra exemplos de cálculo para a complexidade ciclomática para quatro estruturas básicas de programação.

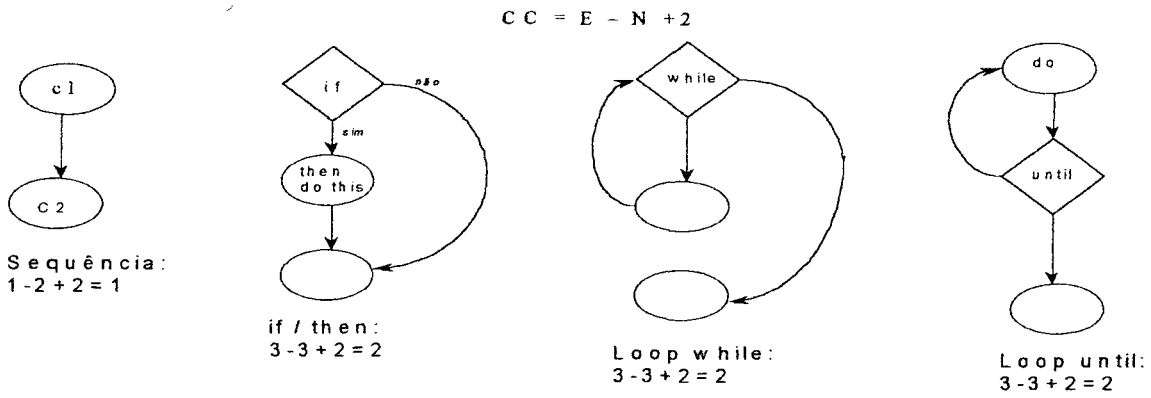


Figura 2.11: Exemplo de Cálculos para a Complexidade Ciclomática.

Um método com uma complexidade ciclomática baixa geralmente é melhor, pois isto pode significar menor volume de teste e maior facilidade de entendimento dos métodos.

A complexidade ciclomática não pode ser usada para medir a complexidade de uma classe devido à herança, mas a complexidade ciclomática individual dos métodos pode ser combinada com outras medidas para se avaliar a complexidade da classe.

Em relação à classe, o tamanho da classe é usado para avaliar a facilidade de entendimento do código pelos desenvolvedores. O tamanho da classe pode ser medido de diversas maneiras. Inclui-se a contagem de todas as linhas físicas de código, o número de linhas em branco e o número de linhas comentadas. A métrica *linhas de código* (LOC) conta todas as linhas do código. A métrica que avalia o número de linhas não comentadas e que não estão em branco é denominada de *linhas fonte de código*, ou NCNB. O resultado da métrica de declarações executáveis (EXEC) é o total de declarações executáveis sem levar em conta o número de linhas de código físicas. Por exemplo, no seguinte trecho de código em C é mostrado o cálculo de LOC, NCNB e EXEC.

```
1.  if (i>0)
2.  /* Se i>0, y recebe x/i */
3.  y=x/i;
4.  else
5.  {
6.  x=i;
7.  y=f(x);
8.  }
```

Figura 2.12: Trecho de Programa na Linguagem C.

Por exemplo, no programa de Figura 2.12, LOC=8, NCNB=7 e EXEC=3. O número de declarações executáveis, EXEC, é a medida menos influenciada pelo programador ou estilo da linguagem.

Valores empíricos para a avaliação da medida de tamanho variam de acordo com a linguagem de codificação e da própria complexidade do método. Entretanto, o tamanho dos programas afeta a facilidade de entendimento pelo desenvolvedor, e as classes e métodos de grande tamanho sempre possuirão um alto risco de falha [5].

A Análise por Pontos de Função, publicada originalmente 1979, por Albrecht [12], mede a funcionalidade do sistema independentemente da tecnologia empregada na implementação. Para tanto, os sistemas são representados em um modelo abstrato que contém os itens que contribuem para o tamanho funcional.. A técnica de pontos de função provê uma medida comparativa objetiva, que assiste o desenvolvimento no que tange à avaliação, planejamento e controle da produção de software. Para a utilização em sistemas OO, os conceitos aplicados no método de desenvolvimento devem ser mapeados no modelo abstrato. Uma proposta de mapeamento é descrita por Fetcke et al [53].

2.4.2 Métricas Específicas para Sistemas OO

Muitas métricas diferentes têm sido propostas para sistemas OO. As métricas escolhidas na literatura para discussão medem as estruturas que, se inapropriadamente projetadas, afetam negativamente a qualidade do projeto e do código.

2.4.2.1 Métricas Propostas por Chidamber e Kemerer

O conjunto de métricas propostas em [4] são as mais referenciadas na literatura corrente. Composto de seis métricas, este conjunto é dependente da linguagem de programação, necessitando portanto de adaptações e refinamentos para cada linguagem. Estas métricas foram avaliadas pelas meta-métricas de Weyuker [16] na sua proposta original. As meta-métricas de Weyuker são apresentadas na Seção 2.5 deste capítulo.

As métricas são apresentadas e discutidas sucintamente nas seções seguintes. Para que a discussão ocorra através de exemplos, na Figura 2.13 temos o diagrama de classes na notação UML, que retrata uma loja de departamentos.

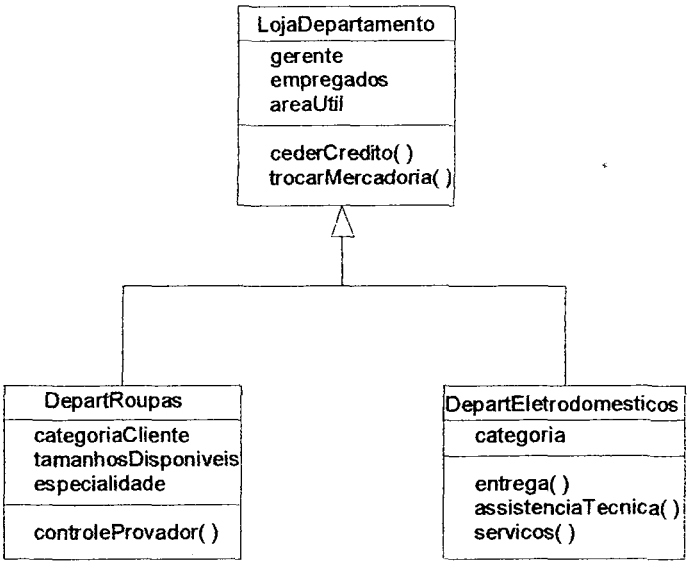


Figura 2.13: Exemplo de um Sistema OO.

2.4.2.1.1 Métodos Ponderados por Classe (*Weighted Method per Class* - WMC)

O métrica WMC corresponde ao número de métodos implementados na classe ou à soma da complexidade dos métodos, complexidade que pode ser medida através da complexidade ciclomática (CC). A segunda medida é mais difícil de se implementar, pois nem todos os métodos são acessíveis na classe devido à herança. O número de métodos e a

complexidade dos métodos podem predizer o esforço necessário para o desenvolvimento e manutenção da classe.

Um elevado número de métodos na classe implica em um impacto considerável nas classes derivadas. Classes com um grande número de métodos são geralmente específicas à aplicação atual, e isso limita a possibilidade de reuso.

Em relação à Figura 2.13, WMC pode ser calculada contando-se o número de métodos, logo:

$WMC_{\text{classe DepartRoupas}} = 1$

$WMC_{\text{classe Eletrodomésticos}} = 3$

2.4.2.1.2 Resposta para a Classe (*Response for a Class* - RFC)

A métrica resposta para a classe representa o número de todos os métodos que podem ser invocados em resposta a uma mensagem para um objeto de uma classe ou por algum método da classe. Isto inclui todos os métodos acessíveis dentro da hierarquia de classe. Portanto, esta métrica avalia a combinação da complexidade da classe através do número de métodos e do volume de comunicação com outras classes.

Quanto maior o número de métodos que podem ser invocados em uma classe como resposta a uma mensagem, maior será a complexidade da classe. Se um grande número de métodos podem ser invocados em resposta a uma mensagem, as atividades de teste e depuração da classe serão mais complexas, por requerer um nível de entendimento considerável por parte do testador. O pior caso possível de RFC obtido pode, por exemplo, ser útil na previsão do tempo de teste necessário.

A métrica RFC para a classe LojaDepartamento, é a soma do número de métodos que podem ser invocados em resposta às mensagens enviadas da própria classe LojaDepartamento (2), das classes derivadas DepartRoupas (1) e DepartEletrodomesticos (3). Temos então:

$RFC_{\text{classe Loja Departamentos}} = 2 + 1 + 3 = 6$

2.4.2.1.3 Falta de Coesão (*Lack of Cohesion* - LCOM)

A métrica falta de coesão, LCOM, mede a não similaridade dos métodos de uma classe através de variáveis de instância ou através de atributos. Um método com alta coesão indica uma boa subdivisão em classes, simplicidade do sistema e alta possibilidade do reuso das classes.

A falta de coesão ou baixa coesão aumenta a complexidade, portanto aumenta a ocorrência de defeitos durante o processo de desenvolvimento. Classes com baixa coesão poderiam provavelmente ser subdivididas em duas ou mais subclasses com coesão melhorada. A Figura 2.14 é um arranjo alternativo do projeto da Figura 2.13. Na Figura 2.14, as duas classes DepartRoupas e Eletrodomésticos foram eliminadas, e os e os métodos combinados na classe LojaDepartamento, formando um novo arranjo com baixa coesão, isto é, alto LCOM, porque existem relativamente poucos atributos e métodos comuns entre os objetos instanciados. Um objeto instanciado para o departamento de roupas não necessita do método assistenciaTecnica() e servicos(), assim como um objeto instanciado para o departamento de eletrodomésticos não necessita do método controleProvador().

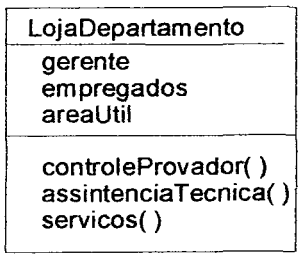


Figura 2.14: Arranjo Alternativo com Baixa Coesão.

2.4.2.1.4 Acoplamento entre Classes (*Coupling Between Object Classes* - CBO)

A métrica de acoplamento entre as classes, CBO, é a soma do número de classes a qual uma classe está acoplada. É medida levando-se em conta o número de classes distintas, não relacionadas pela herança, da qual uma determinada classe depende.

O uso de um acoplamento excessivo, em geral, implica em um sistema mal projetado e dificulta o reuso das classes. Quanto maior o número de elementos acoplados, mais sensível a efeitos colaterais será o sistema quando efetuadas mudanças, implicando em maior dificuldade para manutenção. Um acoplamento forte também dificulta a compreensão e inteligibilidade da classe, já que uma mudança ou correção está relacionada com outras classes.

A complexidade pode ser reduzida ao se projetar o sistema com o menor número possível de classes acopladas. Isto aumenta a modularidade e promove o encapsulamento.

A Figura 2.15 é um exemplo de objetos altamente acoplados. Dois departamentos são identificados através das classes Departamento de Jaquetas e Departamentos de Calças. Ambas possuem os mesmos atributos e métodos. Isto implica que o projeto da Figura 2.15 não é provavelmente o projeto mais eficiente e estes Departamentos deveriam estar combinados em uma mesma classe.

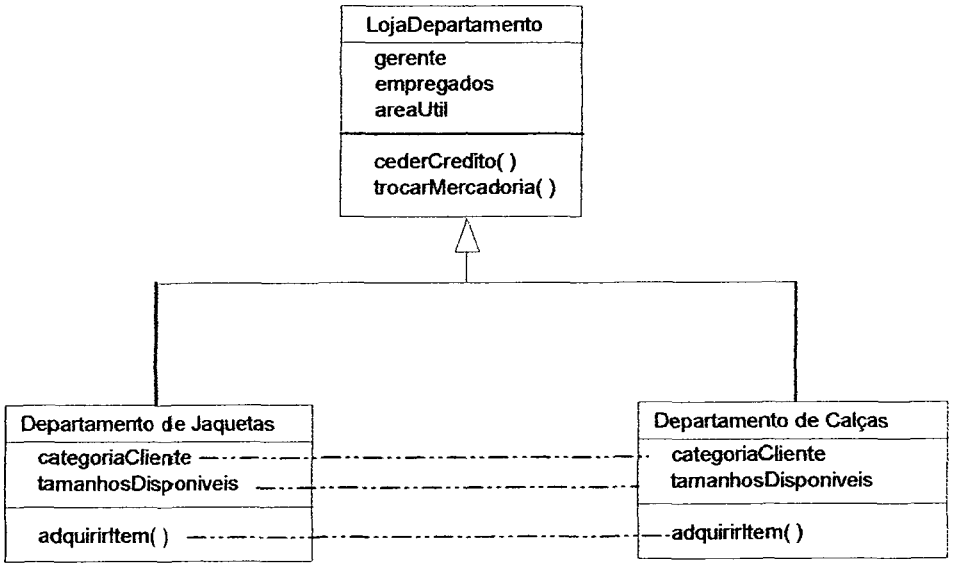


Figura 2.15: Exemplo de um Projeto com Acoplamento Excessivo.

2.4.2.1.5 Profundidade na Árvore de Herança (*Depht of Inheritance Tree* - DIT)

A profundidade de uma classe na hierarquia de herança (DIT) é o maior número de níveis da classe nó para a raiz da árvore.

Quanto mais abaixo estiver uma classe na hierarquia, maior será o número de métodos que a classe tipicamente herdará, tornando-a mais complexa para predição do seu comportamento.

Árvores muito profundas constituem projetos de elevada complexidade, já que mais métodos e classes estão envolvidos. Porém, maior será o potencial para reuso dos métodos herdados. Uma alternativa à métrica DIT seria o número de métodos herdados, NMI.

Na Figura 2.13, LojaDepartamento é a classe raiz e possui DIT=0 e NMI=0. Para a classe DepartRoupas e DepartEletrodomesticos, DIT = 1 e NMI=2.

2.4.2.1.6 Número de Subclasses (*Number of Children* - NOC)

O número de subclasses (NOC) avalia o número de subclasses imediatamente subordinadas a uma classe na hierarquia. É um indicativo da possível influência que uma classe pode ter no projeto e sobre o sistema em si.

Um grande número de subclasses indica uma abstração imprópria da superclasse. Entretanto, quanto maior o número de subclasses, maior o reuso, pois a herança em si é uma forma de reuso. Se a classe possui um grande número de subclasses, pode requerer um volume maior de teste dos seus métodos, aumentando o tempo requerido para teste. Na Figura 2.13, a classe LojaDepartamento possui NOC=2. Para a classe DepartEletrodomesticos, NOC=0, já que é uma terminação, ou nó folha na estrutura árvore.

2.4.2.2 Métricas de Lorenz e Kidd para Sistemas OO

As métricas propostas em [5] são compostas por trinta e oito métricas, subdivididas em duas categorias: a primeira categoria, composta de nove métricas, retrata a situação atual de um projeto OO em andamento, medindo aspectos dinâmicos referentes à situação de um projeto dentro do ciclo de desenvolvimento; a segunda categoria, composta de vinte e nove métricas, mede aspectos estáticos dentro do ciclo de desenvolvimento, procurando avaliar a qualidade durante a implementação.

