UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DE CONJUNTOS DE TESTES FUNCIONAIS NO CONTEXTO DE PROGRAMAS ORIENTADOS A OBJETOS E PROGRAMAS ORIENTADOS A ASPECTOS

Thiago Gaspar Levin

Orientador: Prof. Dr. Fabiano Cutigi Ferrari

São Carlos – SP

Novembro/2014

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DE CONJUNTOS DE TESTES FUNCIONAIS NO CONTEXTO DE PROGRAMAS ORIENTADOS A OBJETOS E PROGRAMAS ORIENTADOS A ASPECTOS

Thiago Gaspar Levin

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Fabiano Cutigi Ferrari

São Carlos – SP

Novembro/2014

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

L665ac

Levin, Thiago Gaspar.
Avaliação de conjuntos de testes funcionais no contexto de programas orientados a objetos e programas orientados a aspectos / Thiago Gaspar Levin. -- São Carlos : UFSCar, 2015.
112 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2015.

1. Software - testes. 2. Programação orientada a objetos (Computação). 3. Programação orientada a aspectos. 4. Estudos experimentais. I. Título.

CDD: 005.14 (20ª)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Thiago Gaspar Levin, realizada em 08/01/2015:

Prof. Dr. Fabiano Cutigi Ferrari
UFSCar

Prof. Dr. Valter Vieira de Camargo
UFSCar

Prof. Dr. Aurí Marcelo Rizzo Vincenzi
UFG

Aos meus pais.

Agradecimentos

Ao meu orientador Dr. Fabiano Cutigi Ferrari, pela grande orientação, incentivo, paciência, profissionalismo e pelo conhecimento transmitido na área de testes, sem o qual não teria realizado este trabalho.

Agradeço especialmente a toda minha família pelo amor, confiança e apoio na vinda para São Carlos. Em especial aos meus pais, Carlos e Cláudia, aos meus avós, Gaspar e Maria, e a minha namorada Francine, por acreditarem sempre em mim e nas minhas escolhas. Só eles sabem o quão difícil foram os obstáculos que enfrentei para chegar até aqui.

Ao meu avô Frido, aos tios Marco, Ivânia, Julio, Lu, Fábio, Lâ e Niva, aos meus primos Bruno, Ana, Sophia e Pedro, e aos pais da minha namorada, Ednan e Ni, por sempre me incentivarem e acreditarem em mim.

Aos meus amigos do LaPES pelas amizades proporcionadas nesses anos de convivência, em especial ao J. Thiago, Odair, Abade, Ana, Elis e André, e aos meus amigos de outros laboratórios, Stéfano, Porto, Tiago Rosa, Ely, Guido, João, Baiano, Vitinho, Maranhão, Renato e Danilo.

Aos professores do DC-UFSCar pelas lições aprendidas e desafios enfrentados.

Ao CNPq pelo apoio financeiro.

Ao Deus que acredito, por colocar pessoas fantásticas e situações inusitadas no meu caminho.

"O que não dá prazer não dá proveito. Em resumo, senhor, estude apenas o que lhe agrade!"

William Shakespeare

Resumo

Contexto: O teste de software é uma atividade importante para revelar defeitos e aumentar a qualidade nos produtos desenvolvidos. Testes estruturais têm se mostrados bastante importantes para a avaliação da qualidade do software ou mesmo do conjunto de testes utilizado para testá-lo.
Objetivo: Este trabalho apresenta um estudo experimental para avaliar a qualidade de conjuntos de testes criados para um paradigma em específico, quando adaptado e aplicado em outro paradigma, avaliando também o esforço para adaptar conjuntos de testes quando ocorre a migração de um paradigma para outro. Os paradigmas considerados são o paradigma orientado a objetos (OO) e o paradigma orientado a aspectos (OA).
Método: Para a avaliação da qualidade dos conjuntos de testes adaptados foi analisada a cobertura de código atingida em ambos os paradigmas; para a avaliação do esforço na adaptação dos conjuntos de testes, métricas de código foram comparadas em relação à implementações dos testes em cada paradigma considerado. No total, 12 aplicações de pequeno porte e uma aplicação de médio porte foram avaliadas.
Resultados: A cobertura de código atingida para diferentes grupos de aplicações não resultou em diferenças expressivas entre os paradigmas, em particular para as aplicações de pequeno porte. Somente no sistema de médio porte o código da implementação OO foi mais coberto do que na implementação OA. Em relação ao esforço na adaptação dos conjuntos de testes, os testes OO migrados para as implementações OA necessitaram de mais incrementos de código, enquanto os conjuntos de testes OA migrados para o paradigma OO necessitaram de mais decrementos e modificações.
Conclusão: Com os resultados alcançados não é possível afirmar que um conjunto de testes adaptado de um determinado paradigma possui maior qualidade em relação a outro paradigma, com exceção de uma aplicação, em que o conjunto de testes OO se apresentou com qualidade superior ao conjunto de testes OA. Em relação à migração de conjuntos de testes entre paradigmas, os testes OO apresentaram-se mais enxutos (em termos de linhas de código) e mais reutilizáveis do que os testes OA.

Palavras-chave: Teste de software, programação orientada a objetos, programação orientada a aspectos, estudos experimentais.

Abstract

Context: Software testing play an important role to reveal faults and increase the quality of developed products. Structural-based testing has shown to be important in the evaluation of both the quality of the software and the quality of the test set itself.

Objective: This work reports on the results of an experimental study that aimed to evaluate the quality of test sets which were originally built for a given programming paradigm, when such test sets are adapted and applied to a different paradigm. Besides this, this we also measure the effort to migrate test sets from one paradigm to another. In this work, we considered the object-oriented (OO) and aspect-oriented (AO) paradigms.

Method: We evaluated the quality of test sets by analysing the code coverage yielded by test sets within both paradigms; to evaluate the effort required to adapt test sets across paradigms, we performed comparisons of code-related metrics applied to test code. In total, we analysed 12 small-sized applications and one medium-sized application.

Results: The achieved coverage for distinct groups of applications did not show expressive differences across the paradigms, particularly for small-sized applications. Only for the medium-sized system tests yielded higher code coverage in the OO implementation. In respect with the test set adaptation effort, migrating OO tests to the AO paradigm required more code additions, whilst migrating AO tests to the OO implementations required more code changes and removals.

Conclusion: With the achieved results, we cannot state that there is a difference in the quality of the test sets, in terms of structural coverage, when both paradigms are taken into account. Only for a single application the coverage difference was evident. In regard to the process of migrating tests from one paradigms to another, OO tests have shown to be more concise (in terms of lines of code) and more reusable than OA tests.

Keywords: Software testing, object-oriented programming, aspect-oriented programming, experimental studies.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Contextualização

A construção de software é uma atividade muitas vezes complexa, sujeita a diversos tipos de problemas que podem levar à obtenção de um produto final que não corresponda às expectativas do usuário. Uma das bases para o desenvolvimento de um software está fundamentada na forma como os problemas do mundo real são entendidos e escritos em linhas de código por uma linguagem de programação. O paradigma de desenvolvimento é uma das abordagens para tratar essa questão, e tais paradigmas implicam em diferenças na forma de abstrair e representar essas estruturas.

Dentre os vários paradigmas utilizados para desenvolvimento de software, destacam-se dois: o paradigma Orientado a Objetos (OO) e o paradigma Orientado a Aspectos (OA), sendo que o primeiro foi concebido nos anos 70, enquanto o segundo foi proposto nos anos 90.

O paradigma OO – também conhecido como Programação Orientada a Objetos (POO) – caracteriza-se pelo entendimento do software como um conjunto de entidades com características próprias e bem definidas, denominadas objetos (BOOCH, 1994). Um objeto é definido por meio de uma classe, que é composta de atributos (dados) e métodos (funções). Algumas das características do paradigma OO são: encapsulamento, ocultação da informação, herança e polimorfismo.

O paradigma OA – ou Programação Orientada a Aspectos (POA) (KICZALES et al., 1997) – surgiu como uma possível solução para algumas dificuldades enfrentadas no desenvolvimento de software, principalmente relacionadas à separação de interesses envolvidos no software. A ideia principal é possibilitar que interesses que se encontram espalhados

ou entrelaçados pela aplicação sejam implementados tão separadamente quanto possível dos demais.

Tanto a POO quanto a POA trouxeram benefícios para o desenvolvimento de software, mas não isentam o software de conter defeitos. Os novos conceitos e elementos de construção de software introduzidos por esses paradigmas consistem em novas fontes de defeitos, representando novos desafios para a atividade de teste (ALEXANDER; BIEMAN; ANDREWS, 2004; BINDER, 1999).