A Tabela 2.2 relaciona as métricas de desenvolvimento com uma descrição sucinta de cada medida.

Tabela 2.2: As Métrica de Lorenz e Kidd para Desenvolvimento de Sistemas OO.

Métrica	**Descrição sucinta**
Número de mensagens enviadas pelo método	Total de mensagens enviadas pelo método
Número de declarações do método	É dependente da linguagem de programação
LOC	Número de linhas ativas no código no método
Tamanho médio do método	Pode ser a média da métrica LOC em todos os métodos.
Complexidade do método	Soma ponderada das seguintes características do método: ❑ chamadas API (interface para programação de aplicativos) ❑ atribuições ❑ expressões binárias ❑ mensagens com parâmetros ❑ expressões aninhadas ❑ parâmetros ❑ chamadas primitivas ❑ variáveis temporárias ❑ expressões sem parâmetros
Número de métodos de instância pública na classe	Medida do nível de responsabilidade da classe que são instanciados.
Número de métodos de instância	Total de métodos na classe.
Número médio de métodos instanciados por classe	Média do número de métodos nas classes.
Números de variáveis instanciadas na classe	Esta medida exclui as variáveis públicas por decisão dos autores, pois as variáveis públicas rompem o princípio de encapsulamento.
Número médio de variáveis instanciadas por classe	Média das variáveis instanciadas nas classes
Número de métodos na	Número de métodos na classe, não em suas

classe	instâncias.
Número de variáveis de classe em uma classe	Número de variáveis globais locais que provêem objetos comuns para todas as instâncias da classe.
Nível de aninhamento da hierarquia de classe	Indica o maior nível de aninhamento de todas as classes; a profundidade do grafo na hierarquia de classes
Número de classes abstratas	Número total de classes abstratas
Uso de herança múltipla	Função métrica boleana que indica a presença ou não de herança múltipla
Número de métodos com o mesmo nome na subclasse (*overriding method*)	
Números de métodos herdados pela subclasse	
Número de métodos adicionados pela subclasse	Medida do número de novos métodos adicionados pelas subclasses.
Coesão da classe	Mede a inter-relação entre os métodos da classe e o padrão de uso das variáveis usadas pelos métodos da classe.
Usos globais	Mede a quantidade de objetos disponíveis para todos os objetos no sistema.
Número médio de parâmetros por método	
Uso de funções *friends*	Mede o uso de funções *friends* que rompem o encapsulamento.
Porcentagem de código orientado a funções	Mede a quantidade de código escrito fora das classes
Número médio de linhas de comentários por método.	
Número médio de métodos comentados	
Número de problemas reportados pela classe ou contrato	Número total de problemas anotados durante o desenvolvimento do diagrama de classes e pelos contratos
Acoplamento da classe	Avalia as conecções entre as classes através das mensagens existentes entre elas
Número de vezes em que uma classe é reutilizada	Número de reutilizações de uma determinada classe pré-existente em novos projetos
Número de classes ou método *jogados fora*	Número de classes e/ou métodos descartados durante o desenvolvimento.

Valores empíricos de cada métrica são descritos baseados na própria experiência dos autores. Entretanto, uma calibração local se faz necessária para se captar as particularidades de cada ambiente de desenvolvimento.

2.4.2.3 Métrica de Complexidade para Programas OO de Lee Et Al

As métricas propostas por [6] são baseadas no fluxo de dados e nas métricas de complexidade de Halstead [3], partindo-se dos seguintes princípios:

- Um objeto armazena informação;
- As mensagens entre objetos podem ser vistas como instrumentos de fluxo de informação assim como a referência a variáveis;
- A herança consiste em um fluxo implícito de informação da classe para as suas subclasses.

Para derivar seu conjunto de métricas, os autores se utilizaram de dois elementos: o tamanho de uma entidade, representando a complexidade interna e o *fan-in x fan-out*, representando a complexidade externa. De acordo com a definição de Henry e Kafura, *fan-in* de um módulo M é o número de fluxos de informação que terminam em M mais o número de estruturas de dados nas quais a informação é armazenada por M. Similarmente, *fan-out* de um módulo M é o número de fluxos locais que emanam de M mais o número de estruturas de dados atualizados por M [3].

A expressão geral para as métricas propostas por [6] pode ser escrita como:

$$LN(E) \times CP^2(E) = LN(E) \times [(IP(E) + OP(E)]^2, \text{ onde}$$

LN(E) é o tamanho da entidade E medido por Halstead, de acordo com a definição encontrada em [3];

CP(E) é o acoplamento de saída entre E e as entidades externas — para o método em particular, seria basicamente o número de argumentos do método E;

IP(E) é o acoplamento de entrada de E — para o método E, seria basicamente o total de argumentos referenciados em outros métodos;

OP(E) é o acoplamento de saída de E — para o método, é o número de variáveis referenciadas pelo próprio método.

As métricas propostas medem a complexidade do método (MC), da classe (CC), da complexidade da hierarquia (HC) e a complexidade do programa (PC).

Este conjunto de métricas foi avaliado formalmente segundo as meta-métricas de Weyuker. Foi realizado no mesmo trabalho uma avaliação empírica usando programas C++, e foi verificada a necessidade de pequenas adaptações nas regras de contagem.

As métricas são apresentadas na Tabela 2.3.

Tabela 2.3: As Métrica de Lee et al baseadas em fluxo de dados para sistemas OO.

Complexidade do Método (MC)	$MC(M) = LN(M) \times CP(M)^2$, onde LN(M) = tamanho do método M = número total de operadores e operandos em M CP(M) = acoplamento do método M = IP(M) + OP(M)
Complexidade da Classe (CC)	É definida de duas maneiras: 1) $CC(C) = \sum_{i=1}^{NCM(C)=n} (LN(M_i) \times CP^2(M_i))$ onde $LN(M_i)$ = tamanho de método M_i; NCM(C) = n = número de métodos componentes da classe C CP(M i) = acoplamento do método M_i 2) $CC(M) = LN(C) \times CP(M)^2$ onde LN(C) = tamanho de classe C = soma do tamanho dos métodos componentes da classe C CP(C) = acoplamento da classe C
Complexidade da Hierarquia (HC)	É definida de três maneiras: 1) $HC(H) = \sum_{i=1}^{NMD(H)} (LN(M_i) \times CP(M_i)^2$ 2) $HC(H) = \sum_{i=1}^{NCD(H)} (LN(C_i) \times CP(C_i)^2$ 3) $HC(H) = LN(H) \times CP(H)^2$
Complexidade do Programa (PC)	É definida de três maneiras: 1) $PC(P) = MPC(P) + \sum_{i=1}^{NMD(P)} (LN(M_i) \times CP(M_i)^2$ 2) $PC(P) = MPC(P) + \sum_{i=1}^{NCD(P)} (LN(C_i) \times CP(C_i)^2$ 3) $PC(P) = MPC(P) + \sum_{i=1}^{NHD(P)} (LN(H_i) \times CP(H_i)^2$

2.5 Estudos Teóricos e Empíricos de Métricas OO

Na literatura encontram-se alguns trabalhos reportando resultados de estudos teóricos e empíricos com métricas para software OO. Esses resultados foram utilizados para propor recomendações para:

- Aprimoramento de algumas métricas, como é o caso dos estudos de Eder [15] com os conceitos de coesão e acoplamento;
- Implementação de ferramentas e coleta automática de métricas, tais como o trabalho de Sichonary [54];
- Propor propriedades a serem utilizadas para avaliação tais como o trabalho de Weyuker [16];

Weyuker, em [16], propôs nove propriedades para a avaliação das métricas de complexidade. A aplicabilidade destas propriedades tem sido criticada, embora a sua trivialidade e simplicidade a fizessem popular para algumas métricas recentemente propostas. As nove propriedades são ilustradas na Tabela 2.4, onde u é uma função métrica e P, Q, e R são entidades de programa.

Tabela 2.4: As Meta-Métrica de Weyuker.

Propriedade 1:	$(\exists P)(\exists Q)$ tal que $u(P) \neq u(Q)$
Propriedade 2:	Existe apenas um número finito de programas com um número c de complexidade não-negativo.
Propriedade 3:	$\exists P, \exists Q, P \neq Q$ tal que $u(P) = u(Q)$
Propriedade 4:	$(\exists P)(\exists Q)$ tal que $((P \equiv Q) \& u(P) \neq u(Q))$
Propriedade 5:	$(\forall P)(\forall Q)$ tal que $u(P) \leq u(P+Q), u(Q) \leq u(P+Q)$
Propriedade 6:	$(\exists P)(\exists Q)(\exists R)$, $u(P) = u(Q)$ não implica que $u(P+R) = u(Q+R)$ e $u(R+P) = u(R+Q)$
Propriedade 7:	Existem programas P e Q tal que Q é formado pela permutação da ordem das declarações de P, e $u(P) \neq u(Q)$
Propriedade 8:	Se P é o renome de Q, então $u(P) = u(Q)$
Propriedade 9:	$(\exists P)(\exists Q)$ tal que $u(P)+u(Q) < u(P+Q)$

A propriedade 1 estabelece que uma métrica não deveria fazer todo programa ter a mesma complexidade. As propriedades 2 e 3 estabelecem que uma métrica não deveria ser muito grosseira nem muito rigorosa. A propriedade 4 estabelece que a implementação de uma função pode afetar sua complexidade. A propriedade 5, a propriedade da monotonicidade, estabelece que um componente de um programa sempre é mais simples que o programa inteiro. A propriedade 6 estabelece que dois programas com a mesma complexidade não precisam ter a mesma complexidade depois de concatenados com um terceiro programa. A propriedade 7 estabelece que a permutação de declarações pode afetar a complexidade. A propriedade 8 estabelece que um programa renomeado não tem sua complexidade afetada. Por fim, a propriedade 9 estabelece que a interação entre dois subprogramas pode aumentar a complexidade.

Briand et al [55], apresenta uma metodologia para validação de atributos internos de software. O método proposto trata de dois tipos de validação: teórica (demonstração de que uma medida está de fato avaliando a característica a que se propôs) e empírica (verificação de que uma medida é útil em um ambiente de desenvolvimento particular). Mais detalhes sobre este estudo são fornecidos na Seção 3.6.

2.6 Considerações Finais

Este capítulo tratou dos conceitos básicos de OO, de metodologias existentes e métricas, necessários ao entendimento do paradigma.

A abordagem UML foi descrita com mais detalhe pois foi utilizada nos experimentos descritos no Capítulo 4. Nesse sentido, a Seção 2.4 é bastante importante pois ela auxilia o entendimento das métricas e hipóteses utilizadas nesse trabalho.

Como nosso objetivo é estudar as métricas no contexto de atividade de teste, o capítulo seguinte é dedicado a essa atividade.

Capítulo 3

Teste de Software

Neste capítulo são apresentados alguns conceitos envolvendo a atividade de teste em geral, e em particular o teste de programas OO, tais como terminologia e critérios correspondentes. São feitas considerações sobre a importância do teste de software durante o processo de desenvolvimento, em seguida são apresentadas as principais técnicas e critérios de teste.

Aspectos importantes relativos ao teste de programas OO são discutidos devido a sua importância nesse trabalho. Por último, estudos empíricos sobre métricas de programas OO no contexto de teste e que serviram como base para o trabalho descrito são apresentados.

3.1 Definição de Erro, Defeito e Falha

A IEEE tem realizado vários esforços de padronização, incluindo a terminologia usada em teoria de teste. O padrão IEEE [59] é utilizado para definir os seguintes termos: defeito (*fault*) — problema no código fonte do programa; engano (*mistake*) — ação humana que produz um resultado incorreto; erro (*error*) — diferença entre o valor obtido e o valor esperado; e falha (*failure*) — produção de uma saída incorreta com relação à especificação.

Segundo Howden [43], os erros de programa se classificam em: erros de domínio, erros de computação e erros de estruturas de dados. Na literatura existem deferentes classificações para erros e para defeitos. [43, 44, 45]. A classificação de Howden [43] está resumida abaixo:

- Erros de domínio:

O domínio de um caminho é formado por todas as entradas que executam tal caminho, sendo que cada domínio corresponde a uma função. Um erro de domínio ocorre se um domínio incorreto for associado a um caminho. Existem dois tipos de erro de domínio: erro na seleção de caminho — existe um erro em um predicado alguma parte do caminho e um caminho incorreto é executado; erro de caminho ausente — não existe um caminho no programa correspondente a uma parte do domínio de entrada ou especificação do programa.

- Erros de computação:

O erro provoca uma computação incorreta mas o caminho executado é igual ao caminho esperado. Erros de computação se dividem em dois tipos: a computação existe no programa mas existe um defeito; e a computação está ausente no programa.

- Erros de estrutura de dados:

Um erro em estruturas de dados corresponde à definição incorreta de uma estrutura de dados.

A terminologia definida nessa seção será utilizada ao longo de todo o trabalho.

3.2 O Teste de Software

A etapa de teste é de grande importância para a redução de erros no software, e representa a última etapa no processo de espeficicação, projeto de codificação [11], podendo ser realizado em etapas, a saber:

- Teste de unidade: procura identificar erros de lógica e de implementação em cada unidade do software;
- Teste de integração: técnica sistemática para se integrar os módulos componentes do software, de modo a identificar erros de interface entre os diferentes módulos;
- Teste de validação: testa os requisitos de software da fase de análise;
- Teste de sistema: realizado após a integração do sistema, visando a identificar erros de funções e características de desempenho, conforme uma especificação.

De forma geral, a realização do teste é composta basicamente das seguintes atividades:

- Construção dos casos de teste;
- Execução do programa com os dados dos casos de teste;
- Análise do comportamento do programa para a verificação da existência de erros.

A seqüência acima é repetida até que se tenha uma confiança suficiente no comportamento esperado do sistema.

Idealmente, um programa deveria ser testado com todas as combinações do domínio de entrada, mas o teste exaustivo não é viável na prática, devido a restrições de tempo e custo. Esse fato é a principal motivação para o desenvolvimento de diferentes técnicas e critérios de teste, que apoiam a construção de um conjunto de casos de teste com o objetivo de se revelar o maior número de erros possíveis, de modo a se ter um controle racional do tempo e custo a serem despendidos nesta atividade.

3.3 Técnicas de Teste

Existem na literatura três técnicas de teste básicas: a funcional, a estrutural e a baseada em erros [11]. O que difere essas técnicas é a origem da informação utilizada na avaliação e construção dos casos de teste, e cada técnica está associada a um conjunto de critérios para esse fim [18]. Essas três técnicas básicas são complementares, devendo, de preferência, ser aplicadas em conjunto, de modo a assegurar a qualidade do teste, e consequentemente a qualidade do produto de software final.

3.3.1 Técnica Funcional

No teste funcional, ou teste de caixa preta, o testador tem acesso apenas aos dados de entrada e saída, de modo que as funções do sistema são exercitadas sem se preocupar com detalhes de implementação.

Segundo [11], os critérios de teste funcional mais conhecidos são:

- Particionamento em classes de equivalência: divisão do domínio de entrada de um programa em classes, a partir das quais os teste são derivados, de modo a se

minimizar o número de casos de teste. Neste caso, gera-se apenas um caso de teste para cada classe;

- Análise do valor limite: procura testar os limites de cada classe de equivalência;

- Grafo de causa e efeito: o efeito combinado de dados de entrada, causas e efeitos são identificados em um grafo a partir do qual uma tabela de decisão é derivada, de tal maneira que os dados de teste e as saídas são derivados da tabela de decisão obtida.

3.3.2 Técnica Estrutural

O teste estrutural, ou teste caixa branca, leva em consideração a implementação na derivação dos casos de teste. Assim, no grafo de fluxo de controle associado ao programa, determinados elementos são exercitados por caminhos do grafo de fluxo, estabelecendo critérios de teste estrutural.

O teste estrutural e o teste funcional são complementares: o primeiro assegura que uma porcentagem desejada do código fonte seja testada pelo menos uma vez e o segundo garante que o software implementa corretamente os requisitos.

Um grafo de fluxo de controle nada mais é que um grafo orientado, com nó de entrada e nó de saída únicos, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro.

Os critérios estruturais fundamentam-se em diferentes tipos de informação elementares da estrutura de programa para determinar quais partes do programa são requeridas na execução, e os critérios mais utilizados são citados a seguir:

3.3.2.1 Critérios Baseados em Complexidade

Utilizam informações sobre a complexidade do programa para obter os requisitos de teste. Um critério bastante conhecido é o critério de McCabe [11, 46], que utiliza a complexidade ciclomática para derivar os requisitos de teste, descrito a seguir:

- Critério todos caminhos linearmente independentes: exige que um conjunto de caminhos independentes do programa seja executado. Um caminho independente deve incluir no mínimo um arco do grafo de fluxo de controle que não tenha sido percorrido por qualquer outro caminho do conjunto. O número de caminhos independentes do programa é dado pela complexidade ciclomática do programa.

3.3.2.2 Critérios Baseados em Fluxo de Controle

Características de controle da execução do programa, tais como comandos ou desvios, são utilizados para determinar quais estruturas são necessárias. Os critérios mais comuns são:

- Critério todos-nós: exige que cada nó do grafo seja exercitado pelo menos uma vez;

- Critério todos-ramos, que exige que cada arco do grafo seja exercitado pelo menos uma vez;

- Critério todos-caminhos, que exige que ao menos uma execução de cada caminho (seqüência finita de nós) do grafo.

3.3.2.3 Critérios Baseados em Fluxo de Dados

Os requisitos de teste são obtidos através das informações do fluxo de dados do programa. Uma característica comum desses critérios é que eles requerem a execução de caminhos a partir de definição da variável até onde ela foi utilizada. Esses critérios são mais exigentes que o critério todos-ramos, porém menos exigentes que o critério todos-caminhos, que geralmente é impraticável, pois um programa pode conter infinitos caminhos na presença de laços.

Para derivar os requisitos de teste requeridos por esses critérios é necessário adicionar ao grafo de fluxo de controle informações sobre o fluxo da dados, que consiste no Grafo Def-Uso (Def-Use Graph) definido por Rapps e Weyuker [47, 48]. Neste grafo são exploradas as associações entre a definição (atribuição de um valor à variável) e o uso (uso do valor previamente atribuído) das variáveis determinando os caminhos a serem exercitados.

Na literatura são encontrados diversos critérios de teste baseados em fluxo de dados [57,58]. Destacam-se os critérios de Rapps e Weyuker [47,48] e a família de critérios *Todos Potenciais Usos* de Maldonado [49].

3.3.3 Técnica Baseada em Erros

Utiliza os tipos de erros cometidos mais freqüentemente em programação para derivar os requisitos de teste. A idéia é de se utilizar os erros que o programador comete durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. A Análise de Mutantes e a Semeadura de Erros são critérios que se concentram em erros [51].

3.3.3.1 Critério Análise de Mutantes

A análise de mutantes é um critério que utiliza um conjunto de programas ligeiramente modificados, denominados mutantes, obtidos a partir de um programa P, de modo a avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é encontrar um conjunto de casos de teste T capaz de revelar diferenças de comportamento existentes entre P e seus mutantes [7].

A idéia central do critério Análise de Mutantes foi originalmente proposta por DeMillo em 1978 [20], também chamada de hipótese do programador competente. Tal idéia assume que programadores experientes escrevem programas muito próximos do correto. Assim, os defeitos nos programas são introduzidos através de pequenos desvios sintáticos e produzem um resultado incorreto.

Evidentemente, o objetivo é de se revelar tais defeitos. Programas mutantes são gerados a partir do programa em teste. Casos de teste são derivados para diferenciar o comportamento do programa original do comportamento do programa mutante. O mutante é morto quando ocorre essa diferenciação. O critério Análise de Mutantes exige que todos os mutantes sejam mortos.

Outra hipótese explorada na aplicação do critério Análise de Mutantes é o efeito de acoplamento [20], que se fundamenta na suposição de que defeitos complexos estão relacionados com defeitos simples. Dessa maneira, casos de teste que de fato revelam defeitos simples seriam também capazes de revelar defeitos mais complexos.

Durante a Análise de Mutantes um programa P é testado com um conjunto de casos T cuja adequação deseja-se avaliar. O programa é executado com T e, se apresentar resultados incorretos, então um defeito foi encontrado e o teste termina. Caso contrário, o programa ainda pode conter defeitos que o conjunto T não foi capaz de revelar. O programa P sofre alterações, gerando os programas P_1, P_2, P_3, ..., P_n que são mutantes de P. Os mutantes são gerados a partir de um conjunto de operadores que são específicos para uma linguagem de programação, conjunto esse que procura retratar e inserir mutações baseadas nos defeitos mais comuns.

O passo seguinte é a execução dos mutantes com o mesmo conjunto de casos de teste T. Caso o comportamento do mutante P_i seja diferente de P, então esse mutante é considerado morto. Caso contrário, existem duas razões para esse mutante estar vivo:

a) O conjunto de casos de teste T não é suficiente para distinguir o comportamento de P e P_i e novos casos de teste devem ser incluídos ao conjunto; ou
b) P_i é considerado equivalente a P, ou seja, não existe um caso de teste capaz de matar P_i.

Segundo [20], um conjunto de casos de teste é adequado a um programa em teste depois que todos os mutantes vivos foram mortos ou determinados como equivalentes.

3.3.4 Ferramentas de Teste

A atual e crescente complexidade do software exige a utilização de ferramentas no suporte a atividade de teste. A ferramenta auxilia na sistematização e automatização dos testes, diminuindo a propensão a erros e a possibilidade prática de se testar programas grandes e complexos, antes inviável pela aplicação manual dos critérios.

Pode-se destacar a ferramenta Poke-Tool [22], desenvolvida no DCA/FEEC/Unicamp para possibilitar a aplicação dos critérios Potenciais-Usos baseados em fluxo de dados. Esta ferramenta permite a execução de um conjunto de casos de teste e faz a análise de adequação do conjunto executado em relação a esses critérios. Características importante: multi-linguagem (pode ser configurada para diversas linguagens como C, Pascal e COBOL).

A Proteum (Program Testing Using Mutants) [7], desenvolvida no Instituto de Ciências Matemáticas e de Computação – ICMC-USP, é uma ferramenta que apoia a

utilização do Critério Análise de Mutantes para programas escritos na linguagem C. Devido a sua característica multi-linguagem, esta ferramenta pode ser configurada para o teste de programas escritos em outra linguagens de programação. A seguir são listados os principais recursos desta ferramenta:

- Definição de casos de teste;
- Execução do programa em teste;
- Seleção dos operadores de mutação que serão utilizados para a geração dos mutantes;
- Geração dos mutantes;
- Execução dos mutantes com os casos de teste definidos;
- Análise dos mutantes remanescentes e cálculo do escore de mutação.

Foram definidos na Proteum 71 operadores de mutação. Esses operadores descrevem erros típicos em programação, e foram utilizados nesse trabalho, e por isso são listados. Um exemplo de operador de mutação é o SWDD, que significa troca o comando *while* pelo comando *do-while*. Os demais operadores podem ser consultados no Apêndice A.

3.5 Teste Orientado a Objeto

3.5.1 Aspectos Gerais

Apesar de existirem algumas similaridades do teste OO com o teste de software convencional estruturado, existem diferenças e essas diferenças trazem novas questões à atividade de teste. Os recursos da orientação a objeto tais como encapsulamento, herança, polimorfismo e ligação dinâmica representam um novo desafio imposto às fases de teste e manutenção [32, 33]. Devido ao encapsulamento, as interações entre dois ou mais objetos ficam implícitas no código, dificultando o seu entendimento, e consequentemente a elaboração dos casos de teste para exercitar tais interações [34]. A herança de métodos testados previamente e considerados como corretos implica em um reteste dos mesmos devido ao novo contexto em que a subclasse se insere [35]. O polimorfismo e ligação dinâmica dificultam o teste, pois o tipo exato dos dados e a implementação dos métodos não podem ser determinados estaticamente, e, como já foi dito, o controle de fluxo de um programa OO é menos transparente [36].

Existe um consenso na literatura de que a classe é a unidade natural para o projeto de casos de teste. Outra unidade de teste é a agregação de classes: conjuntos de classes e pequenos subsistemas [5, 8, 17].

O teste de uma instância de uma determinada classe (objeto) exercita a classe isoladamente, e a isso damos o nome de Teste de Unidade. Entretanto, depois que as classes são testadas individualmente, o sistema como um todo, composto das diversas instâncias de classes (ou objetos) interagindo entre si, deve ser testado. A esse teste damos o nome de Teste de Integração [11].

3.5.2 Teste Funcional ou Caixa Preta para Sistemas OO

Existe um consenso quase que geral entre os autores que o teste funcional é útil para sistemas OO. Jacobson apresenta uma estratégia geral básica para o teste funcional, baseada em técnicas de teste já estabelecidas [39]. Uma estratégia de derivação de casos de teste baseadas em tentativa e erro é apresentada em [35]. Vários sistemas de pesquisa têm sido desenvolvidos para explorar como a verificação de tipos abstratos de dados (ADTs) pode ser adotado em sistemas OO. A abordagem ADT requer *drivers* de teste separados da aplicação. Algumas abordagens requerem informações sobre a especificação, enquanto outras são baseadas estritamente no código. A metodologia para projeto de teste FREE [40] produz um arranjo de teste integrado para o teste de unidade, subsistema e sistema para qualquer modelo OO de análise e projeto [41].

O modelo de casos de uso tem sido utilizado para geração de casos de teste funcionais. A partir deles, diversos cenários mostrando instâncias de usos do sistema são gerados e utilizados como entradas para o sistema.

Mesmo especificações bem desenvolvidas não expressam (e nem deveriam) cada detalhe de implementação, e assim alguma inspeção do código fonte é necessária. Por exemplo, o reteste de funcionalidades herdadas requer um exame da estrutura de classe. A distância entre especificação OO e a implementação OO é tipicamente pequena, quando comparada a sistemas convencionais. A distância entre os testes funcionais e estruturais, portanto, é diminuída sob este paradigma

3.5.3 Teste Estrutural ou Caixa Branca para Sistemas OO

Esta técnica de teste pode ser adaptada para o teste de métodos, mas não é suficiente para o teste das classes. A aplicabilidade do fluxo de controle através de um grafo orientado é questionada por diversos autores [35, 37, 38]. As duas razões mais citadas são:

a) Abordagens baseadas em grafos de fluxo de controle são inconsistentes com o paradigma de OO;
b) Erros em métodos não são os mais comuns.

Um ponto positivo desta técnica é a derivação dos casos de teste a partir exclusivamente da implementação, sem considerar a especificação do sistema. Isso é um ponto positivo, já que uma nova perspectiva em relação aos casos de teste é empregada.

3.6 Trabalhos Empíricos Relacionados

As técnicas de teste descritas nesse capítulo e utilizadas para software OO têm como objetivo facilitar o teste e reduzir os custos dessa atividade bem como aumentar a eficácia dos dados gerados. A utilidade de tais técnicas só pode ser avaliada através de estudos empíricos. Esses estudos também podem auxiliar a entender a atividade e a aprimorar as técnicas no contexto de OO.

Nesses experimentos, a utilização de métricas é fundamental. Embora seja muito importante a validação empírica, poucos trabalhos relacionam métricas com a atividade de teste. Entre esses trabalhos destacam-se:

O estudo feito por Briand et al [55] propôs uma metodologia para a validação de atributos internos de software, tratando de duas validações: uma teórica e outra empírica. A validação teórica se ateve ao objetivo de demonstrar que uma dada técnica está de fato medindo o conceito a que se propôs medir. Em síntese, a validação teórica, segundo este estudo, requer que:

- Um sistema de relação empírica seja definido, isto é, através de suas definições das propriedades e baseado em operações e relações entre entidades;

- Um sistema formal relacional adequado seja definido;

- As propriedades do sistema relacional empírico sejam preservados pelo sistema relacional formal quando a medida a ser validada é usada para o mapeamento entre eles.

Uma vez que uma medida pode ser considerada válida do ponto de vista teórico, as medidas precisam ser validadas empiricamente, já que o modelo teórico por si só não garante a validade das medidas. Então um modelo empírico foi proposto.

A validação empírica, segundo os autores, determinam a utilidade de uma dada medida em um ambiente de desenvolvimento particular. Os estudos empíricos correlacionam um atributo interno com um atributo externo.

Em Briand et al [42] foi explorado o relacionamento existente entre os conceitos OO de acoplamento, coesão, herança e probabilidade de existir defeitos nas classes do sistema durante as atividades de teste. Um estudo estatístico sofisticado foi realizado, com o intuito de validar as hipóteses propostas naquele experimento (um total de 8 hipóteses). O estudo foi conduzido com o objetivo de melhorar o entendimento do relacionamento existente entre medidas de projeto OO e a qualidade do software sendo desenvolvido. Os resultados encontrados indicaram que muitas das medidas consideradas capturam dimensões similares nos dados analisados, logo refletindo o fato de que muitas delas estão baseadas em princípios e hipóteses similares. Entretanto, é mostrado que usando um subconjunto de medidas, modelos precisos podem ser construídos para predizer em quais classes os erros têm maior propensão a estar. Segundo os autores, inspecionando-se um total de código comparável ao contido em classes com erro, é possível encontrar entre 90 e 95% dos defeitos, dependendo do tipo de modelo de validação considerado. Além do tamanho da classe, o número de mensagens enviadas/recebidas e a profundidade na hierarquia de herança parecem ser os principais fatores de propensão a erros em classes.

As investigações empíricas de Basili et al [17] apresentam os resultados de um estudo sobre o conjunto de métricas OO de Chidamber e Kemerer, propostas em [4]. O objetivo do trabalho de Basili é estudar a propensão de uma classe de conter defeitos a partir dos dados coletados por essas métricas. Dessa maneira, as métricas funcionariam como indicadores de qualidade durante o desenvolvimento. Para conduzir a validação, os autores coletaram dados de desenvolvimento de oito projetos médios baseados nos mesmos requisitos. Todos os projetos foram desenvolvidos usando um modelo de ciclo de vida seqüencial, um método bem conhecido de análise e projeto OO e a linguagem de programação C++.