Técnicas tradicionais como a funcional, estrutural e baseada em defeitos, em conjunto com seus respectivos critérios, têm sido investigadas quanto à sua aplicabilidade e, quando possível, adaptadas para cobrir necessidades específicas no contexto de software OO (HARROLD; ROTHERMEL, 1994; VINCENZI et al., 2006; LEMOS; FRANCHIN; MASIERO, 2009; CAFEO; MASIERO, 2011; LEMOS et al., 2013) e OA (ZHAO, 2003; LEMOS et al., 2007; LEMOS; FRANCHIN; MASIERO, 2009; LEMOS; MASIERO, 2011; FERRARI; RASHID; MALDONADO, 2013; LEMOS et al., 2013). Ressalta-se que neste trabalho, entende-se por teste de software OO (ou OA) como a aplicação de técnicas e critérios de teste em software desenvolvido com o emprego da POO (ou POA).

1.2 Motivação e Definição do Problema

No contexto de teste de software OO e OA, pontos fundamentais a serem considerados são:

- O teste de software é uma das atividades mais utilizadas para a garantia da qualidade no desenvolvimento de software;
- A avaliação experimental é fundamental para comparar critérios e técnicas de teste executados em diferentes contextos; e
- O paradigma de programação pode apresentar impacto sobre a realização da atividade de teste de software e das propriedades associadas (por exemplo, esforço e qualidade dos testes criados).

Levantados esses pontos, observa-se que na pesquisa em teste de software OA existe um certo consenso de que testar esse tipo de software é mais complexo do que testar software OO, quando conjuntos de funcionalidades equivalentes são considerados (ZHAO, 2003; CECCATO; TONELLA; RICCA, 2005; ZHAO; ALEXANDER, 2007; XIE; ZHAO, 2007;

FERRARI et al., 2013). Entretanto, não há evidência de que essa maior complexidade se reflita na prática. Ou seja, ainda não se demonstrou que de fato a POA impõe mais dificuldades para o teste de software do que outros paradigmas mais tradicionais, como a POO e a própria programação procedimental.

Outro fator importante é a reutilização de testes implementados em um paradigma, quando aplicado em outro. Por exemplo, um software pode ser implementado em uma linguagem para dispositivos móveis e o mesmo pode ser implementado em outra linguagem para outros dispositivos com as funcionalidades equivalentes entre os diferentes dispositivos. Com a migração do software, o testador pode ou não ter o trabalho de retestá-lo. Nesse contexto, é importante avaliar o quão reutilizáveis são os conjuntos de testes funcionais entre as diferentes linguagens de programação.

Tendo em vista a suposição de que há uma maior complexidade em testar aplicações OA em relação à aplicações OO e a viabilidade de reutilizar conjuntos de testes funcionais na refatoração de software entre diferentes paradigmas, é definido na próxima seção o objetivo deste trabalho.

1.3 Objetivo

O objetivo deste trabalho é avaliar a qualidade de conjuntos de testes entre os paradigmas OO e OA por meio do emprego de critérios da técnica estrutural e por métricas de esforço de implementação de software. Implementações de software OO e OA funcionalmente equivalentes foram utilizadas como objetos de estudo. A contribuição pretendida se deu por meio da avaliação desses conjuntos de testes adequados à técnica funcional pelo critério Teste Funcional Sistemático (SFT) para um paradigma em específico, quando aplicado no outro paradigma. Os conjuntos de testes criados a partir de um determinado paradigma foram migrados para a implementação do outro paradigma com o intuito de avaliar a suposição de que testar software OA é mais difícil do que testar software OO, conforme destacado na Seção 1.2.

1.4 Metodologia

Para atingir o objetivo proposto foi necessário realizar um estudo para avaliar a qualidade de conjuntos de testes executados em aplicações implementadas nos paradigmas OO e OA. A análise da qualidade se deu pela aplicação de critérios da técnica estrutural com

o aproveitamento dos conjuntos de testes funcionais criados, o que pode ser útil para analisar o impacto da migração entre conjuntos de testes implementados em um determinado paradigma para serem executados com as devidas adaptações no outro paradigma.

Foram coletadas 13 especificações com implementações funcionalmente equivalentes nos paradigmas OO e OA que foram divididas em dois grupos, Grupo-A e Grupo-B, totalizando 26 aplicações. Para o Grupo-A, foram criados conjuntos de testes tendo como base as aplicações OO implementadas na linguagem Java. Para o Grupo-B, os testes foram criados com base nas implementações OA com linguagem AspectJ.

Os conjuntos de testes foram criados para cada grupo baseado no critério teste funcional sistemático da técnica funcional e foram migrados para a implementação oposta para se tornarem executáveis de acordo com os elementos de programação empregados do respectivo paradigma. Com esses conjuntos de testes adaptados ao paradigma, testes estruturais foram executados para calcular a cobertura de código atingida entre as aplicações dos paradigmas OO e OA, tanto de conjunto de testes de implementações OO $\rightarrow$ OA (Grupo-A) quanto de aplicações OA $\rightarrow$ OO (Grupo-B).

Para a migração dos conjuntos de testes entre os paradigmas foi analisada a quantidade de linhas de código (LOC) de teste adicionadas, modificadas e removidas para o conjunto de teste adaptado. Com isso pôde-se comparar o quão difícil é testar e migrar conjunto de testes entre diferentes paradigmas. Após os conjuntos de testes serem adaptados aos paradigmas foi realizado o teste estrutural para a análise de cobertura de código atingida. Finalmente pôde-se comparar a qualidade desses conjuntos de testes em relação à cobertura de código atingida nas implementações de ambos os paradigmas.

1.5 Organização

Neste capítulo foram apresentados o contexto, a motivação e a definição do problema, o objetivo e a metodologia em que o trabalho está inserido. O restante do documento está disposto da seguinte forma:

No Capítulo 2 são apresentados os fundamentos de teste de software, teste de software orientado a objetos e orientado a aspectos com seus desafios e tipos de defeitos, e algumas abordagens para teste estrutural OO e OA. São apresentados os fundamentos de engenharia de software experimental, o processo de experimentação, alguns estudos teóricos e experimentais de técnicas e critérios de teste, e alguns estudos experimentais de teste de software no contexto inter-paradigma de programação.

No Capítulo 3 são descritos o planejamento e a execução do experimento. No planejamento é descrito a definição do estudo com a abordagem GQM (*Goal Question Metric*), detalhando os objetivos, as questões e as métricas utilizadas, a seleção do contexto, o desenho do experimento, a preparação do ambiente de testes, os artefatos gerados e a descrição das aplicações. Na execução é descrito o procedimento para a criação dos conjuntos de testes, o procedimento para coleta de métricas de LOC (*Lines Of Code*) com a aplicação do critério teste funcional sistemático e o procedimento para coleta de métricas de cobertura de código com a aplicação dos critérios todas-instruções e todos-desvios.

No Capítulo 4 são apresentadas a análise e a interpretação dos resultados, com as métricas de LOC e de cobertura de código, a análise e discussão sobre o esforço na migração de conjuntos de testes entre os paradigmas, da cobertura estrutural atingida pelos conjuntos de testes adaptados entre os paradigmas, uma análise adicional sobre a geração de requisitos estruturais de teste pela ferramenta de teste utilizada e uma comparação de resultados entre trabalhos relacionados.

No Capítulo 5 são descritos a conclusão e trabalhos futuros, com as contribuições, dificuldades e limitações.

Capítulo 2

Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo são apresentados os conceitos básicos relacionados à atividade de teste de software, conceitos de Engenharia de Software Experimental e alguns estudos experimentais no contexto inter-paradigma de programação.

O teste é uma atividade de grande importância no desenvolvimento de software, ajudando na redução de defeitos e aumentando a confiança nos produtos. Devido aos diversos critérios de seleção de casos de teste atualmente disponíveis, um ponto importante nessa perspectiva é a determinação de uma estratégia, que passa pela escolha de critérios de teste, fazendo com que as vantagens de cada critério sejam combinadas de forma que se tenha uma atividade de teste com maior qualidade. Estudos teóricos e experimentais são de muita importância para a formação desse conhecimento, fornecendo formas para estabelecer estratégias de baixo custo e alta eficácia (MALDONADO et al., 1998).

Neste capítulo, a Seção 2.2 apresenta um embasamento sobre as principais técnicas e critérios de teste. Nas Subseções 2.2.1 e 2.2.2 são apresentadas as características do teste de software OO e OA com seus respectivos desafios, além de algumas abordagens encontradas na literatura. Também são apresentados na Seção 2.3 os conceitos da Engenharia de Software Experimental e o funcionamento de um processo de experimento, e alguns trabalhos sobre teste de software que envolvem o caráter inter-paradigma no contexto experimental. Por fim, na Seção 2.4 são apresentadas as considerações finais deste capítulo.