Baseados na análise quantitativa e empírica dos dados coletados, as vantagens e desvantagens das métricas selecionadas são então discutidas. Como conclusão, os autores afirmam que algumas das métricas propostas por Chidamber e Kemerer parecem ser úteis na predição de classes propensas a conter defeitos durante as fases iniciais do ciclo de vida do software. E ainda tais métricas são melhores indicativos que as métricas de código *tradicionais*, que podem apenas ser coletadas em fases mais adiantadas do desenvolvimento.

3.7 Considerações Finais

Este capítulo tratou inicialmente de definições básicas da terminologia empregada no contexto de teste de software, além das principais técnicas e critérios existentes na literatura. Com relação ao teste para sistemas OO verificou-se a necessidade de adaptações das técnicas aos novos conceitos introduzidos pelo paradigma e que, infelizmente, a necessidade e a dificuldade de teste em nada diminuíram, constituindo ainda uma tarefa onerosa e difícil..

Na literatura existem poucos trabalhos relacionando métricas para software OO e a atividade de teste. Isso talvez se deva ao fato de que técnicas de teste específicas para software OO começaram aos poucos a serem propostas e ainda não alcançaram um estágio maduro. Os principais trabalhos desenvolvidos nesse contexto foram descritos. Eles serviram como base para o trabalho aqui descrito, composto de dois experimentos detalhados no capítulo a seguir.

Capítulo 4

Avaliação Empírica de Métricas para Software OO

Com o objetivo de avaliar empiricamente as métricas para software orientado a objeto, no contexto da atividade de teste, dois experimentos foram realizados, para se obter uma avaliação da influência de características de sistemas orientados a objeto nesta atividade. Tais características são dadas pelas métricas selecionadas para este experimento, que estão classificadas como métricas para o sistema e métricas para a classe.

O objetivo central dos experimentos realizados consiste na obtenção de conhecimentos empíricos que podem auxiliar as atividades relacionadas ao teste OO.

Além de apresentar as métricas utilizadas (conjunto de métricas formado tendo como base o estudo das métricas OO existentes), este capítulo traz as hipóteses a serem validadas empiricamente, a descrição detalhada dos experimentos com a análise dos resultados e, por fim, as considerações finais, especialmente importantes, pois resumem os resultados encontrados e dão algumas diretrizes de como se aproveitar os resultados empíricos obtidos na atividade de teste.

4.1 Métricas Utilizadas nos Experimentos

O conjunto de métricas selecionadas para o sistema são descritas nas seções subsequentes. Esse conjunto foi escolhido tendo-se como base as métricas descritas no Capítulo 2. O conjunto não engloba métricas tradicionais e adaptadas, pois o objetivo é avaliar a influência de características específicas dos projetos OO tais como classe, envio de mensagens, métodos, encapsulamento, etc.

4.1.1 Métricas para o Sistema

a) Número de classes do sistema

O número de classes que compõem o software pode servir como um indicativo da complexidade do programa, da dificuldade de manutenibilidade, teste e outras atividades.

b) Número de métodos

A quantidade de métodos que o sistema comporta pode servir como parâmetro para se avaliar a dificuldade de teste do sistema como um todo.

4.1.2 Métricas para a Classe

a) Profundidade da classe na hierarquia

Esta métrica corresponde à métrica DIT de Chidamber e Kemerer, apresentada no Capítulo 2.

b) Número de métodos da classe

Caso particular da métrica WMC, parte do conjunto de métricas de Chidamber e Kemerer, apresentada no Capítulo 2. O número de métodos da classe é simplesmente a soma ponderada dos métodos no caso em que todos os pesos valem a unidade, ou seja, o número total de métodos. Os métodos da classe podem ser públicos, protegidos ou privados. O número de métodos de uma classe se relaciona ao total de colaboração praticada entre as classes. Classes com muitos métodos podem estar efetuando muitas tarefas que deveriam ser delegadas a outras classes [5]. Classes com muitos métodos têm alta complexidade, e portanto tendem a ser difíceis de manter e testar.

c) Número de atributos

O número de atributos na classe é uma medida de tamanho da classe. As variáveis podem ser subdivididas em três categorias: públicas, privadas e protegidas. É

importante lembrar que os atributos públicos quebram o princípio de encapsulamento. O fato de que uma classe tem muitos atributos indica que a classe mantém mais relacionamentos com outras classes no sistema [5], portanto dificultando o teste.

d) Número de superclasses

O número de superclasses associados a uma classe é definido como o número de classes-pai desta classe. Apesar de na maioria dos casos existirem arranjos alternativos à herança múltipla, em alguns casos seu uso é inevitável (por exemplo, quando há uma fusão entre diferentes árvores de duas bibliotecas distintas) ou mesmo desejável. Quando suportada pela linguagem de programação, a herança múltipla resulta em dois fatores complicadores: possível colisão entre as assinaturas dos métodos herdados de diferentes árvores e maior complexidade de entendimento exigida por parte do desenvolvedor, pois tem-se uma hierarquia mais complexa. Desta maneira, esta métrica pode ser utilizada para avaliar a complexidade, mais especificamente a complexidade de teste.

e) Número de subclasses

Analogamente ao item anterior, o número de subclasses é definido como o número de subclasses imediatas para uma dada classe. Em geral, um alto número de subclasses pode indicar a necessidade de inserção de subclasses intermediárias para reduzir replicações de informação ou inconsistências entre as classes irmãs.

f) Número de mensagens recebidas e enviadas pela classe

Estas métricas podem ser obtidas por inspeção direta dos diagramas de seqüência, colaboração e da indicação de medidas tais como coesão e acoplamento.

4.2 Hipóteses a Serem Avaliadas

As métricas selecionadas na seção anterior influenciam a atividade de teste, aumentando o número de defeitos e a dificuldade de teste, que pode ser contabilizada, por exemplo, pelo número de horas gastas nessa atividade.

Assim, as hipóteses descritas a seguir servem de guia para os experimentos. A idéia é que essas hipóteses sejam avaliadas empiricamente, através dos dois experimentos

realizados, descritos em maiores detalhes nas seções subsequentes. As hipóteses a serem validadas empiricamente são listadas na Tabela 4.1. Elas têm como objetivo avaliar:

- A propensão de um sistema a apresentar defeitos;
- A dificuldade em revelar um dado defeito no programa.

Tabela 4.1: Hipóteses a serem avaliadas nos experimentos.

Hipótese	Descrição
H-1	Quanto maior o número de métodos de um sistema, maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito
H-2	Quanto maior o número de classes de um sistema, maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.
H-3	Quanto maior o número de subclasses em uma classe, maior a sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.
H-4	Quanto maior a profundidade da classe numa hierarquia de classes, maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.
H-5	Quanto maior o número de métodos de uma classe, maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.
H-6	Quanto maior o número de atributos de uma classe maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.
H-7	Quanto maior o número de mensagens recebidas (ou enviadas) por uma classe, maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.
H-8	Quanto maior o número de superclasses associadas a uma classe, maior sua probabilidade de conter defeitos, maior a dificuldade de teste, e de revelar o defeito.

4.3 Desenvolvimentos dos Sistemas para os Experimentos

Aos alunos do curso de Engenharia de Software, nível mestrado e graduação, com experiências médias em desenvolvimento de software, foram atribuídos sistemas de pequeno porte. Tentou-se conduzir o desenvolvimento dando total liberdade aos

desenvolvedores, de maneira a se obter projetos o mais próximo possível dos projetos de software comerciais orientados a objeto.

4.3.1 Os Projetos Propostos aos Alunos

Os três projetos distribuídos aos alunos possuem o mesmo grau de dificuldade, que é relativamente baixo, no que se refere às funcionalidades requisitadas, pois os sistemas representam soluções de problemas simples do cotidiano. Assim, as dificuldades de construção dos sistemas propostos se limitam às dificuldades de análise, projeto e implementação.

A seguir são listadas na Tabela 4.2 as descrições funcionais fornecidas aos alunos:

Tabela 4.2: Descrições funcionais fornecidas para os Experimentos.

A) VIDEOLOCADORA Descr. do projeto: Projetar e implementar um sistema de locação para uma locadora de vídeo. *Descrição funcional:* O software mantém um arquivo com o número de referência de cada filme, o título em português, o título original, se o filme é lançamento ou não e o número de cópias disponíveis no estoque da loja. O sistema atualiza este arquivo no momento da retirada e da entrega dos filmes locados. O sistema é capaz de registrar em arquivo as locações efetuadas, para cobrança no ato da entrega das fitas de acordo com o tempo de empréstimo e o preço dos filmes e proporcionar estatísticas para os filmes mais locados. Os lançamentos possuem preços mais elevados em relação aos não-lançamentos. Os preços também estão registrados em arquivo. Além disso, relatórios são emitidos pelo atendente na forma de resumo dos filmes locados quando o cliente empresta o filme e na forma de recibos de pagamento no ato da devolução. O relatório de movimentação da locadora é emitido quando o gerente deseja emitir um histórico das locações das fitas, deste a última emissão. Um sistema de busca por título (título completo ou não) pode informar ao cliente a disponibilidade do filme procurado na loja.
B) LIVRARIA Descr. do projeto: Projetar e implementar um sistema de consulta para uma livraria. *Descrição funcional:* O software mantém o registro em arquivo com as seguintes informações: título da obra, autor, edição, ano, editora, preço e número de exemplares disponíveis na loja. No ato da pesquisa, o cliente ou o vendedor se utiliza de um mecanismo de busca capaz de procurar uma obra por título, autor, editora e uma conjugação destas informações, completas ou não, apresentando o resultado na tela. O sistema deve ser capaz de gravar as pesquisas de modo a emitir um relatório com o histórico das pesquisas,

desde a última emissão, para utilização posterior. O sistema apresenta interface gráfica amigável e de fácil utilização, dispondo de dois modos de pesquisa: normal (por título ou ator) e avançada (pela conjugação das informações título, autor, e editora).

C) TERMINAL ELETRÔNICO DE SAQUE

Descr. do projeto: Projetar e implementar um sistema que simule a operação de um caixa de eletrônico de saque.

Descrição funcional:
O software deve simular o comportamento de um caixa eletrônico simples, que permita apenas o saque de dinheiro via cartão bancário, e o carregamento de notas via administrador de sistema. Para efetuar o saque o cliente fornece seus dados (número da conta, agência e senha). O sistema deve verificar se o valor informado para o saque é compatível com as notas existentes. Além disso, o débito no valor do saque deve ser devidamente processado pelo sistema. Um comprovante de operação deve ser emitido. Todas as situações de erro (por exemplo, saldo insuficiente e término de cédulas) devem ser previstas e devidamente tratadas.
O administrador é responsável pelo carregamento das notas do caixa. Ele se identifica via senha especial e daí pode então carregar as notas no equipamento.

4.4 Aspectos Gerais de Desenvolvimento para os Experimentos

Os sistemas foram desenvolvidos pelos alunos utilizando-se os artefatos da UML nas etapas de análise e projeto, usando a ferramenta Rational ROSE [25]. A etapa de codificação foi realizada utilizando-se a linguagem C++ [14] ou Java [24]. Toda a análise, projeto e implementação ficaram a cargo dos alunos, aos quais foram entregues apenas as descrições funcionais..

Todos os participantes têm a mesma experiência na linguagem utilizada, ferramentas e modelos empregados. Os sistemas foram implementados utilizando os seguintes modelos da UML: modelo de casos de uso, diagrama de classes e diagrama de colaboração.

4.5 Teoria Estatística Utilizada nos Experimentos

Nesta seção serão descritos detalhadamente os métodos estatísticos utilizados nos experimentos realizados neste trabalho. Optou-se por escolher métodos clássicos, quando possível simples, e bem aceitos na comunidade científica.

Se fizermos que y represente alguma característica de interesse (dificuldade de teste, por exemplo) e x represente outra variável relacionada e y (alguma métrica OO), então pode-se usar x para se fazer inferências sobre y. Isto é apenas possível, entretanto, se pudermos estar certos que tal relacionamento existe de fato e se pudermos determinar a natureza do relacionamento. Tal relacionamento entre variáveis é chamado de uma função, e pode ser representado por uma expressão $y = f(x)$.

Em tal relacionamento funcional, x é referenciada como a variável independente e y é chamada de variável dependente. Toda vez que este relacionamento funcional é da forma $y = a + bx$, é chamado de relação linear [28].

A técnica envolvida na determinação do relacionamento entre a variável dependente e a variável independente e o uso deste relacionamento para se fazer a estimação é chamada de análise de regressão [29]. A técnica relacionada, na qual se mede a força de relacionamento entre variáveis é chamada de análise de correlação [28].

Consideraremos apenas a regressão linear neste estudo, isto é, a regressão na qual o relacionamento entre variáveis pode ser expresso por uma equação linear. Assim, x é tratado como um valor conhecido de (alguma métrica) e y é usado para predizer valores de y (dificuldade de teste) para algum valor de x.

4.5.1 Correlação

Correlação é uma medida do grau de associação entre duas variáveis [28]. Toda vez que se necessita usar esta relação para se fazer inferências ou predições, é certamente lógico determinar primeiramente se existe realmente um relacionamento e, se existe, calcular alguma medida de sua força.

Um método rápido para a determinação da existência ou não de um relacionamento aparente entre duas variáveis é o diagrama de dispersão [28]. Um diagrama de dispersão é simplesmente um gráfico onde o eixo horizontal está na escala de unidades correspondendo a uma das variáveis – neste caso a variável independente – e o eixo vertical está na escala em unidades correspondendo à variável dependente. Um

exemplo de diagrama de dispersão é o da Figura 4.1, onde x pode representar, por exemplo, o número de mensagens enviadas para cada classe e y representa a porcentagem de casos de teste que não revelaram defeito (ou simplesmente a dificuldade de teste).

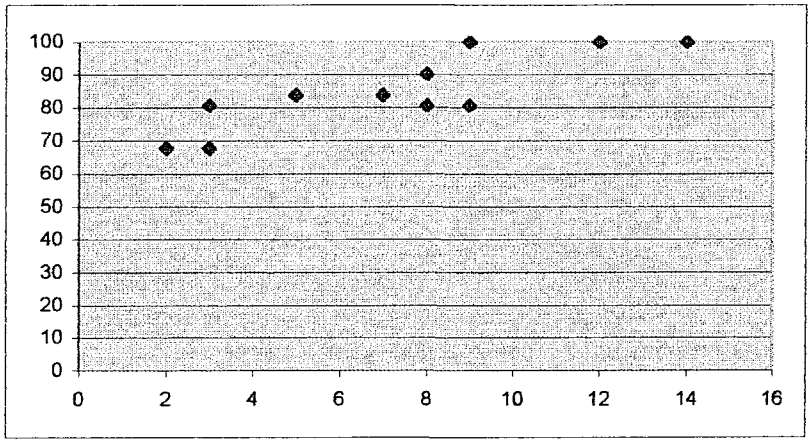


Figura 4.1: Exemplo genérico de um diagrama de dispersão.