2.2 Teste de Software

O teste de software tem como principal objetivo revelar defeitos presentes no software, sendo que um teste bem sucedido é aquele que revela a presença de um ou mais defeitos até então não encontrados (MYERS; BADGETT; SANDLER, 2011). Apesar de não ser possível por meio da atividade de teste, em geral, provar que uma aplicação está correta, o teste contribui para aumentar a confiança de que a aplicação desempenha as funções especificadas, quando executado de forma sistemática e criteriosa (HARROLD, 2000; WEYUKER, 1996).

A seguir são apresentados os conceitos fundamentais relacionados ao teste de software. Dentre eles, destacam-se os conceitos de defeito, engano, erro e falha; as fases de teste; o processo de teste; e as técnicas e critérios para seleção de casos de teste.

Defeito, engano, erro e falha: Neste trabalho, as definições utilizadas para os termos falha, defeito, erro e engano seguem a terminologia em Engenharia de Software estabelecida pelo padrão IEEE 610.12-1990 (IEEE, 1990). Um **defeito** (*fault*) trata-se de um passo, processo ou definição de dados incorretos, e que é inserido no software devido a um **engano** (*mistake*) cometido durante o desenvolvimento do software. A existência de um defeito pode ocasionar a ocorrência de um **erro** (*error*) durante uma execução da aplicação, que se caracteriza por um estado inconsistente ou inesperado. Tal estado pode levar a uma **falha** (*failure*), ou seja, pode fazer com que o resultado produzido pela execução do software seja diferente do resultado esperado.

Fases de teste: Segundo Delamaro, Maldonado e Jino (2007), a atividade de teste é dividida em fases com objetivos distintos. De uma forma geral, pode-se estabelecer como fases o teste de unidade, de integração e de sistema, que são realizadas em sequência:

- **Teste de Unidade:** o teste de unidade tem como foco as menores unidades de uma aplicação, que podem ser funções, procedimentos, métodos ou classes. Neste contexto, espera-se que sejam identificados defeitos relacionados a algoritmos incorretos ou mal implementados, ou simples enganos de programação. Cada unidade é testada separadamente, então o teste de unidade pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio desenvolvedor, sem a necessidade do sistema estar totalmente finalizado.
- **Teste de Integração:** após testadas as unidades do sistema, a ênfase é dada na construção da estrutura do sistema. À medida que as diversas partes do software são

colocadas para trabalhar juntas, é preciso verificar a interação entre elas de maneira adequada. Neste caso é necessário um grande conhecimento das estruturas internas e das interações existentes entre as partes do sistema, por isso, o teste de integração tende a ser executado pela própria equipe de desenvolvimento.

- **Teste de Sistema:** após feito o teste de integração e o sistema estar completo, inicia-se o teste de sistema. O objetivo é verificar se as funcionalidades especificadas nos documentos de requisitos estão todas corretamente implementadas, como os requisitos funcionais e não funcionais.

Além dessas três fases de teste, destaca-se ainda o teste de regressão. Esse tipo de teste não se realiza durante o processo "normal" de desenvolvimento, mas sim durante a manutenção de software. A cada modificação efetuada no sistema, após a sua liberação, corre-se o risco de que novos defeitos sejam introduzidos. Ressalta-se que o teste de regressão é transversal às três fases anteriormente descritas, pois pode ser aplicado em qualquer um dos três níveis: unidade, integração e sistema.

Processo de teste: Independentemente da fase de teste, existem algumas etapas bem definidas para a execução da atividade de teste. Tais atividades são contempladas por processos de teste em geral. São elas:

1. Planejamento de testes: são definidos como os testes serão aplicados, os objetivos, os métodos, os critérios e ferramentas adotadas para a realização dos testes;

2. Projeto de casos de teste: é projetada uma estratégia para a construção dos casos de testes com o objetivo de cobrir as condições de satisfação dos critérios estabelecidos;

3. Execução: com ou sem o auxílio de uma ferramenta, os casos de testes construídos na fase anterior são executados na aplicação em teste e sua saída comparada com a saída esperada. Caso seja revelado algum defeito, este é documentado;

4. Análise dos resultados dos testes: os resultados dos testes realizados são registrados, organizados e apresentados na forma de relatórios.

Dentre essas atividades, Myers, Badgett e Sandler (2011) citam o projeto de casos de teste como sendo a mais importante. Porém, o teste completo de um software é uma atividade geralmente impraticável devido ao seu alto custo computacional e financeiro. Com isso, o objetivo então passa a ser minimizar a não-completude do teste. Nessa

direção, Myers, Badgett e Sandler (2011) sugerem a definição de conjuntos de casos de teste que apresentem alta probabilidade de revelar a maioria dos defeitos com esforço e tempo reduzidos.

Caso de Teste: Um caso de teste é um par ordenado $(d, S(d))$ tal que d é um elemento de um determinado domínio D ($d \in D$), e $S(d)$ é a saída esperada para uma dada função da especificação, quando d é utilizado como entrada.

Técnicas e Critérios de Teste: Uma das principais questões relacionadas à atividade de teste refere-se ao tamanho do domínio de entrada, que pode ser muito grande para certas funcionalidades do software. Nesse contexto, critérios de seleção de casos de teste – ou simplesmente *critérios de teste* – auxiliam o testador a obter subconjuntos desse domínio, reduzindo a quantidade de casos de teste que devem ser criados, além de auxiliarem na avaliação de conjuntos de casos de teste (FRANKL; WEYUKER, 2000).

Cada critério de teste está associado a uma técnica de teste específica, que se baseia em diferentes artefatos de software para derivar os requisitos de teste. Os critérios são utilizados como diretrizes para delimitar a quantidade de requisitos, determinando os elementos e características do software que devem ser exercitados.

Uma ou mais técnicas de teste podem ser aplicadas em cada uma das fases de teste. Dentre as mais investigadas, destacam-se as técnicas funcional, estrutural e baseada em defeitos. Para cada uma das técnicas apresentadas, são sucintamente descritos os principais critérios a partir dos quais os requisitos de teste são derivados.

Teste Funcional

O teste funcional é uma técnica utilizada para projetar casos de teste na qual o software é uma *caixa-preta* (MYERS; BADGETT; SANDLER, 2011). Para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados. Nessa técnica, os detalhes de implementação não são considerados e o software é avaliado segundo o ponto de vista do usuário.

Em princípio, o teste funcional pode detectar todos os defeitos, submetendo a aplicação ou sistema a todas as possíveis entradas, o que é denominado *teste exaustivo* (MYERS; BADGETT; SANDLER, 2011). No entanto, em geral, o domínio de entrada pode ser muito grande ou até mesmo infinito, tornando o tempo da atividade de teste inviável.

Para particionar o domínio de entrada de dados e, consequentemente, viabilizar o teste em termos de custo e esforço para realização, os seguintes critérios de teste funcional

são tipicamente empregados:

Particionamento de Equivalência: tem o objetivo de apoiar a determinação de um subconjunto de todas as possíveis entradas de uma aplicação, tornando a quantidade de dados de entrada finita e mais viável para uma atividade de teste. Esse critério divide o domínio de entrada em classes de equivalência que, de acordo com a especificação da aplicação, são tratadas da mesma maneira. Assim, uma vez definidas as classes de equivalência, pode-se assumir, com alguma segurança, que qualquer elemento da classe pode ser considerado um representante da classe, pois todos eles devem se comportar de forma similar, ou seja, se um elemento detectar um defeito, qualquer outro também detecta; se não detectar, os outros também não detectam. De acordo com Myers, Badgett e Sandler (2011), uma classe de equivalência representa um conjunto de estados válidos ou inválidos para as condições de entrada.

Análise do Valor Limite: Esse critério é usado em conjunto com o Particionamento de Equivalência. A diferença é que os dados de teste não são escolhidos aleatoriamente dentro da faixa de valores da classe, mas sim selecionados com base nos valores que estão na fronteira superior e inferior de cada uma das classes, tanto para entradas quanto para saídas (MYERS; BADGETT; SANDLER, 2011).

Teste Funcional Sistemático: Segundo Linkman, Vincenzi e Maldonado (2003), o objetivo desse critério (do inglês, *Systematic Functional Testing* – SFT) é associar os benefícios de testes funcionais, combinando os critérios Particionamento de Equivalência e Análise do Valor Limite, resultando em uma maior cobertura de teste. O conjunto de teste deve incluir pelo menos dois casos de teste que cubram cada classe de equivalência e um caso de teste para cobrir cada valor limite. Possui uma série de diretrizes para construção dos casos de testes relacionados a casos especiais, valores ilegais, vetores, valores reais, entre outros. De acordo com Myers, Badgett e Sandler (2011), para o critério Particionamento de Equivalência, um único caso de teste pode ser desenhado para cobrir mais de uma classe de equivalência válida. Com isso, o número total de casos de teste para o critério SFT não é necessariamente duas vezes maior na soma dos critérios Particionamento de Equivalência e Análise do Valor Limite.