Observando-se a disposição dos pontos nesse diagrama, é possível determinar, ao menos grosseiramente, quando existe ou não um relacionamento entre duas variáveis, e, da mesma maneira, o tipo de relacionamento.

Por exemplo, na Figura 4.1, uma relação aparente existe entre x (eixo horizontal) e y (eixo vertical). Os pontos parecem estar dispostos segundo uma linha reta, que parte do canto inferior esquerdo para o canto superior direito do gráfico. Este padrão tende a indicar a existência de uma correlação linear positiva.

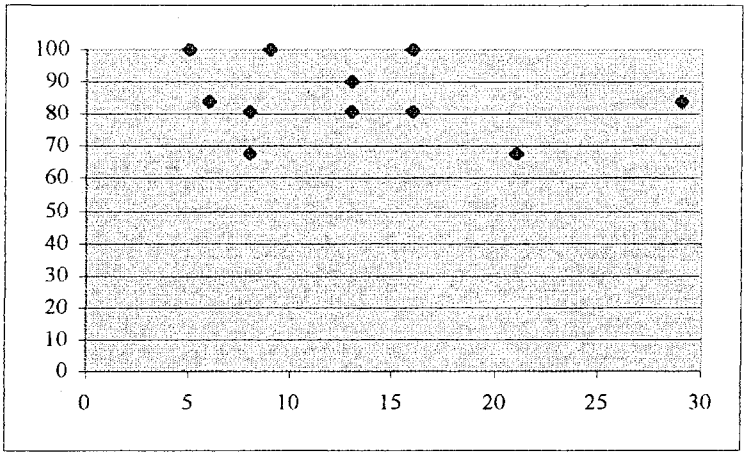


Figura 4.2: Exemplo genérico de um diagrama de dispersão com correlação negativa.

Na Figura 4.2, parece existir um relacionamento entre as variáveis x e y, que poderia representar, respectivamente, o número de atributos na classe e a porcentagem de casos de teste não revelados (ou dificuldade de teste). Entretanto, neste caso, valores altos de x parecem estar associados com valores baixos de y, e valores baixos de x com valores altos de y. Os pontos parecem também estar arranjados segundo uma linha reta, mas desta vez com origem no canto superior esquerdo do gráfico, e se dirigindo para o canto inferior direito. Este padrão tende a indicar uma correlação linear negativa. Nota-se, ainda, que os pontos neste diagrama parecem estar mais dispersos sobre uma linha reta aparente, mais dispersos que na Figura 4.1. A quantidade de dispersão indica a força no relacionamento. Pode-se concluir, observando as Figuras 4.1 e 4.2 que intuitivamente as variáveis em 4.1 têm um relacionamento mais forte que em 4.2.

O diagrama de dispersão, entretanto, não deve ser considerado conclusivo, mas considerado simplesmente como um indicador de que um relacionamento entre variáveis deve ou não ser melhor investigado. E é justamente esta abordagem utilizada nos estudos efetuados neste trabalho.

O próximo passo seguido durante os experimentos foi o cálculo do coeficiente de correlação entre as variáveis y e x, para cada análise de validade das hipóteses consideradas.

O coeficiente de correlação é uma medida do grau de associação linear entre duas populações representadas por x e y, é designado pela letra grega ρ, e indica o grau de associação entre a população de x e a população de y [29]. Em termos práticos, nos experimentos não se tem acesso à população inteira, e o coeficiente de correlação da

população precisa ser estimado a partir da amostra disponível. Assim, o coeficiente de correlação da amostra é definido como.

$$r_{y,x} = \frac{\sum (x - \overline{x})(y - \overline{y})}{\sqrt{\sum (x - \overline{x})^2 \sum (y - \overline{y})^2}}$$

onde y e x representam as suas variáveis envolvidas. A média amostral de x e y á dada por $\overline{x}$ e $\overline{y}$.

Se uma distinção é feita entre a variável dependente e a independente, a variável independente aparece em primeiro lugar em $r_{y,x}$.

O coeficiente de correlação é sempre um número do intervalo [-1, +1]. Um valor de r = +1 representa uma correlação positiva perfeita.; r = -1 representa uma correlação negativa perfeita; r = 0 indica falta completa de correlação.

4.5.2 Inferência sobre o Coeficiente de Correlação

O coeficiente de correlação r é o coeficiente de correlação da amostra, e é usado para se fazer inferências sobre ρ, o coeficiente de correlação da população, que é desconhecido.

A inferência mais comum sobre ρ é um teste de hipótese [29] de que ρ é igual a zero, ou seja, que não existe relacionamento (correlação nula) entre as duas variáveis. A hipótese nula e a hipótese alternativa (não nula) a ser testada pode ser escrita como:

$$H_0 : \rho = 0$$
$$H_1 : \rho \neq 0$$

A estatística usada para este teste é r, o coeficiente de correlação, calculado a partir de uma amostra de tamanho n.

Para se usar qualquer estatística para o teste de hipótese, a distribuição amostral desta estatística precisa ser conhecida. Neste caso, r não segue nenhuma das distribuições comumente usadas e tabuladas. Porém, uma transformação muito simples resulta na estatística de distribuição t, com n-2 graus de liberdade:

$$t = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}}$$

A hipótese sobre ρ pode ser testada com a estatística t calculada a partir da amostra e comparada com o valor crítico t para o nível de significância especificado. Os valores que t assume para os graus de liberdade estão tabulados no Apêndice B [29].

Para que o teste de hipótese resulte em na confirmação da hipótese alternativa (H-1 : $\rho \neq 0$), ou seja existe uma correlação não nula entre as variáveis, o valor de t calculado deve ser inferior ao valor de t tabulado para este teste (Apêndice B).

4.6 O Experimento 1

4.6.1 Objetivo

O objetivo deste experimento consiste em avaliar empiricamente as hipóteses descritas na Tabela 4.1. Mais especificamente, quer-se avaliar quantitativamente a influência do conjunto de métricas da Seção 4.1 na atividade de teste. Essa influência se manifesta devido a uma possível correlação entre cada uma das métricas e a dificuldade de teste, neste experimento caracterizada pela dificuldade de se revelar defeitos em um sistema OO.

O Experimento 1 consiste na inserção de defeitos no código fonte de programas considerados corretos –com respeito às suas especificações – e na coleta das métricas para cada sistema, preferencialmente através dos diagramas UML, ou alternativamente pela inspeção do código-fonte final, para complementação e/ou verificação. Os defeitos introduzidos foram derivados tendo-se como base a técnica de teste baseada em erros e o critério *Análise de Mutantes*. Foram utilizados os operadores de mutação da ferramenta

Proteum [7], aqueles considerados como mais representativos de cada classe de operadores, relacionados na listagem abaixo. A relação completa dos operadores da ferramenta pode ser consultada no Apêndice A.

- Supressão de um comando (SSDL);
- Inserção de negação lógica (OCNG);
- Troca de operador aritmético por outro operador aritmético (OAAN)
- Troca de uma constante por outra constante (Cccr);
- Troca do comando *while* por *do-while* (SWDD);
- Troca de variáveis (vários operadores conjugados).

Após a geração e aplicação dos casos de teste para todos os sistemas, os resultados obtidos no teste foram analisados estatisticamente, utilizando-se as métricas previamente coletadas, de modo a se obter os resultados deste estudo empírico.

4.6.2 As Etapas Seguidas na Realização do Experimento 1

O estudo foi conduzido seguindo os passos abaixo:

1. Seleção dos sistemas: 9 sistemas foram construídos, mas 4 sistemas não possuíam uma documentação completa, ou apresentaram problemas na implementação. Logo, não foram considerados e apenas 5 foram usados no estudo.

2. Coleta das métricas: Para cada um dos 5 sistemas, as seguintes métricas foram coletadas: número de métodos no sistema, número de atributos no sistema, número de classes e total de mensagens recebidas e enviadas no sistema. Vide Tabela 4.3. Essas métricas estão relacionadas, respectivamente, às hipóteses H-1, H-6, H-2 e H-7. As demais métricas, relacionadas às hipóteses H-3, H-4, H-5 e H-8, não foram coletadas, e a justificativa para esse fato é discutida na Seção 4.9.

3. Inclusão de defeitos no código: para cada sistema um conjunto de defeitos aleatoriamente gerados foi inserido no código fonte e aproximadamente 10 versões "incorretas" foram geradas para cada um dos 5 projetos. Cada versão incorreta possui apenas um defeito inserido. Os defeitos foram derivados baseando-se nos operadores da ferramenta Proteum [7], e estão classificados na Tabela 4.4. Além da classificação dos defeitos, a Tabela 4.4 apresenta o número de versões geradas com

cada categoria de operadores de mutação e o número de defeitos não revelados por nenhum caso de teste. Os defeitos introduzidos e a suas respectivas posições no código (em que classe e em que método) podem ser consultados em [26].

4. Geração dos casos de teste para os sistemas: Os testadores dos sistemas também foram os estudantes, mas não os desenvolvedores. Eles geraram os casos de teste usando apenas a descrição funcional e os modelos de casos de uso. Não foram aplicados critérios de teste e conhecimento prévio sobre o código fonte. Três conjuntos de casos de teste foram gerados, um para cada descrição. O número de casos de teste de cada conjunto está na Tabela 4.5. Os casos de teste podem ser consultados em detalhes em [26].

5. Execução das versões incorretas dos sistemas usando os conjuntos de teste: Observando-se as saídas obtidas, foram derivados dois resultados para cada sistema, apresentados na Tabela 4.5. O primeiro é o número (ou porcentagem) de defeitos não revelados, ou seja, o número de versões incorretas que não produziram nenhuma saída incorreta para todos os casos de teste. A segunda mostra a dificuldade de se revelar um defeito, dado pelo número (ou porcentagem) de execuções não efetivas, ou seja, a execução para cada versão incorreta que não revelou um defeito. A porcentagem média para cada sistema é apresentada na última coluna da Tabela 4.5. Por exemplo, para o Sistema 1, 10 versões incorretas foram geradas, ou seja, 10 defeitos foram inseridos, e 1 defeito (10%) não foi revelado pelo conjunto de casos de teste composto por 12 casos de testes. Usando as versões incorretas e todos os casos de teste, o Sistema 1 teve um número de 120 execuções e 79,17% delas não revelaram nenhum defeito.

6. Coleta de algumas métricas para as classes incorretas, ou seja, classes em que defeitos foram introduzidos. A Tabela 4.6 apresenta as métricas coletadas para as classes incorretas do Sistema 2: número de métodos, número de atributos e número de mensagens enviadas e recebidas. O número de defeitos inseridos e revelados também é apresentado. Por exemplo, a classe 1 possui 13 métodos, 35 atributos, 12 mensagens recebidas e 8 mensagens enviadas. Dois defeitos foram inseridos nesta classe, todos eles revelados e a porcentagem média de execuções efetivas é também apresentada (85.48%). Nota-se que o Sistema #2 possui 16 classes mas apenas as classes nas quais defeitos foram inseridos são apresentadas na Tabela 4.6. Neste trabalho foi apresentado apenas o Sistema #2 para efeito da análise das métricas coletadas para a classe, por motivo de simplicidade, já que para entre os demais

sistemas estudados os resultados foram basicamente os mesmos. Detalhes em relação aos resultados para os demais sistemas podem ser obtidos em [26].

Tabela 4.3: Métricas coletadas para os sistemas.

	Métricas			
Sistema	Número de métodos	Número de atributos	Número de classes	Número de mensagens Enviadas / Recebidas
#1	40	91	7	49
#2	223	168	16	102
#3	70	40	7	53
#4	17	104	7	48
#5	18	239	17	123

Tabela 4.4: Tipo de defeitos inseridos, versões e defeitos não revelados.

Tipo do defeito	Número de versões	Número de defeitos não revelados
#1: Supressão de um comando (SSDL)	8	1 (12,50%)
#2: Inserção de negação lógica (OCNG)	8	0 (0.00%)
#3: Troca de operador aritmético por outro operador aritmético (OAAN)	17	1 (5.88%)
#4: Troca de uma constante por outra constante (Cccr)	12	2 (16.67%)
#5: Troca do comando while por do-while (SWDD)	3	1 (33.33%)
#6: Troca de variáveis (vários operadores conjugados)	3	0 (0.00%)

Tabela 4.5: Número de casos de testes x defeitos não revelados.

Sistema	Número de casos de teste	Número de versões incorretas	Número (ou %) de defeitos não revelados	Média de execuções não efetivas
#1	12	10	1 (10%)	79.17%
#2	31	11	3 (27,27%)	85.04%
#3	24	10	0	90.00%
#4	12	10	1 (10%)	76.67%
#5	31	10	0	77.42%

Tabela 4.6: Métricas e defeitos não revelados nas classes do Sistema #2.

Classe	Numero de métodos	Número de atributos	Mensagens recebidas	Mensagens enviadas	Número de defeitos inseridos	Número (e %) de defeitos não revelados)	Média em % de execuções não efetivas
# 1	13	35	12	8	2	0	85.48
# 2	8	23	3	3	2	0	74,20
# 3	6	11	5	7	1	0	83.87
# 4	16	42	9	9	2	1 (50%)	90.32
# 5	5	15	14	14	1	1 (100%)	100
# 6	9	0	12	12	1	1 (100%)	100
# 7	21	4	1	2	1	0	67.74
# 8	29	17	4	5	1	0	83.87

4.6.3 Análise dos Resultados

Para se verificar a influência das métricas coletadas na dificuldade de teste, as hipóteses definidas na Seção 4.2 foram usadas.

4.6.3.1 Resultados para o Sistema

As métricas coletadas na Tabela 4.3 foram usadas e as Tabelas 4.7 e 4.8 apresentam os indicativos estatísticos de correlação. Mostra-se claramente, analisando-se o parâmetro R^2 que a correlação é fraca. A correlação obtida utilizando-se o número de métodos (hipótese H-1) é mais forte em ambos os casos, usando o número de defeitos não revelados e a porcentagem média de execuções não efetivas. A análise estatística indica que a métrica de número total de métodos no sistema, que reflete a complexidade ou tamanho de um sistema, pode influenciar na dificuldade de teste. Este resultado dá um suporte empírico à noção intuitiva de que quanto mais complexo for um sistema OO, ou seja, quanto mais métodos ele possuir, maior a dificuldade de teste deste sistema. Essa dificuldade de teste se evidenciou na dificuldade de se revelar defeitos no código através dos casos de teste. Assim, aumenta-se a dificuldade de revelar defeitos em um sistema à medida em que o número de métodos aumenta.

Através dos experimentos obteve-se um indicativo de que a hipótese H-1 é verdadeira. Os coeficientes de correlação baixos em módulos para as demais hipóteses

indicam que as mesmas não puderam ser validadas, porém não foram contrariadas, e estudos empíricos futuros podem ser derivados para se tentar validá-las.

Tabela 4.7: Resultados da análise estatística de regressão usando o número de defeitos não revelados.

Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	0.825	68.07	H_0 rejeitada
Número de atributos	0.135	1.82	H_0 rejeitada
Número de classes	0.283	8.01	H_0 rejeitada
Número de mensagens recebidas / enviadas	0.132	1.74	H_0 rejeitada

Tabela 4.8: Resultados da análise estatística de regressão usando a porcentagem de execuções não efetivas.

Métrica	Coeficiente de Correlação	R^2 (%)	Teste t para α=0.01
Número de métodos	0.547	29.96	H_0 rejeitada
Número de atributos	-0.489	23.95	H_0 rejeitada
Número de classes	-0.101	1.01	H_0 rejeitada
Número de mensagens recebidas / enviadas	-0.120	1.45	H_0 rejeitada

4.6.3.2 Resultados para a Classe

A mesma análise foi conduzida para validar as hipóteses com relação às classes do Sistema #2. Os dados da Tabela 4.6 foram usados e as Tabelas 4.9 e 4.10 mostram os resultados estatísticos obtidos. Estas tabelas mostram uma correlação significativa entre as métricas número de mensagens enviadas na classe e número de mensagens recebidas na classe com o número de defeitos não revelados e a porcentagem dos casos de teste não efetivos. Entretanto não se pode estabelecer uma correlação significativa para as outras métricas. Assim, apenas a hipótese H-7 pode se validada. Devido aos valores

baixos em módulo dos coeficientes de correlação das Tabelas 4.9 e 4.10, as demais hipóteses não estão descartadas, apenas não puderam ser validadas. Este fato abre frente a novos experimentos empíricos.

Tabela 4.9: Resultados da análise estatística de regressão usando o número de defeitos não revelados.

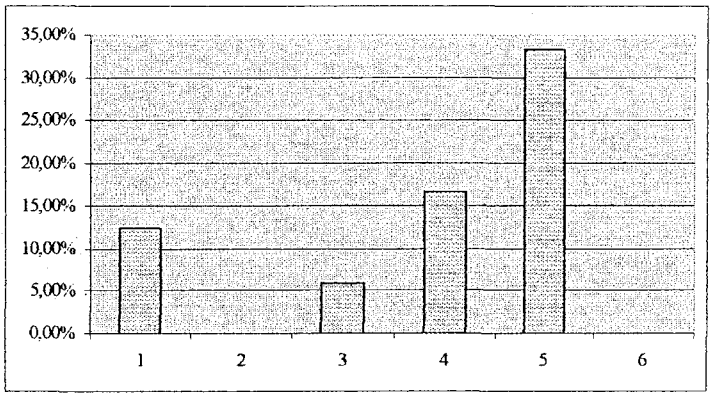
Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	-0.337	11.35	H_0 rejeitada
Número de atributos	0.036	0.13	H_0 rejeitada
Número mensagens enviadas	0.826	68.13	H_0 rejeitada
Número de mensagens recebidas	0.708	50.20	H_0 rejeitada

Tabela 4.10: Resultados da análise estatística de regressão usando a porcentagem de execuções não efetivas.

Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	-0.2247	5.05	H_0 rejeitada
Número de atributos	0.0571	0.33	H_0 rejeitada
Número mensagens enviadas	0.8521	72.62	H_0 rejeitada
Número de mensagens recebidas	0.5927	35.14	H_0 rejeitada

4.6.3.3 Tipos de Defeitos e Dificuldades em Revelá-los

Os defeitos inseridos foram classificados em 6 categorias e a porcentagem de defeitos não revelados para cada categoria é mostrada no gráfico da Figura 4.3. Os defeitos classificados na categoria 5 (troca do comando while por do-while, SWDD) foram os mais difíceis de se revelar e foi mais fácil revelar defeitos inseridos em operadores de *negação da condição* e *troca de variáveis*. Esta informação pode ser usada para estender a técnica de teste baseada em defeitos para sistemas OO. Casos de teste para matar os mutantes gerados considerando estes tipos de defeitos devem ser gerados e executados, podendo assim aumentar a eficácia dos testes.



1: Supressão de um comando (SSDL)
2: Inserção de negação lógica (OCNG)
3: Troca de operador aritmético por outro operador aritmético (OAAN)
4: Troca de uma constante por outra constante (Cccr)
5: Troca do comando while por do-while (SWDD)
6: Troca de variáveis (vários operadores conjugados)

Figura 4.3: Tipo de Defeitos e Dificuldade em Revelá-los.

4.7 O Experimento 2

4.7.1 Objetivo

O objetivo geral deste experimento é o mesmo do anterior: a validação empírica das hipóteses da Tabela 4.1. Assim, quer-se avaliar o conjunto de métricas na atividade de teste.

No entanto, o Experimento 2 também tem como objetivo relacionar as métricas OO com a propensão do sistema em conter defeitos. Durante o desenvolvimento do sistema pelos alunos, a atividade de teste foi realizada pelos próprios desenvolvedores que produziram um relatório de teste reportando os defeitos reais encontrados. Com base nesses relatórios, o Experimento 2 foi realizado.

4.7.2 As Etapas Seguidas na Realização do Experimento 2

O estudo foi conduzido seguindo os passos abaixo:

1. Seleção dos sistemas: o conjunto de projetos utilizados nesse experimento foi diferente. Ainda assim, três sistemas analisados foram comuns aos dois experimentos, e três novos sistemas foram considerados. Ao total, para o Experimento 2, foram utilizados 6 sistemas para a coleta das métricas a análise dos resultados.

2. Coleta das métricas dos sistemas: Para cada um dos 6 sistemas, as seguintes métricas de sistema foram coletadas: número de métodos no sistema, número de atributos no sistema, número de classes e total de mensagens recebidas e enviadas no sistema. As métricas estão escritas na Tabela 4.11. As razões para coleta apenas dessas duas métricas (e não de todas as métricas relacionadas com as hipóteses) se deve ao mesmo motivo do Experimento 1, explicados na Seção 4.9.

3. Coleta dos relatórios de teste para os sistemas: Durante a atividade de teste realizada pelos alunos no desenvolvimento dos sistemas, relatórios de teste foram gerados. Esses relatórios contêm: relação dos casos de teste executados, tempo de execução e correção dos defeitos revelados, se o caso de teste revelou o defeito, o tipo de defeito revelado, e a localização do defeito (em que classe e em que método). Para os sistemas foram coletados os dados das Tabelas 4.12 e 4.13. O relatório de teste detalhado para o Sistema #1 é apresentado na Tabela 4.15. A Tabela 4.16 mostra de forma resumida os tipos de defeitos encontrados no Sistema #1. Elas são, portanto, parte integrante das informações coletadas nas Tabelas 4.12 e 4.13.

4. Coleta das métricas para as classes: Para que se pudessem analisar as métricas para a classe, cada projeto teve suas métricas para a classe devidamente coletadas. Os resultados para a classe foram basicamente os mesmos para os 6 projetos, quando analisados de forma individual. Assim, por simplicidade, é discutido aqui apenas o Sistema #1 cujas métricas estão escritas na Tabela 4.14; os dados coletados para os demais projetos podem ser consultados em [26].

5. Repetição dos passos 3 e 4 para cada projeto: Participaram do Experimento 2 um total de 6 projetos, e para cada um deles as métricas das classes e os relatórios de teste gerados pelos alunos foram analisados.

Tabela 4.11: Métricas Coletadas para os Sistemas.

	Métricas			
Sistema	Número de métodos	Número de classes	Número de atributos	Número de mensagens Enviadas / Recebidas
#1	223	16	168	102
#2	70	7	40	53
#3	178	17	239	123
#4	120	12	164	103
#5	320	25	227	159
#6	81	9	52	65

Tabela 4.12: Dados dos Relatórios de Teste para cada Sistema.

	Dados dos Relatórios de Testes	
Sistema	Número de defeitos revelados	Tempo gasto no teste e/ou correção (min)
#1	22	442
#2	8	175
#3	14	341
#4	11	295
#5	28	580
#6	10	203

Tabela 4.13: Análise dos Defeitos Encontrados para os Sistemas.

Defeitos encontrados	Número total de defeitos encontrados
Tipo 1 : Comparação	27
Tipo 2 : Supressão da declaração	4
Tipo 3 : Negação da condição	7
Tipo 4 : Troca de constante por constante	4
Tipo 5 : Erro nos loops *for*, *do* e *do-while*	7
Tipo 6 : Troca de operador	7
Tipo 7 : Troca de variáveis	7
Tipo 8 : Atribuição de variáveis	30

Tabela 4.14: Métricas Coletadas para o Sistema #1.

Classe	N. de Métodos	N. de Atributos	Mensagens Rec.	Mensagens Env.
TFManutençãoObra	13	35	12	8
TFVenda	8	23	3	3
TFPesquisa	6	11	5	7
TDMObra	16	42	9	9
TDMVenda	5	15	14	14
Loja	16	2	8	9
Livro	9	0	12	12
Venda	21	4	1	2
Autor	12	1	2	5
Editora	11	1	3	3
Item_Venda	17	4	2	4
Obra	29	17	4	5
Pagamento	12	2	4	1
PDV	15	1	9	6
Pesquisa	20	6	2	2
Periodico	13	4	12	12

Tabela 4.15: Relatório de teste gerado pelos alunos para o Sistema #1.

	t (min)	Revelou defeito	Tipo do defeito	Na classe	No método
C.T. 1	13	Não			
C.T. 2	6	Sim	1 – Comparação	TFVenda	finaliza()
C.T. 3	21	Sim	3 - Negação da condição	Obra	consultanormal()
C.T. 4	9	Sim	7 - Troca de variáveis	TFManutenção Obra	excluiobra()
C.T. 5	15	Sim	2 - Supressão da declaração	Venda	efetuapgto()
C.T. 6	7	Não			
C.T. 7	7	Sim	6 - Troca de operador	Livro	Livro()
C.T. 8	8	Sim	5 - Troca do comando while por do-while	Editora	inclui()
C.T. 9	4	Sim	4 - Troca de constante por constante	TFPesquisa	create()
C.T. 10	15	Sim	8 - Atribuicao de variaveis	Venda	finalizavenda()
C.T. 11	9	Sim	1 – Comparação	Autor	localiza()
C.T. 12	7	Não			
C.T. 13	12	Sim	8 - Atribuição de variáveis	TDMObra	tipoobra()
C.T. 14	12	Sim	8 - Atribuição de variáveis	TDMObra	tipoobra()
C.T. 15	4	Sim	8 - Atribuição de variáveis	TFPesquisa	calculaindice()
C.T. 16	12	Não			
C.T. 17	21	Sim	4 - Troca de constante por constante	Obra	consultanormal()
C.T. 18	12	Sim	8 - Atribuição de variáveis	Item_Venda	subtotal()
C.T. 19	12	Sim	8 - Atribuição de variáveis	Item_Venda	subtotal()
C.T. 20	9	Sim	7 - Troca de variáveis	Obra	baixaestq()
C.T. 21	8	Não			
C.T. 22	12	Sim	1 - Comparação	Loja	incluiobra()
C.T. 23	9	Sim	1 - Comparação	Pagamento	create()
C.T. 24	13	Não			
C.T. 25	10	Não			
C.T. 26	15	Sim	3 - Negação da condição	Pesquisa	Pesquisa()
C.T. 27	15	Sim	7 - Troca de variáveis	Pesquisa	Pesquisa()
C.T. 28	12	Não			
C.T. 29	11	Sim	4 - Troca de constante por constante	PDV	finalizavenda()
C.T. 30	12	Sim	6 - Troca de operador	Loja	consultaavancada()

Obs.: t é o tempo de execução dos casos de teste e da correção de um eventual defeito encontrado.

Tabela 4.16: Defeitos Encontrados no Sistema #1.

Tipo dos defeitos encontrados	Número de defeitos encontrados
1 – Comparação	4
2 - Supressão da declaração	1
3 - Negação da condição	2
4 - Troca de constante por constante	3
5 - Erro nos loops for, do e do-while	1
6 - Troca de operador	2
7 - Troca de variáveis	3
8 - Atribuição de variáveis	6

4.7.3 Análise dos Resultados

Para se verificar a influência das métricas coletadas na dificuldade de teste, da mesma forma que o Experimento 1, as hipóteses definidas na Seção 4.2 foram usadas em conjunto com as métricas e relatórios de teste coletados.

4.7.3.1 Resultados para o Sistema

As métricas coletadas na Tabela 4.1 foram usadas e as Tabelas 4.17 e 4.18 apresentam os indicativos estatísticos de correlação. Analisando-se o parâmetro R^2, pode-se verificar forte correlação entre as métricas número de métodos do sistema, número de classes do sistema e número de mensagens recebidas e enviadas do sistema versus o número de defeitos revelados (Tabela 4.17) e versus o tempo gasto no teste e correção (Tabela 4.18). Uma correlação média obteve-se para o número de atributos do sistema versus o número de defeitos revelados. Assim, pode-se obter desse experimento um forte indicativo da validade das hipóteses H-1, H-2 e H-7, da influência do número de métodos, de classes e de mensagens enviadas e recebidas em um sistema; de que quando maior for o valor dessas métricas, maior é a probabilidade do sistema de conter defeitos.

Tabela 4.17: Resultados da análise estatística de regressão usando o número de defeitos revelados.

Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	0.9799	96.03	H_0 rejeitada
Número de atributos	0.6748	45.54	H_0 rejeitada
Número de classes	0.9218	84.98	H_0 rejeitada
Número de mensagens recebidas / enviadas	0.8240	67.91	H_0 rejeitada

Tabela 4.18: Resultados da análise estatística de regressão usando o tempo gasto no teste e/ou correção.

Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	0.9938	98.77	H_0 rejeitada
Número de atributos	0.7938	63.02	H_0 rejeitada
Número de classes	0.9650	93.12	H_0 rejeitada
Número de mensagens recebidas / enviadas	0.9116	83.10	H_0 rejeitada

A análise da Figura 4.4 deixa claro que os defeitos predominantes para os 6 sistemas em conjunto foram os tipos de defeitos 1 (de comparação) e 8 (de atribuição de variáveis). Isto representa um resultado importante, na medida que é um indicativo dos tipos de defeitos mais comuns cometidos por um desenvolvedor de experiência mediana.

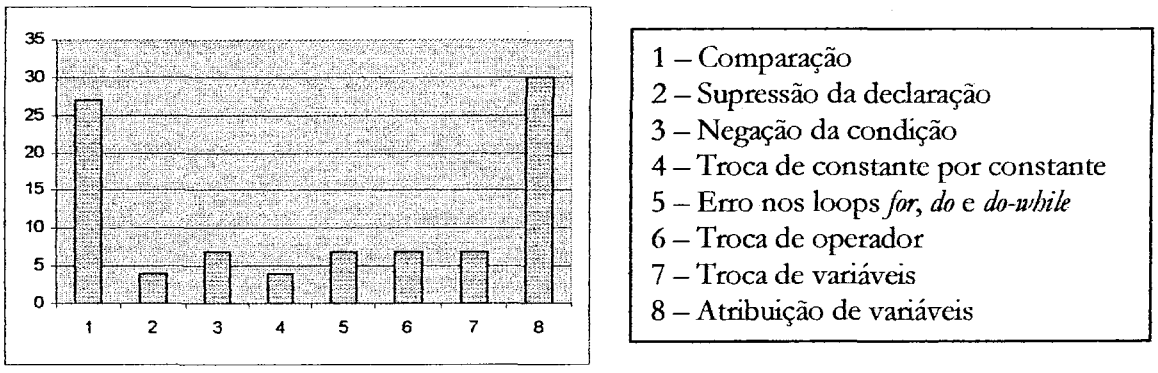


Figura 4.4: Análise dos Defeitos Encontrados para os Sistemas.