Teste Estrutural

Segundo Pressman (2010), a técnica estrutural é vista como complementar à técnica funcional, porém baseia-se no conhecimento da estrutura interna da implementação. A

técnica estrutural (também conhecida como teste *caixa-branca*) estabelece os requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou de componentes elementares da aplicação.

A técnica estrutural utiliza como auxílio na definição dos requisitos uma representação do código-fonte denominada Grafo de Fluxo de Controle (GFC). Para representar a aplicação em um GFC é preciso decompor o mesmo em blocos, isto é, regiões de código sequencial sem qualquer salto de execução. Dessa forma, todos os comandos contidos em um bloco, com exceção do primeiro e do último, possuem exatamente um comando predecessor e um sucessor.

A principal dificuldade em aplicar o teste estrutural encontra-se na determinação dos *caminhos não executáveis*. Um caminho não executável é um caminho do GFC que não pode ser coberto por nenhum valor do domínio de entrada.

Com base no GFC, alguns critérios são propostos como critérios baseados em fluxo de controle e critérios baseados em fluxo de dados. Esses critérios são sucintamente apresentados na sequência.

Critérios de Fluxo de Controle

De acordo com Myers, Badgett e Sandler (2011) e Pressman (2005), os critérios baseados em fluxo de controle utilizam apenas as informações do controle da execução da aplicação, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos dessa classe são:

Todos-Nós: requer que os casos de testes executem ao menos uma vez cada vértice do GFC, ou seja, que todos os comandos são executados ao menos uma vez.

Todas-Arestas (ou Todos-Arcos): requer que cada desvio do fluxo de controle da aplicação seja exercitado pelo menos uma vez, ou seja, que cada aresta do grafo seja coberta.

Todos-Caminhos: requer que todos os caminhos possíveis do GFC sejam executados.

É importante ressaltar que a cobertura do critério Todos-Nós é o mínimo esperado de uma boa atividade de teste, pois a aplicação em teste é entregue com a certeza de que todos os comandos presentes foram executados ao menos uma vez. Além disso, executar todos os caminhos da aplicação é, na maioria dos casos, uma tarefa impraticável, pois na presença de laços, o número de caminhos de uma aplicação pode ser muito grande ou até mesmo infinito (MYERS; BADGETT; SANDLER, 2011).

Critérios Baseados em Fluxo de Dados

Segundo Delamaro, Maldonado e Jino (2007), uma motivação para a introdução dos critérios baseados em fluxo de dados foi que, mesmo para aplicações pequenas, o teste baseado somente no fluxo de controle não era eficaz para revelar a presença de defeitos. Foram então propostos os critérios baseados em fluxo de dados, que levam em consideração as definições e usos de variáveis dentro de uma aplicação. Os caminhos no fluxo de controle que vão de uma definição de uma dada variável até o uso dessa variável (sem que ocorram redefinições do seu valor) constituem os requisitos de teste para esses critérios. Dentre os critérios de fluxo de dados, destacam-se os propostos por Rapps e Weyuker (1982) que utilizam uma extensão do GFC denominada Grafo Def-Uso e que podem derivar requisitos de teste baseados em informações de fluxo de dados da aplicação.

Nesse grafo, além das arestas e nós criados no GFC, são inseridas também informações a respeito do fluxo de dados. Com isso é possível identificar pontos em que um valor é atribuído a uma variável (caracterizando uma definição de variável) e pontos onde esses valores atribuídos são utilizados (caracterizando um uso de variável). O uso de uma variável pode ocorrer de duas formas: caso seja utilizado para determinar o desvio do fluxo de controle da aplicação, ele é chamado de uso predicativo (p-uso). Quando o valor da variável é utilizado em uma computação, ele é classificado como uso computacional (c-uso) (RAPPS; WEYUKER, 1982).

A premissa geral é que todo par definição-uso de uma variável deve ser exercitado por ao menos um caso de teste, para aumentar a confiança no correto funcionamento da aplicação. Os critérios apresentados a seguir têm essas características e fazem parte de duas famílias de critérios propostas por Rapps e Weyuker (1982) e por Maldonado (1991):

Todas-Definições: requer que cada variável definida em cada nó do GFC seja exercitada atingindo qualquer um de seus usos pelo menos uma vez. O critério é baseado na hipótese de que se um valor é atribuído a alguma variável, então essa variável deve ser utilizada posteriormente, caso contrário, não existe a necessidade de tal atribuição (RAPPS; WEYUKER, 1982).

Todos-Usos: requer que todas as associações entre uma definição de variável e seus subsequentes usos (c-usos e p-usos) sejam exercitados pelos casos de teste, por pelo menos uma vez, em um caminho livre de definição; ou seja, deve-se criar um conjunto de casos de teste que exercite cada caminho entre a definição de variável e todos os usos dessa variável, antes que ela seja redefinida (RAPPS; WEYUKER, 1982).

Todos-Du-Caminhos (ou Todos-P-Usos e Todos-C-Usos): requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação (RAPPS; WEYUKER, 1982).

Todos-Potenciais-Usos: requer que pelo menos um caminho livre de definição de cada variável definida em cada nó do grafo e todos os nós alcançáveis do grafo onde possam ocorrer o uso da variável sejam exercitados, antes que a variável seja redefinida (MALDONADO, 1991). A motivação para esse critério é que a não utilização de uma variável em um determinado ponto da aplicação pode representar um defeito de implementação.

Teste Baseado em Defeitos

As abordagens de teste baseado em defeitos utilizam as informações sobre os tipos de enganos mais frequentes cometidos no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos defeitos que o programador ou projetista pode inserir no código durante o desenvolvimento, e como abordar o problema para detectar a presença dos defeitos (MALDONADO et al., 2000). A seguir são apresentados os critérios baseados em defeitos mais investigados pela comunidade de Engenharia de Software:

Semeadura de defeitos: consiste basicamente na inserção randômica de um número pré-definido de defeitos no software. Um conjunto de casos de teste é executado para que sejam detectados os defeitos semeados e para que seja possível obter a taxa real de defeitos semeados e defeitos reais presentes no software que foram revelados (BUDD, 1981).

Teste de Mutação (ou Análise de Mutantes): segundo DeMillo, Lipton e Sayward (1978), esse critério consiste em gerar várias versões do produto original ligeiramente modificadas, com o objetivo de revelar os defeitos mais comuns introduzidos nos produtos pelos programadores. Com isso, o conjunto de casos de teste também é avaliado quanto a sua capacidade de revelar esses defeitos. O trabalho do testador é escolher casos de teste que mostrem a diferença de comportamento entre a aplicação original e as aplicações modificadas, chamadas *mutantes*. De acordo com DeMillo, Lipton e Sayward (1978), a aplicação desse critério tem como base a *Hipótese do Programador Competente* e a *Hipótese do Efeito de Acoplamento*. A

primeira assume que aplicações a serem testadas estão corretas ou próximas do correto. A segunda assume que casos de teste capazes de revelar defeitos mais simples também revelam defeitos mais complexos. Um defeito simples está associado a uma alteração sintática simples, enquanto que um defeito complexo pode ser considerado como uma composição de defeitos simples.

2.2.1 Teste de Software Orientado a Objetos

A POO surgiu para tentar suprir as deficiências dos paradigmas já existentes, principalmente relacionadas ao paradigma procedimental, a fim de fornecer um mecanismo para isolar os dados da forma como são manipulados. A ideia da Orientação a Objetos é agrupar em uma entidade (chamada classe) os dados (chamado atributos) e as funções (chamada métodos) que realizam operações sobre os dados. Com isso, os dados podem permanecer isolados (encapsulados na classe) e o acesso a eles só pode ser feito por meio dos métodos definidos na classe (BINDER, 1999).

Características específicas do paradigma OO como herança, encapsulamento, polimorfismo e acoplamento dinâmico trouxeram benefícios. Entretanto, não há motivos para supor que programadores estão menos propensos a cometer enganos durante a codificação das aplicações com o emprego dos mecanismos que implementam tais características (BINDER, 1999). O que se observa é que as características principais da Orientação a Objetos introduzem novas fontes de defeitos representadas como desafios para o teste de software OO. Ressalta-se que o termo *teste de software OO* (ou simplesmente *teste OO*) pode resultar em dupla interpretação: (1) emprego de conceitos OO para testar um software; ou (2) emprego de técnicas e critérios de teste para testar software desenvolvido sob o paradigma OO. Neste trabalho, utiliza-se a interpretação (2) aqui descrita. Um sumário dos principais desafios relacionados ao teste OO é apresentado a seguir.

Desafios para o Teste de Software OO

As ocorrências de alguns tipos de defeitos comuns na utilização de programação procedimental são reduzidas com a adoção da POO (BINDER, 1999). Por exemplo, os métodos de uma classe geralmente possuem poucas linhas de código, portanto, defeitos de fluxo de controle são menos prováveis. O mecanismo de encapsulamento reduz o risco de defeitos relacionados ao acesso global à variáveis (VINCENZI, 2004). Entretanto, a possibilidade de ocorrência de outros tipos de defeitos pode aumentar. Por exemplo, métodos podem aparecer em grande quantidade dentro de uma classe, o que pode aumentar a quantidade

de defeitos relacionados à comunicação entre os módulos, fazendo com que aumente a necessidade de testes de integração (KIM; CLARK; MCDERMID, 1999).

Tipos de Defeitos de Software OO

A seguir são apresentadas algumas dificuldades para o teste e alguns possíveis defeitos que podem ocorrer em função das características específicas de aplicações OO:

- **Encapsulamento:** pode representar um obstáculo para a atividade de teste, pois o estado de um objeto, que é um requisito necessário para o teste, pode ser de difícil obtenção (BINDER, 1999).
- **Herança:** pode enfraquecer o encapsulamento e grandes hierarquias de herança podem dificultar a compreensão, aumentar a chance de ocorrência de defeitos e reduzir a testabilidade das classes (BINDER, 1999).
- **Herança múltipla:** pode implicar que a mudança em qualquer uma das superclasses pode resultar em interações indesejadas nas subclasses (BINDER, 1999). Outro risco é a herança repetida, quando uma superclasse aparece mais de uma vez na hierarquia de herança. O teste na presença de herança repetida é ainda mais complicado que no caso de herança múltipla, uma vez que existe um número de características que são renomeadas ou removidas e a possibilidade de defeitos aumenta. Por exemplo, existem as classes B e C derivadas da superclasse A, e uma classe D derivada de B e C. Essa estratégia pode levar à ocorrência de defeitos, pois se os métodos e atributos não forem explicitamente qualificados pelo nome da classe, pode ocorrer um conflito de nomes (DELAMARO; MALDONADO; JINO, 2007).
- **Classes abstratas:** esses tipos de classes fornecem somente uma interface sem nenhuma implementação, oferecendo um importante apoio para o reuso (BINDER, 1999). Segundo Delamaro, Maldonado e Jino (2007), o teste de uma classe abstrata só poderá ser realizado após esta ter sido especializada e uma classe concreta ter sido obtida. Não é possível criar objetos ou instâncias de classes abstratas. Esse processo pode ser complicado se um método concreto utiliza um método abstrato para implementar sua funcionalidade.
- **Classes genéricas:** segundo Smith e Robson (1990), os principais problemas no teste de classes genéricas são:

1. Classes genéricas precisam ter o parâmetro genérico substituído por um parâmetro de um tipo específico para serem testadas.
2. Da mesma forma que superclasses abstratas, as classes derivadas da classe genérica devem ser retestadas se mudanças são feitas nas classes genéricas.

- **Polimorfismo:** traz indecibilidade para o teste, uma vez que é impossível predizer em tempo de compilação qual trecho de código será executado, tendo em vista que nomes polimórficos podem denotar objetos de diferentes classes (BARBEY; STROHMEIER, 1994) *apud* (VINCENZI, 2004).

- **Outros problemas:** alguns outros problemas são descritos por Binder (1999), podendo gerar defeitos relacionados com sequências de mensagens e estados de objetos. Uma sequência incorreta de mensagens pode levar um objeto a um estado inconsistente, o que carcteriza um estado de erro (MCDANIEL; MCGREGOR, 1994).

Fases de Teste OO

Nos trabalhos de Harrold e Rothermel (1994), Colanzi (1999) e outros posteriores, define-se que a menor unidade a ser testada em uma aplicação OO é o método. Contudo, ressalta-se que sem a existência da classe, não é possível executar os métodos. Dessa forma, na literatura de Engenharia de Software podem-se encontrar trabalhos que definem a classe como a menor unidade a ser testada, o que implica em uma abordagem de teste em que testes de unidade e de integração devem ser vistos de uma perspectiva diferente.

No paradigma procedimental, o teste de unidade também é chamado de *intraprocedimental* e no paradigma OO, **intra-método**. Por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos. Assim sendo, considerando uma única classe, já é possível pensar em teste de integração (HARROLD; ROTHERMEL, 1994). Harrold e Rothermel (1994) definem os níveis de teste a seguir:

Intra-métodos: são métodos da mesma classe que podem interagir entre si para desempenhar funções específicas caracterizando uma integração entre módulos que deve ser testada e esses métodos são testados separadamente. No paradigma procedimental essa fase também pode ser chamada de teste intraprocedimental.

Inter-métodos: são testados métodos públicos junto com outros métodos da classe que são chamados direta e indiretamente. O nível de teste é equivalente ao teste de integração em aplicações procedimentais.

Intra-classe: são testadas as interações entre métodos públicos fazendo chamadas a esses métodos em diferentes sequências. O objetivo é identificar possíveis sequências de ativação de métodos inválidas que levem o objeto a um estado inconsistente.

Pequenas variações quanto à divisão das fases de teste para aplicações OO são identificadas na literatura. Por exemplo, Colanzi (1999) caracteriza a fase do teste de classe, que tem por objetivos descobrir defeitos de integração entre métodos dentro do escopo da classe em teste, e a fase do teste de integração para software OO que tem por objetivo encontrar defeitos na integração de classes do sistema. Assim, segundo Colanzi (1999), o teste de POO é organizado em quatro fases:

- **Teste de Unidade:** testa os métodos individualmente;
- **Teste de Classe:** testa a interação entre métodos de uma classe;
- **Teste de Integração:** testa a interação entre classes do sistema; e
- **Teste de Sistema:** testa a funcionalidade do sistema como um todo.

Abordagem para Teste Estrutural de Software OO - Vincenzi et al. (2006)

O trabalho de Vincenzi et al. (2006) é uma das abordagens recentes para teste de unidade, mas outras abordagens já foram desenvolvidas para explorar o teste de integração, como os trabalhos de Lemos, Franchin e Masiero (2009) e Cafeo e Masiero (2011).

Um conjunto de critérios estruturais foi proposto por Vincenzi et al. (2006) para a derivação de requisitos a partir de código compilado da linguagem Java, conhecido como *bytecode*. O *bytecode* é formado por um conjunto de instruções que são interpretadas e executadas pela Máquina Virtual Java (do inglês, *Java Virtual Machine*, JVM) e pode ser vista como uma linguagem de máquina intermediária (KAMIN; MICKUNAS; REINGOLD, 2002; SAVITCH, 2001).

Um modelo de fluxo de dados subjacente é definido, caracterizando as instruções *bytecode* responsáveis pelas definições e pelos usos de variáveis. A partir desse modelo, o grafo DU (Def-Uso) é construído. Os autores tratam de forma separada com os mecanismos de manipulação e tratamento de exceção em Java. Com isso, o grafo DU é utilizado para representar o fluxo de controle e de dados intra-método, tanto da execução normal quanto do fluxo relacionado ao tratamento de exceções.

Na Figura 2.1 é apresentado um exemplo de grafo DU. É apresentado um trecho do *bytecode* de método que possui um tratador de exceções na parte da Figura 2.1(a). O

grafo DU desse método está na parte (b) da Figura 2.1. Os nós regulares da unidade são rotulados com a primeira instrução em *bytecode* do bloco de instruções representado pelo nó, os nós em negrito representam nós de saída, nós com círculos duplos representam nós de chamada e as arestas pontilhadas representam desvios do fluxo de execução para tratadores de exceção. As informações sobre definições e usos de variáveis estão associadas a cada nó ou aresta do grafo.

```
 0 aload_1
 1 getfield javaFonts.figura.Ponto.x I
   ...
 9 aload_1
10 getfield javaFonts.figura.Ponto.y I
   ...
18 aload_1
19 iload_2
   ...
43 aload_1
44 aload_1
45 invokevirtual javaFonts.figura.Ponto.
   printPonto (LjavaFonts/figura/
   Ponto;)V
   ...
61 astore %4
63 getstatic java.lang.System.out
   Ljava/io/PrintStream;
66 ldc "Excecao capturada!" (50)
68 invokevirtual java.io.PrintStream.
   println (Ljava/lang/String;)V
71 return
```

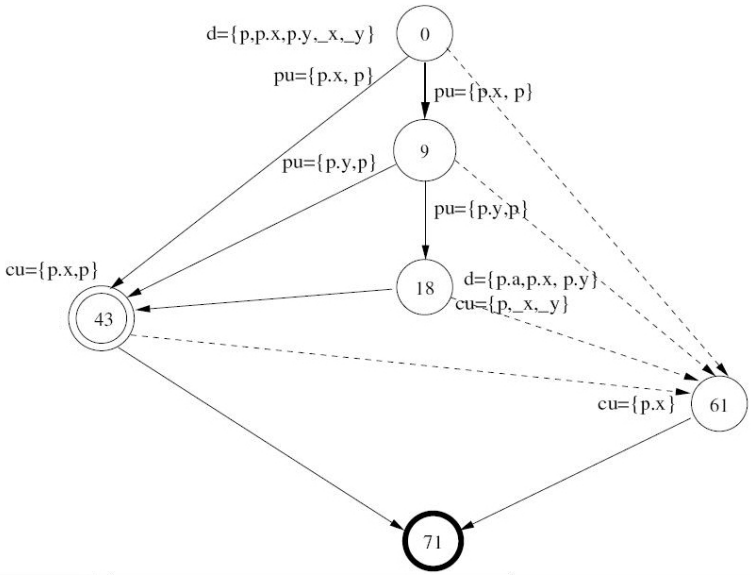

Figura 2.1: *Bytecode* Java e respectivo grafo DU (LEMOS, 2005).