4.7.3.2 Resultados para a Classe

Para o estudo das hipóteses relacionadas às classes, foram analisados 6 projetos. Como obteu-se o mesmo resultado para todos os projetos, a título de análise, por motivo de clareza, está aqui analisado apenas o Sistema #1.

Analisando-se o parâmetro R^2 (coeficiente de correlação ao quadrado) da Tabela 4.19, pode-se perceber que a métrica número de métodos na classe versus o número de defeitos revelados é forte e versus o tempo gasto nos testes da classe é média (considerando o contexto de correlações). Portanto, a H-1 pôde ser confirmada para todos os 6 projetos.

As demais hipóteses não puderam ser validadas para a classe, porém, devido aos baixos módulos dos coeficientes de correlação, experimentos futuros devem ser realizados, de modo a se tentar validar tais hipóteses.

Tabela 4.19: Resultados da análise estatística de regressão usando o número de defeitos revelados.

Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	0.9070	82.27	H_0 rejeitada
Número de atributos	0.1009	1.02	H_0 rejeitada
Número de mensagens enviadas	-0.3785	14.33	H_0 rejeitada
Número de mensagens recebidas	-0.4679	21.90	H_0 rejeitada

Tabela 4.20: Resultados da análise estatística de regressão usando o tempo gasto nos testes da classe.

Métrica	Coeficiente de Correlação linear	R^2 (%)	Teste t para α=0.01
Número de métodos	0.5000	25.00	H_0 rejeitada
Número de atributos	-0.085	0.72	H_0 rejeitada
Número de mensagens enviadas	-0.2496	6.23	H_0 rejeitada
Número de mensagens recebidas	-0.2972	8.83	H_0 rejeitada

4.7.3.3 Tipos de Defeitos Encontrados no Sistema #1

A análise da Figura 4.5 deixa claro que os defeitos predominantes para o sistema #1, analisado individualmente, foram os defeitos do tipo 1 (de comparação) e 8 (de atribuição de variáveis). O mesmo resultado foi obtido para os demais projetos e estão resumidos na Seção 4.7.3.1

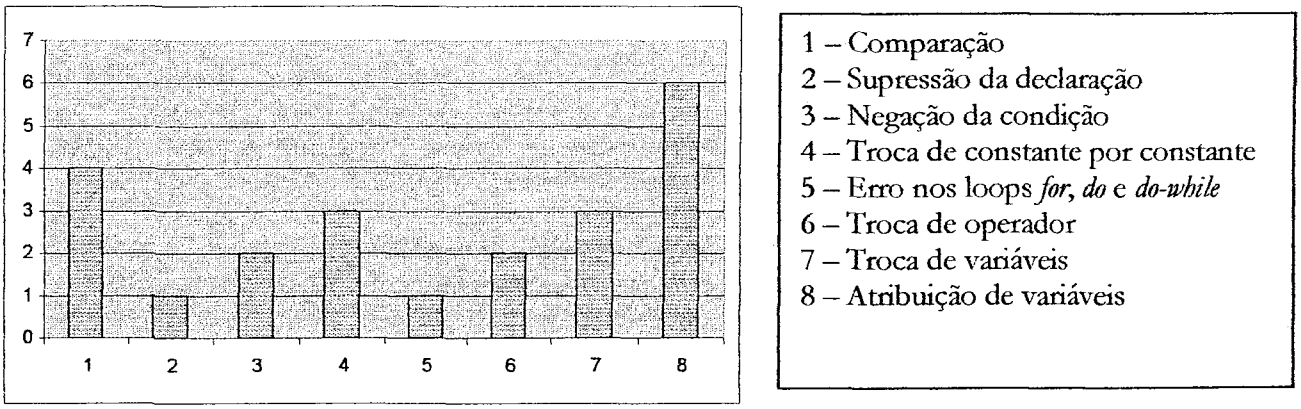


Figura 4.5: Defeitos Encontrados no Sistema #1.

4.9 Considerações Finais

Pode-se observar a partir dos experimentos que as métricas orientadas a objeto podem retratar fielmente, além do tamanho e complexidade dos sistemas, problemas no desenvolvimento. Nos sistemas analisados durante os experimentos, pode-se perceber uma notória subutilização do mecanismo de herança nos sistemas desenvolvidos pelos alunos. Os alunos e desenvolvedores dos sistemas fizeram pouco uso da herança nas etapas de projeto e implementação. Os valores excessivamente baixos (predominantemente zero, seguido dos valores 1 e 2) obtidos na métrica *profundidade da classe numa hierarquia* confirmam esta tendência. Uma possível explicação para tal fato seria inexperiência no projeto e implementação OO por parte dos alunos, os quais muitos estavam acostumados apenas ao desenvolvimento estrutural tradicional; outra explicação seria a simplicidade dos projetos propostos (videolocadora, livraria e terminal de saque).

Esses valores baixos obtidos para a métrica profundidade da classe numa hierarquia implicariam, em um ambiente industrial, a reestruturação do modelo da UML diagrama de classes. Nos estudos empíricos conduzidos durante os experimentos, as

hipóteses H-3, H-4 e H-8 não puderam ser avaliadas, por motivos estatísticos, já que os valores obtidos foram na maioria iguais a zero, ou alternativamente 1 ou 2. Por esta razão, nenhuma discussão foi desenvolvida em relação a estas hipóteses.

Com respeito ao Experimento 1, pôde-se obter um indicativo de que a hipótese H-1 é verdadeira para os 5 sistemas analisados ao mesmo tempo. Assim, quanto maior o número de métodos de um sistema, maior a dificuldade de teste e de se revelar o defeito. Quando a análise estatística é conduzida para os sistemas individualmente, a hipótese H-7 pôde ser validada, de modo que quanto maior o número de mensagens recebidas e enviadas por uma classe maior é a sua dificuldade de teste e de se revelar o defeito.

O Experimento 2 apontou para a validade das hipóteses H-1, H-2 e H-7 para os 6 sistemas analisados em conjunto. Dessa maneira, quanto maior o número de métodos de um sistema, de classes de um sistema e mensagens enviadas e recebidas, maior a probabilidade de conter defeitos. A hipótese H-1 pôde ser validada para os projetos analisados individualmente. Assim, quanto maior o número de métodos de um sistema, maior a sua probabilidade de conter defeitos.

Com exceção das métricas relacionadas à hierarquia de classe, que não puderam ser analisados por motivo relacionado acima, os resultados obtidos coincidiram com aqueles obtidos por Briand et al [42], experimento descrito na Seção 3.6. Este resultado é importante, na medida que reforça a validade dos resultados aqui obtidos. Assim, através de uma metodologia estatística mais simples que aquela conduzida no estudo de Basili [17], chegou-se ao mesmo resultado.

Capítulo 5

Conclusões e Trabalhos Futuros

Métricas para o desenvolvimento OO representam um campo de estudo relativamente novo, não possuindo ainda maturidade. As métricas auxiliam a obtenção da qualidade do projeto que está sendo desenvolvido.

Estudos teóricos e empíricos [3, 4, 5, 9, 11] mostram que, em geral, as métricas de software têm ajudado na detecção de problemas durante o ciclo de desenvolvimento, proporcionando suporte a tomadas de decisão baseadas em dados objetivos, números, e não mais puramente na experiência acumulada pelos desenvolvedores. Assim, os estudos sobre métricas contribuem, em geral, para uma atitude cada vez mais científica em relação ao desenvolvimento de software.

Nesse contexto, ressalta-se a importância do trabalho apresentado cuja contribuição consiste num melhor entendimento de quais métricas estão melhor correlacionadas às atividades de teste, de modo a se entender quais características do sistema, ou métricas, devem servir de guia nas etapas de projeto e implementação, para se obter um software mais fácil de se testar e manter.

Como resultado desse trabalho, as seguintes hipóteses puderam ser validadas:

- **Experimento 1**: obteve-se um indicativo de que a hipótese H-1 (quanto maior o número de métodos de um sistema, maior a dificuldade de teste e de se revelar o defeito) é verdadeira para os 5 sistemas analisados ao mesmo tempo. Para os sistemas analisados individualmente, a hipótese H-7 (quanto maior o

número de mensagens recebidas e enviadas por uma classe maior é a sua dificuldade de teste e de se revelar o defeito) foi validada;

- **Experimento 2**: apontou para a validade das hipóteses H-1 (quanto maior o número de métodos de um sistema, maior a dificuldade de teste e de se revelar o defeito), H-2 (quanto maior o número de classes de um sistema, maior a dificuldade de teste e de se revelar o defeito) e H-7 (quanto maior o número de mensagens enviadas e recebidas por uma classe, maior a dificuldade de teste e de se revelar o defeito) para os 6 sistemas analisados em conjunto. A hipótese H-1 pôde ser validada para os projetos analisados individualmente.

Outro resultado importante obtido refere-se às hipóteses que não puderam ser validadas (H-3, H-4, H-5, H-6 e H-8). Como os coeficientes de correlação destas hipóteses assumem valores baixos em módulo nas tabelas dos resultados do Capítulo 4, nenhuma relação pôde ser obtida para essas hipóteses, porém as mesmas não foram negadas. Isso ocorreria se os coeficientes de correlação assumissem valores negativos próximos de -1. Assim, esse resultado reforça a necessidade de novos experimentos com o intuito de validar estas hipóteses.

Uma contribuição do trabalho é um melhor entendimento das relações entre as métricas dos sistemas OO e a qualidade do software desenvolvido. A atividade de teste em geral consome muito esforço e tempo, e os resultados aqui apresentados dão subsídios empíricos que podem ajudar a facilitar a atividade de teste. Os resultados obtidos mostram forte indicativo de que o número de mensagens enviadas e recebidas por uma classe ou por um sistema influencia a dificuldade de teste e a propensão do sistema em conter defeitos. Essa métrica está fortemente ligada aos conceitos de coesão e acoplamento. Portanto, uma diretriz importante que é reforçada com o resultado desse trabalho é a busca por alta coesão e baixo acoplamento. Classes com papéis específicos e bem definidos e diagramas de interação com reduzido número de mensagens facilitam o teste de software

Os defeitos que mais ocorreram no Experimento 2 foram os de comparação e de atribuição de variáveis. Este resultado é importante, pois é um indicativo dos tipos de defeitos mais comuns cometidos por um desenvolvedor de experiência mediana. Esses resultados podem ser usados para estender a técnica de teste baseada em defeitos para sistemas OO. Assim, casos de teste para matar mutantes que consideram estes tipos de defeitos devem ser utilizados.

Os resultados empíricos obtidos através dos experimentos realizados contribuem no sentido de proporcionar uma melhor utilização das métricas no desenvolvimento OO, através de diretrizes para a redução da complexidade, permitindo um desenvolvimento melhor controlado, já que uma correlação mais clara entre as métricas e as características reais do software podem ser inferidas. Também podem contribuir na qualidade do desenvolvimento OO, já que os dados empíricos quando bem utilizados resultam em projetos de software com métricas tais que os sistemas sejam mais fáceis de testar e com menos defeitos.

5.1 Trabalhos Futuros

O presente trabalho abre espaço a muitas linhas de pesquisa futura, no sentido de se complementar este trabalho ou se partir para pesquisas semelhantes. Alguns pontos são destacados abaixo:

- Repetir os procedimentos do Experimento 1 e 2 para sistemas mais complexos (maior número de classes com uma estrutura hierárquica mais complexa, maior número de métodos e atributos), com geração e aplicação de mais casos de teste, para avaliar as hipóteses H-3, H-4, H-5, H-6 e H-8;

- Outros conjuntos de hipóteses podem ser empregadas nos estudos empíricos, de modo a se aumentar os conhecimentos empíricos de métricas aplicadas na atividade de teste. A inter-relação entre esses conjuntos de hipóteses, e a identificação das características comuns captadas pelas métricas aplicadas às hipóteses podem ser investigadas;

- Utilização de programas maiores e em maior número para os experimentos, de modo a se viabilizar a derivação de modelos de predição, que tentariam estimar características de dificuldade de teste a partir das métricas coletadas;

Referências Bibliográficas

[1] G. Booch. *Object Oriented Design with Applications.* Editora Benjamin-Cummings. Califórnia, 1991.

[2] C.Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design,* 1999.

[3] N. E. Fenton e S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach.* International Computer Press, 1997.

[4] S. R. Chidamber e C. F. Kemerer. *A Metrics Suite for Object-Oriented Design.* In: IEEE Transation on Software Engineering, Junho 1994, pp. 476 – 493.

[5] M. Lorenz e J. Kidd. *Object-Oriented Software Metrics - A Practical Guide.* Englewood Cliffs, Prentice Hall, 1994.

[6] Y. S. Lee, B. S. Liang e F. J. Wang. *Some Complexity Metrics for Object-Oriented Programs Based on Information Flow: A Study of C++ Programs.* Institute of Computer Science and Information Engineering. National Chiao Tung University, 1996.

[7] Delamaro. M.E. *Proteum – Um ambiente de teste baseado na Análise de Mutantes.* Dissertação de Mestrado, ICMSC/USP – São Carlos, SP, Brasil, Outubro, 1993.

[8] J.D. McGregor. *Functional Testing of Classes.* Technical Report Clemson University, USA, www.cs.clemson.edu, 1995.

[9] F. B. Abreu and R. Carapuça. *Candidate Metrics for Object Oriented Sofftware within a Taxonomy Framework.* Journal of System and Software, vol 26, no. 1, pp 87-96, Jan. 1994.

[10] N. I. Churcher and M.J. Shepperd. *Comments on A Metrics Suite for Object-Oriented Design.* IEEE Transation on Software Engineering, , Mar. 1995, pp. 263-265.

[11] R. S. Pressman. *Software Engineering: A Practioner's Approach.* McGraw-Hill, New York, EUA, third edition, 1992.

[12] A. J. Albrecht. *Measuring Applications Development Productivity.* IBM Applications Development Symposium, Monterey, CA, 1979.

[13] P. Wegner: *Dimensions of Object-Based Language Design.* Object oriented Programming Systems Languages and Applications (OOPSLA). SIGPLAN, vol. 22, Dezembro de 1987, pp. 68-187.

[14] Bjarne Stroustrup. *The C++ Programming Language.* 3ª Edição. Addison-Wesley, 1997.

[15] Johan Eder, Gerti Kappel, Michael Schrefl. *Coupling and Cohesion in Object-Oriented Systems.* Institut für Informatic, Universität Klagenfurt, Klagenfurt, Austria, 1992

[16] E. J. Weyuker. *Evaluating Software Complexity Measure.* In IEEE Transaction on Software Engineering, Setembro de 1988, pp. 1357 – 1365.