Com base no grafo DU, Vincenzi et al. (2006) definem quatro critérios baseados em fluxo de controle e quatro critérios baseados em fluxo de dados, cujas descrições em alto nível são apresentadas a seguir:

- **Todos-Nós-Independentes-de-Exceção (Todos-Nós$_{ei}$):** requer que cada nó de um grafo DU que é alcançável por meio de um caminho livre de exceção seja exercitado por algum caso de teste.

- **Todos-Nós-Dependentes-de-Exceção (Todos-Nós$_{ed}$):** requer que cada nó de um grafo DU que não é alcançável por meio de pelo menos um caminho livre de exceção seja exercitado por algum caso de teste.

- **Todas-Arestas-Independentes-de-Exceção (Todas-Arestas$_{ei}$):** requer que cada aresta de um grafo DU que é alcançável por meio de um caminho livre de exceção seja exercitada ao menos uma vez por algum caso de teste.

- **Todas-Arestas-Dependentes-de-Exceção (Todas-Arestas$_{ed}$):** requer que cada aresta de um grafo DU que não é alcançável por meio de um caminho livre de exceção

seja exercitada ao menos uma vez por algum caso de teste.

- **Todos-Usos-Independentes-de-Exceção (Todos-Usos$_{ei}$):** requer que todo par def-c-uso e def-p-uso independente de exceção e livre de definição, ou seja, alcançável por um caminho que não inclua lançamento ou tratamento de exceções, seja exercitado por algum caso de teste.

- **Todos-Usos-Dependentes-de-Exceção (Todos-Usos$_{ed}$):** requer que todo par def-c-uso e def-p-uso dependente de exceção e livre de definição, ou seja, alcançável por um caminho que inclua lançamento ou tratamento de exceções, seja exercitado por algum caso de teste.

- **Todos-Potenciais-Usos-Independentes-de-Exceção (Todos-Pot-Usos$_{ei}$):** requer que seja exercitado pelo menos um caminho livre de exceção e livre de definição de uma variável definida em um nó n para todo nó e toda aresta possível de ser alcançada a partir de n.

- **Todos-Potenciais-Usos-Dependentes-de-Exceção (Todos-Pot-Usos$_{ed}$):** requer que seja exercitado pelo menos um caminho que não seja livre de exceção mas que seja livre de definição para a variável definida em um nó n para todo nó e toda aresta possível de ser alcançada a partir de n.

2.2.2 Teste de Software Orientado a Aspectos

A POO revolucionou as técnicas de programação, ajudando no tratamento de complexidades inerentes ao software e possibilitando uma melhor separação de interesses na implementação e no projeto de sistemas (CAPRETZ, 2003). Apesar disso, alguns problemas não foram completamente resolvidos. A Orientação a Objetos não permite a clara separação de certos tipos de requisitos - geralmente não funcionais - que têm sido chamados de interesses transversais (do inglês *crosscutting concerns*) (ELRAD et al., 2001), pois sua implementação tende a se espalhar por diversos módulos do sistema, em vez de ficar localizada em unidades isoladas.

Como uma proposta para resolver esse problema, no final da década de 1990 surgiu a POA (KICZALES et al., 1997), que oferece mecanismos para a construção de aplicações em que a implementação dos interesses transversais fica separada da implementação dos interesses base. A POA é uma abordagem relativamente recente que já motivou a proposição de uma série de abordagens específicas para o teste de aplicações OA. Em geral,

tais abordagens consistem em adaptações de abordagens de testes propostas para outros tipos de aplicações (por exemplo, procedimentais e OO).

Desafios para o Teste de Software OA

Os desafios para o teste de software OA – ou seja, o teste de software desenvolvido sobre o paradigma OA – estão relacionados principalmente ao teste de integração. Considerando o teste de unidade como os adendos e os métodos da aplicação, as técnicas e critérios aplicados tradicionalmente no teste de software procedimental e OO podem ser aplicados sem grandes modificações (FERRARI, 2010).

Segundo Filman e Friedman (2000), em relação ao teste de integração em aplicações procedimentais e OO, consegue-se definir claramente as interfaces entre duas unidades que se comunicam. Em relação à Orientação a Aspectos, um aspecto pode acrescentar funcionalidades ou alterar o comportamento por meio de vários módulos de forma que as declarações que as definem se encontram nos próprios aspectos, e não espalhados pelos módulos da aplicação como na POO. Em POA há um mecanismo de separação de interesses que pode ser feito com base em duas propriedades: a propriedade de inconsciência (do inglês, *obliviousness*), que se refere à liberdade dada aos desenvolvedores para desenvolverem o software preocupando-se apenas com as funcionalidades básicas, nas quais possuem domínio do conhecimento; e a propriedade da quantificação (do inglês, *quantification*), que se refere à possibilidade de se definir a execução de uma ação em uma aplicação a qualquer momento que uma dada condição ocorrer. Os conceitos de quantificação e inconsciência possibilitam a execução dos comportamentos transversais em diversos pontos da execução da aplicação, sem a necessidade da inserção de chamadas explícitas a esses comportamentos.

Para implementar as propriedades de inconsciência e quantificação, a POA faz uso de mecanismos que permitem identificar pontos bem definidos no fluxo de execução de uma aplicação de forma a inserir ou alterar seu comportamento original. Esses pontos são chamados pontos de junção (*join points*). Durante a execução de uma aplicação, ao passo que os pontos de junção são alcançados, um ou mais aspectos podem executar funcionalidades extra ou alterar o comportamento da aplicação original. Essas funcionalidades são implementadas em unidades chamadas adendos, que são construções similares aos métodos de uma classe (FILMAN; FRIEDMAN, 2000).

Em relação ao ponto de junção, quando ele é encontrado, o fluxo de controle é passado para o adendo do aspecto que afeta aquele ponto, retornando ao final da execução. Isso

pode ser comparado com uma chamada de método, mas a dependência é inversa, pois o desenvolvedor do método precisa inserir explicitamente a chamada de método de acordo com a sua necessidade. Já o ponto de junção, são os aspectos os responsáveis por definir em comportamento extra em pontos de junção específicos. Kiczales e Mezini (2005) afirmam que a composição de aplicações OA resulta no surgimento de novas interfaces em diversos módulos do sistema. Essas novas interfaces devem ser consideradas nas abordagens de teste.

Tipos de Defeitos de Software OA

Apesar dos novos mecanismos introduzidos pela POA, foram identificados alguns potenciais defeitos durante os últimos anos (ALEXANDER; BIEMAN; ANDREWS, 2004), e foi possível representar novos desafios para melhorar a qualidade de software. Ferrari et al. (2010) agregam tipos de defeitos de acordo com os elementos e construções de software OA caracterizados por Alexander, Bieman e Andrews (2004), Ceccato, Tonella e Ricca (2005), Mceachen (2005), van Deursen, Marin e Moonen (2005), Lemos et al. (2006), Zhang e Zhao (2007) e Coelho et al. (2008).

A primeira iniciativa para definir uma taxonomia de defeito para software OA foi apresentada por Alexander, Bieman e Andrews (2004):

- Strength incorreto em padrões DCJs - Descritores de Conjuntos (de pontos) de Junção (do inglês, *pointcuts*);
- Precedência de aspecto incorreta;
- Defeito para estabelecer pós-condições;
- Defeito para preservar invariantes de estado;
- Foco incorreto no fluxo de controle;
- Modificações incorretas no controle de dependências.

Ceccato, Tonella e Ricca (2005) adicionaram três novos defeitos:

- Modificações incorretas no fluxo de controle excepcional;
- Defeito devido à declarações intertipo;
- Alterações incorretas em chamadas polimórficas.

Lemos et al. (2006) aprimoraram o tipo de defeito "Strength incorreto em padrões de pointcut" dentro de quatro categorias:

- Seleção de um super conjunto de pontos de junção desejados;
- Seleção de um subconjunto de pontos de junção desejados;
- Seleção de um conjunto de pontos de junção errado, que inclui itens tanto intencionais e não intencionais;
- Seleção de um conjunto de pontos de junção errado, que inclui somente os itens indesejados.

Outros autores como Mceachen (2005), van Deursen, Marin e Moonen (2005), Zhang e Zhao (2007) também definiram tipos de defeitos específicos para POA, mas que se sobrepõem parcialmente à taxonomia de Alexander, Bieman e Andrews (2004). Posteriormente, Coelho et al. (2008) realizaram um estudo exploratório do impacto de aspectos no fluxo de exceções em sistemas OA. Como adição, introduziu um catálogo de nove padrões de *bugs* para manipulação de exceção em AspectJ, divididos em três categorias:

- Aspectos como manipuladores;
- Aspectos como sinalizadores;
- Aspectos como aliviadores de exceção.

Abordagem para Teste Estrutural de Software OA - Lemos et al. (2004, 2005, 2007)

Essa abordagem trata-se da mesma linha de pesquisa realizada por Vincenzi et al. (2006), também para o nível de teste estrutural de unidade, que a partir de informações extraídas dos *bytecodes* Java, definem grafos de fluxo de controle e de dados para os métodos da aplicação. Os trabalhos de Lemos et al. (2004), Lemos, Maldonado e Masiero (2005), Lemos et al. (2007) apresentados a seguir mostram uma abordagem de teste estrutural de unidade de aplicações OA escritas em AspectJ, possibilitando o teste dos aspectos separadamente dos módulos-base da aplicação.

Lemos et al. (2004) definiram o grafo AODU (*Aspect-Oriented Def-Use*) para cada unidade que pertence aos módulos em teste. O grafo AODU estende o tradicional DUG (Def-Use Graph) para incluir os nós transversais, que consistem em nós do gráfico que

envolvem informações de fluxo de controle e fluxo de dados sobre os pontos de junção no código base. Dada uma unidade em teste (por exemplo, um método de classe), os nós transversais indicam os pontos de junção de que a unidade são afetados por *advices*.

Na Figura 2.2 é apresentado um grafo AODU. Os nós pontilhados representam os nós transversais, tendo adicionado aos seus rótulos informações sobre o tipo de adendo que afeta aquele ponto e o nome do aspecto ao qual o adendo pertence. Por exemplo, o nó rotulado como "*33≪afterReturning-telecom.Billing≫*" indica que um adendo do tipo "`after returning`", implementado no aspecto `Billing` do pacote `telecom`, está interceptando esse ponto da execução da aplicação. Ainda, nós negritados representam nós de saída, nós com círculos duplos representam nós de chamada e arestas pontilhadas representam desvios do fluxo de execução para tratadores de exceção (essas últimas não aparecem na Figura 2.2). Para deixar o grafo mais legível, optou-se por não apresentar as definições e usos de variáveis que ocorrem em cada nó do grafo, o que caracterizaria um AODU.

Lemos et al. (2007) definem dois critérios baseados em fluxo de controle tradicionais (todos-nós e todas-arestas) e um critério baseado em fluxo de dados (todos-usos). Considerando C o conjunto (possivelmente vazio) de nós transversais de um AODU, os seguintes critérios são definidos:

- **Todos-Nós-Transversais (Todos-Nós$_c$):** Requer que cada nó c, ($c \in C$) seja exercitado por pelo menos um caso de teste. Ou seja, este critério exige que cada execução do adendo que ocorre na unidade afetada seja alcançada.
- **Todos-Arestas-Transversais (Todos-Arestas$_c$):** Requer que cada aresta transversal, ou seja, cada aresta que tem como nó origem ou destino um nó transversal, seja exercitada por pelo menos um caso de teste.
- **Todos-Usos-Transversais (Todos-Usos$_c$):** Requer que cada par def-uso cujo uso está em um nó transversal seja exercitado por pelo menos um caso de teste.

A abordagem de teste estrutural de unidade (LEMOS et al., 2007) foi estendida para contemplar o teste de integração par-a-par (LEMOS; FRANCHIN; MASIERO, 2009) e o teste de integração baseado no contexto dos pontos de junção capturados por um Descritor de Conjunto de Junção (ou seja, *pointcut*) (LEMOS; MASIERO, 2011). Lemos, Franchin e Masiero (2009) disseram que essa extensão consiste numa abordagem baseada em pares em que os gráficos AODU (*Aspect-Oriented Def-Use*) de duas unidades de comunicação são combinadas em um único gráfico chamado PWDU (*PairWise Def-Use*). Assim como

```
1         public class Call {
2           private Customer caller, receiver;
3           private Vector connections = new Vector();
4
5   0       public Call(Customer caller,
6                   Customer receiver, boolean iM) {
7
8   4         this.caller = caller;
9   4         this.receiver = receiver;
10  4         Connection c;
11  4         if (receiver.localTo(caller)) {
12  33-72        c = new Local(caller, receiver, iM);
13              } else {
14  78-117       c = new LongDistance(caller,receiver,iM);
15              }
16  120       connections.addElement(c);
17  120     }
18          ...
19        } // end class
20
21
22        public aspect Billing {
23          private Customer Connection.payer;
24
25          pointcut createConnection(Customer caller,
26                    Customer receiver, boolean iM) :
27              args(caller, receiver, iM) &&
28              call(Connection+.new(..));
29
30          after(Customer caller, Customer receiver,
31                  boolean iM) returning (Connection c):
32              createConnection(caller,receiver,iM) {
33
34            if (receiver.getPhoneNumber().
35                                    indexOf("0800")==0)
36              c.payer = receiver;
37            else
38              c.payer = caller;
39            c.payer.numPayingCalls += 1;
40          }
41          ...
42        } // end aspect
```

```
 0 aload_0
 1 invokespecial #15 <Method Object()>
 4 aload_0
 ...
27 invokevirtual #30 <Method boolean
   localTo(telecom.Customer)>
30 ifeq 78
33 aload_1
...
52 invokespecial #34 <Method
   Local(telecom.Customer,
   telecom.Customer, boolean)>
...
69 invokevirtual #110 <Method void
   ajc$afterReturning$telecom_Billing$
   1$8a338795(
   telecom.Customer, telecom.Customer,
   boolean, telecom.Connection)>
72 nop
73 astore 4
75 goto 120
78 aload_1
...
97 invokespecial #37 <Method
   LongDistance(telecom.Customer,
   telecom.Customer, boolean)>
...
114 invokevirtual #110 <Method void
   ajc$afterReturning$telecom_Billing$
   1$8a338795(
   telecom.Customer, telecom.Customer,
   boolean, telecom.Connection)>
117 nop
118 astore 4
120 aload_0
121 getfield #20 <Field java.util.
   Vector connections>
124 aload 4
126 invokevirtual #41 <Method void
   addElement(java.lang.Object)>
129 return
```

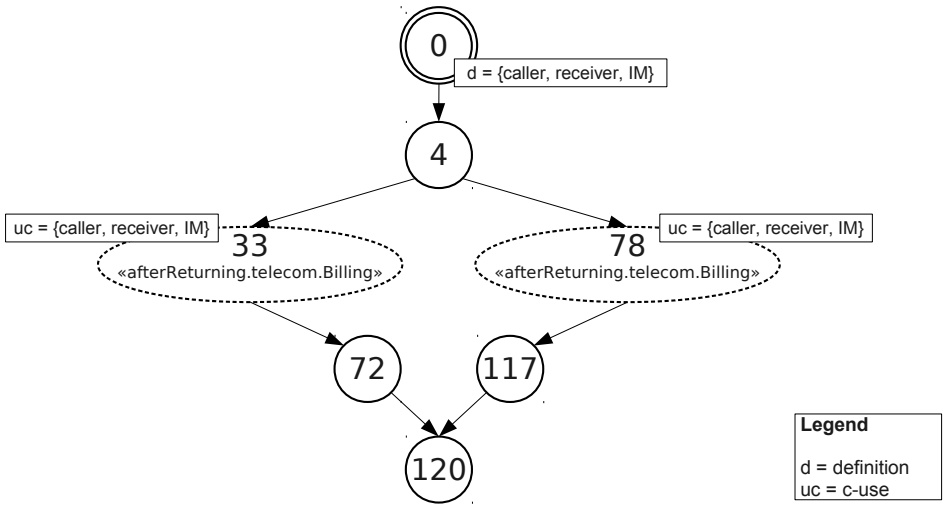

Figura 2.2: Exemplo de um grafo AODU – adaptado do trabalho de (LEMOS et al., 2007).

a abordagem de teste de unidade, ele também depende de análise de *bytecode* Java para permitir a geração de grafos.