[17] V. R. Basili, L. Briand e W. L. Melo. *A Validation of Object-Oriented Design Metrics as Quality Indicators.* Technical Report, Univ. of Maryland, Dep. of Computer Science, College Park, MD, 20742 USA. Abril, 1995.

[18] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, Julho 1991.

[19] R. A. De Millo. *Software Testing and Evaluation.* The Benamin/Cummings Publishing Company, Inc., 1987.

[20] R. A. De Millo. R.J. Lipton, F.G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer.* IEEE Computer, Abril, 1978.

[21] A. T. Acree. *On Mutation.* Tese de Doutorado. Georgia Institute osf Technology. Atlanta GA. 1980.

[22] M. L. Chaim. *POKE-TOOL – Uma ferramenta para Suporte ao Teste Estrutural de Programas Baseados em Análise de Fluxo de Dados.* Tese de Mestrado, DCA/FEEEC/Unicamp, Campinas - SP, Brasil, Abril de 1991.

[23] S. R. Vergilio. *Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes.* Tese de Doutorado, DCA, FEEC/Unicamp, Campinas, SP, Julho 1997.

[24] P. Naughton. Java Handbook – *The Authoritative Guide to the Java Revolution.* 1ª Edição. McGraw-Hill, 1996.

[25] Rational Corporation. Home page em http://www.rational.com/product/rose.

[26] Leonardo B. Chaves e Silvia R. Vergilio. *Resultados Empíricos da Utilização de Métricas para*

Software Orientado a Objeto na Atividade de Teste. Relatório Técnico, UFPR, Janeiro 2001.

[27] G. J. Myers. *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

[28] Peter J. Bickel e Kjell A. Dorksum. *Mathematical Statistics, Basic Ideas and Selected Topics.* Holden-Day Inc., 1977.

[29] Alexander M. Mood, Franklin A. Graybill e Duane C. Boes. *Introduction to the Theory of Statistics.* Mc-Graw-Hill Inc., 1974.

[30] Dewayne E. Perry and Gail E. Kaiser, *Adequate Testing and Object-oriented Programming*, Journal of Object-Oriented Programming, v 2, n 5, Jan/Feb 1990, pp. 13-19.

[31] Steven P. Fiedler, *Object-Oriented Unit Testing*, Hewlett-Packard Journal, April 1989, v. 40, n. 2, pp. 69-75.

[32] N. Wilde and R. Huitt, *Maintenance Support for Object-Oriented Programs*, IEEE Trans. Software Eng., Vol. 18, No. 12, Dec. 1992, pp. 1038-1044.

[33] M. Lejter, S. Meyers, and S.P. Reiss. *Support for Maintaining Object-Oriented Programs*, IEEE Trans. Software Eng., Vol. 18, No. 12, Dec. 1992, pp. 1045-1052

[34] S. Letovski and E. Soloway, *Delocalized Plans and Program Comprehension*, IEEE Software, Vol. 3, No. 3, May 1986, pp. 41-49.

[35] D.E. Perry and G.E. Kaiser, *Adequate Testing and Object-Oriented Programming*, J. Object-Oriented Programming, Vol. 2, Jan.-Feb. 1990, pp. 13-19.

[36] M.D. Smith and D.J. Robson, *Object-Oriented Programming—The Problems of Validation*, Proc. IEEE Conf. Software Maintenance, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 272-281.

[37] R. Binder (Guest Editor), *Object-Oriented Software Testing*, Comm. ACM, Vol. 37, No. 9, Sept. 1994, pp. 28-29.

[38] D. Kung, et al, *On Object State Testing*, Proc. COMPSAC '94, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 222-227.

[39] Jacobson et al. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley MA. 1992.

[40] Robert V. Binder, *The FREE-flow Graph: Implementation-based Testing of Objects Using State-determined Flows*, Proceedings, 8th Annual Software Quality Week. Software Research, Inc. San Francisco, 1994.

[41] Robert V. Binder, *The FREE Approach for System Testing: use-cases, Threads, and Relations, Object*, v 6, n 2, February 1996.

[42] Lionel C. Briand, John W. Daly, Victor Porter, Jürgen Wüst. *A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems*, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 1998.

[43] W. Howden. *Reliability of the path analysis testing strategies.* IEEE Transactions on Software Engineering, Setembro de 1976, pp. 208-215.

[44] J. B. Goodenough and S.L. Gerhart. *Toward a theory of test data selection.* IEEE Transactions on Software Engineering, Setembro de 1975, pp.156-173.

[45] V. R. Basili and C. U. Munoz. *Automatic generations of random selfchecking test cases.* IBM System Journal, Agosto de 1983, pp. 229-245.

[46] T. McCabe. *A software complexity measure.* IEEE Transactons on Software Engineering, Dezembro de 1976, pp. 308-320.

[47] S. Rapps, E. J. Weyuker. *Data Flow Analysis Techniques for Test Data Selection*, In Proc. Int. Conf. Software Eng., Tokio, Japão,Set. 1982.

[48] S. Rapps, E. J. Weyuker. *Selecting Software Test Data Using Data Flow Information*, IEEE Trans. on Software Engineering, Abril de 1985, pp. 367-375.

[49] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Tese de Doutorado, DCA/FEEC/Unicamp, Campinas, SP, Julho 1991.

[50] F. G. Frankl. *The use os Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD Thesis, Department of Computer Science, New York University, New York, USA, October 1987.

[51] T. A. Budd. *Mutation Analisys: Ideas, Examples, Problems and Prospects*, Computer Program Testing, North-Holand Publishing Company, 1981.

[52] John J. Marcianiak. *Object-Oriented Requiements Analysis.* Encyclopedia of Software Engineering, Vol. 1, 1984.

[53] Thomas Fetcke, Alain Abran e Tho-Hau Nguyen. *Mapping the OO-Jacobson Approach into Function Point Analysis.* Proceedings of TOOLS-23'97, Santa Barbara, CA, 1997.

[54] Oni Reasilvia de Almeida Oliveira Sichonany. *Métricas de Complexidade de Sistemas*

Orientados a Objeto. Tese de Mestrado, DCA/UFRGS, Porto Alegre, RS, Agosto 1999.

[55] L. Briand et al. *Theoretical and Empirical Validation of Software Product Measures*. Technical Report ISERN-95-03, International Software Engineering Research Network, 1995.

[56] K. N. King and A. J. Offutt. *A Fortran Language System for Mutation-Based Software Testing*. Software-Practice and Experience, Julho de 1991, pp.685-718.

[57] S. C. Ntafos. *On Requirement Element Testing*, IEEE Transactions on Software Engineering, SE-10(16), Novembro, 1984.

[58] S. C. Ntafos. *A Comparison of Some Structural Testing Strategies*, IEEE Transactions on Software Engineering, Junho de 1988, pp. 868-876.

[59] Padrão IEEE número 610.2-1990, 1990.

Apêndice A

Operadores de Mutação da Ferramenta Proteum

Os operadores de mutação estão divididos em quatro grupos: mutação de comandos, mutação de operadores, mutação de variáveis e mutação de constantes. Ao todo são 71 operadores de mutação disponíveis na ferramenta Proteum. Na Tabela A.1 são apresentados os operadores e seus significados. Informações mais detalhadas podem ser obtidas em [7].

Tabela A.1: Operadores de Mutação de Ferramenta Proteum.

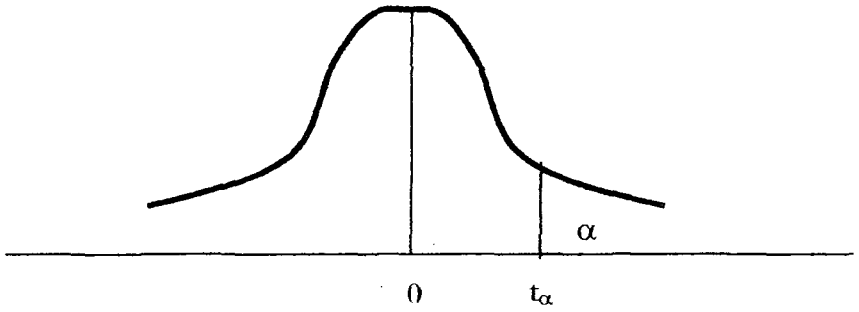
Operador	**Descrição**
STRP	Força a execução de todos os comandos do programa.
STRI	Força a execução para TRUE e FALSE em cada *if*
SSDL	Retira um comando de cada vez do programa.
SRSR	Troca cada comando por todos os *returns* que existem na função em teste.
SGRL	Troca os comando *goto* por todos os rótulos existentes na função.
SCRB	Troca do comando *continue* pelo comando *break*.
SBRC	Troca o comando *break* pelo comando quando é possível.
SBRn	Troca dos comandos continue ou break por uma função break_out_to_level_n(J), onde J varia de acordo com o número de laços aninhados. Essa função força a interrupção dos J laços externos.
SCRn	Troca dos comandos continue ou break por uma função break_out_to_level_n(J), onde J varia de acordo com o número de laços aninhados. Essa função força a transferência de programa para o final de J laços acima.
SWDD	Troca o comando *while* pelo comando *do-while*.
SDWD	Troca o comando *do-while* pelo comando *while*.
SMTT	Força a execução dos laços mais de uma vez.
SMTC	Interrompe a execução do laço após duas execuções.
SMVB	Move "}" para acima ou para baixo, quando possível.

SSWM	Força a execução do todos os *cases* do comando *switch*.
OAAA	Troca atribuição aritmética por outra atribuição aritmética.
OAAN	Troca operador aritmético por outro operador aritmético.
OABA	Troca atribuição aritmética por operador de atribuição *bitwise*.
OABN	Troca operador aritmético por outro operador *bitwise*.
OAEA	Troca atribuição aritmética por operador de atribuição plana.
OALN	Troca operador aritmético por operador lógico.
OARN	Troca operador aritmético por operador relacional.
OASA	Troca atribuição aritmética por atribuição de deslocamento.
OASN	Troca operador aritmético por operador de deslocamento.
OBAA	Troca atribuição *bitwise* por atribuição aritmética.
OBAN	Troca operador *bitwise* por operador aritmético.
OBBA	Troca atribuição *bitwise* por atribuição aritmética.
OBBN	Troca operador *bitwise* por operador *bitwise*.
OBEA	Troca atribuição *bitwise* por atribuição plana.
OBLN	Troca operador *bitwise* por operador lógico.
OBRN	Troca operador *bitwise* por operador relacional.
OBSA	Troca atribuição *bitwise* por operador de atribuição de deslocamento.
OBSN	Troca operador *bitwise* por operador de deslocamento.
OEAA	Troca atribuição plana por atribuição aritmética.
OEBA	Troca atribuição plana por atribuição *bitwise*.
OESA	Troca atribuição plana por operador de atribuição de deslocamento.
OLAN	Troca operador lógico por operador aritmético.
OLBN	Troca operador lógico por operador *bitwise*.
OLLN	Troca operador lógico por outro operador lógico.
OLRN	Troca operador lógico por operador relacional.
OLSN	Troca operador lógico por operador de deslocamento.
ORAN	Troca operador relacional por operador aritmético.
ORBN	Troca operador relacional por operador *bitwise*.
ORLN	Troca operador relacional por operador lógico.
ORRN	Troca operador relacional por outro operador relacional.
ORSN	Troca operador relacional por outro operador de deslocamento.
OSAA	Troca atributo de deslocamento por atribuição aritmética.
OSAN	Troca operador de deslocamento por operador aritmético.
OSBA	Troca atributo de deslocamento por atributo *bitwise*.
OSBN	Troca operador de deslocamento por operador *bitwise*.
OSEA	Troca atributo de deslocamento por atribuição plana.
OSLN	Troca operador de deslocamento por operador lógico.
OSRN	Troca operador de deslocamento por operador relacional.
OSSA	Troca atributo de deslocamento por outro de deslocamento.
OSSN	Troca de operador de deslocamento por outro de deslocamento.
Ouor	Troca operador de incremento/decremento.

OLNG	Insere negação lógica em condições compostas.
OCNG	Insere negação lógica.
OBNG	Insere negação no operador *bitwise*.
OIPM	Substitui os operadores de incremento/decremento que não possuem direção.
OCOR	Troca o tipo primitivo do operador *cast*.
Vsrr	Substitui as referência escalares por variáveis escalares, globais e locais do programa.
Varr	Substitui as referências a vetores por variáveis escalares, globais e locais do programa.
Vtrr	Substitui as referências a estruturas e uniões por variáveis escalares, globais e locais do programa.
Vprr	Substitui as referências a apontadores por variáveis escalares, globais e locais do programa.
VSCR	Troca as referência e componentes de uma estrutura por demais componentes da mesma estrutura.
VTDR	Força cada referência escalar possuir cada um dos valores: negativo, positivo e zero.
VTWD	Substitui referência escalar pelo seu valor sucessor e antecessor.
CRCR	Troca constante por 0, 1 ou –1, dependendo do tipo de referência.
Cccr	Troca constantes por todas as constantes do programa.
Cssr	Troca referência escalares por constantes.

Apêndice B

Valores Tabulados para o Teste t



Graus de Liberdade	$t_{.100}$	$t_{0.050}$	$t_{0.025}$	$t_{0.010}$	$t_{0.005}$
1	3.078	6.314	12.706	31.821	63.657
2	1.886	2.920	4.303	6.965	9.925
3	1.638	2.353	3.182	4.541	5.841
4	1.533	2.132	2.776	3.747	4.604
5	1.476	2.015	2.571	3.365	4.032
6	1.440	1.943	2.447	3.143	3.707
7	1.415	1.895	2.365	2.998	3.499
8	1.397	1.860	2.306	2.896	3.355
9	1.383	1.833	2.262	2.821	3.250
10	1.372	1.812	2.228	2.764	3.169
11	1.363	1.796	2.201	2.718	3.106
12	1.356	1.782	2.179	2.681	3.055
13	1.350	1.771	2.160	2.650	3.012
14	1.345	1.761	2.145	2.624	2.977
15	1.341	1.753	2.131	2.602	2.947
16	1.337	1.746	2.120	2.583	2.921
17	1.333	1.740	2.110	2.567	2.898
18	1.330	1.734	2.101	2.552	2.878
19	1.328	1.729	2.093	2.539	2.861
20	1.325	1.725	2.086	2.528	2.845
21	1.323	1.721	2.080	2.518	2.831
22	1.321	1.717	2.074	2.508	2.819
23	1.319	1.714	2.069	2.500	2.807

24	1.318	1.711	2.064	2.492	2.797
25	1.316	1.708	2.060	2.485	2.787
26	1.315	1.706	2.056	2.479	2.799
27	1.314	1.703	2.052	2.473	2.771
28	1.313	1.701	2.048	2.467	2.763
29	1.311	1.699	2.045	2.462	2.756
30	1.310	1.697	2.042	2.457	2.750
∞	1.282	1.645	1.960	2.326	2.576

Exemplo: O valor do limite t com 12 graus de liberdade para a área de 1% é de 2.681.