2.3 Engenharia de Software Experimental

A experimentação, quando aplicada corretamente, ajuda a construir uma base de conhecimento confiável, pois reduz as incertezas sobre as teorias propostas. Sendo assim, é importante avaliar os experimentos dessas teorias, pois acelera a consolidação de ideias e ajuda a comunidade científica.

Segundo Travassos (2002), quanto maior o controle sobre um experimento, menos recursos são necessários para sua execução e mais rápido o experimento pode ser executado. Por outro lado, quando uma avaliação experimental é executada sem muito rigor, além da difícil replicação, pode colocar a validade das conclusões obtidas em risco.

A literatura define três tipos de experimentos, diferenciando-os em relação ao controle de execução, o controle de medição, o custo de investigação e a facilidade de repetição. Para decidir qual tipo de experimento a ser executado, o pesquisador deve identificar o rigor necessário e os recursos disponíveis para a execução do experimento. A seguir são apresentados os três principais tipos de experimentos (TRAVASSOS, 2002):

Pesquisa de Opinião: conhecida também como *survey*, é utilizada como uma forma de avaliação de uma técnica proposta. A pesquisa de opinião auxilia os pesquisadores a compreender as razões que levaram um grupo de pessoas a utilizar determinada técnica em relação à outra. A principal forma de coleta de dados é por meio de questionários ou entrevistas. É considerado o método menos rigoroso para a realização de experimentos, pois não oferece nenhum controle sobre a execução ou medição dos dados. É um método com baixo custo de aplicação e a quantidade de participantes pode ser grande, aumentando a validade dos resultados estatísticos.

Estudo de Caso: geralmente é utilizado no monitoramento de técnicas e ferramentas. Uma aplicação muito comum é na validação de uma nova tecnologia, em que a tecnologia proposta é comparada com a tecnologia atual em relação aos aspectos de interesse e suas vantagens são detalhadas. O estudo de caso não tem controle sobre a execução do experimento; no entanto, esse método é mais rigoroso do que a pesquisa de opinião, pois a medição das variáveis é controlada. No estudo de caso existem os fatores de confusão, em que mais de um fator pode ocasionar um mesmo efeito, e devido à falta de controle sobre a execução do experimento, não é possível determinar qual a origem do efeito obtido. Estudos de casos devem ser planejados com cuidado para evitar fatores de confusão.

Experimento controlado: geralmente é realizado em laboratório, onde todas as variáveis são controladas sob condições naturais. O objetivo do experimento controlado é manipular uma ou algumas variáveis e manter outras fixas, medindo o efeito do resultado. É o método mais rigoroso a ser aplicado, e permite grande controle sobre a execução e investigação da proposta, além de maior validade e confiança agregada.

Observa-se que embora essa seja uma classificação de tipos de experimentos geralmente empregada no contexto de Engenharia de Software, outras classificações podem ser encontradas. Por exemplo, Zannier, Melnik e Maurer (2006) definem quatro parâmetros:

1. *Tipo de estudo*: é o método de avaliação empírica e pode ser um experimento controlado, um *quasi-experimento*, um estudo de caso, um estudo de caso exploratório, um relato de experiência, uma meta-análise, uma aplicação de exemplo, um survey ou uma discussão.

2. Tipo de amostragem: é o método por meio do qual a amostra foi escolhida. Pode ser de forma aleatória simples, aleatória estratificada, aleatória multi-estágio, conveniência não aleatória, auto-selecionado não aleatória, investigador selecionado não aleatório, cota não aleatória, amostragem de referência não aleatória, intencional não aleatória ou caso crítico não aleatório.

3. População usada/alvo: é a amostra em que a avaliação foi realizada da população abordada.

4. Tipo de avaliação: é quem desenvolveu o assunto em estudo e quem avaliou.

A seguir são apresentados alguns outros conceitos básicos relacionados à experimentação, tendo como base o trabalho de Travassos (2002):

Variáveis: em um experimento, as variáveis são as informações que são manipuladas pelos pesquisadores e podem ser classificadas em dois grupos, dependendo da forma como são obtidas. O relacionamento entre as variáveis é detalhado na Figura 2.3.

1. *Variáveis Independentes (ou fatores):* são referentes à entrada do processo de experimentação e correspondem às causas que podem afetar o resultado do experimento. Cada valor que uma variável independente pode assumir é denominado *tratamento*.

2. *Variáveis Dependentes:* apresentam o efeito causado pelos fatores do experimento, ou seja, são variáveis referentes à saída, e seu valor é denominado *resultado*, que é obtido após a execução do experimento.

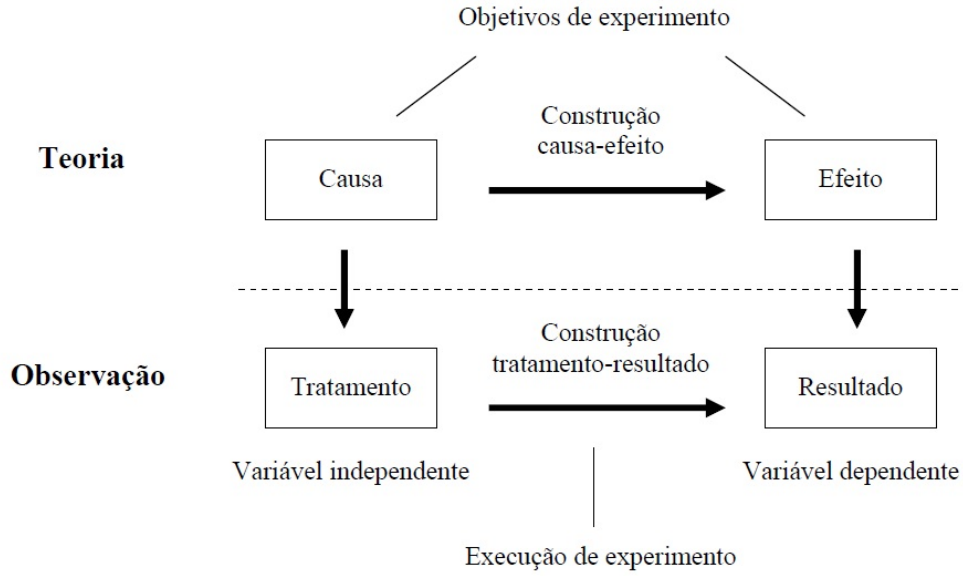

Figura 2.3: Apresentação dos conceitos de um experimento (WOHLIN et al., 2000).

Objetos: são todos os artefatos que podem ser manipulados durante o processo de experimentação. Os **participantes** são os indivíduos selecionados dentre a população para conduzir o experimento. A escolha dos participantes deve ser feita de forma cuidadosa, para que seja selecionada uma amostra representativa da população a ser estudada.

Contexto do experimento: refere-se às condições de execução do experimento. Também inclui a definição de quem serão os responsáveis pela condução do experimento, a natureza do problema tratado e a validade do experimento, apenas para um contexto particular ou generalizado para outros domínios.

O **objetivo do experimento** consiste em avaliar a proposta do autor, identificando os efeitos atingidos com base na execução da proposta do experimento. As propostas são formuladas por meio de **hipóteses**, sendo a hipótese principal chamada de *hipótese nula*, que não oferece indícios de diferenças entre a proposta inicial e seu resultado final. O experimento é realizado, e com a análise do resultado obtido é possível identificar se a hipótese nula pode ser rejeitada ou não. Para isso, alguns princípios devem ser considerados.

2.3.1 Princípios e Validade de Experimentos

Os princípios definidos na Engenharia de Software Experimental devem ser seguidos durante todo o ciclo de vida do experimento, desde o processo de organização, de execução até na fase de apresentação e empacotamento dos resultados. Wohlin et al. (2000)

descrevem os três princípios da Engenharia de Software Experimental:

Aleatoriedade: todos os métodos estatísticos usados para analisar os dados exigem que a observação seja feita a partir das variáveis independentes e aleatórias. É aplicada na alocação dos objetos, dos participantes e na ordem em que os testes são executados. É utilizada para diminuir efeitos indesejados que podem ocorrer devido à ordenação proposital dos artefatos.

Agrupamento: deve ser utilizado em casos nos quais existam uma variável independente que tenha um efeito sobre o resultado, mas esse efeito não é interessante para os pesquisadores. Nesses casos, agrupar variáveis em blocos é uma possível solução para "diluir" efeitos indesejados causados por essas variáveis (TRAVASSOS, 2002).

Balanceamento: procura garantir que todos os tratamentos possuem o mesmo número de participantes com o mesmo nível de conhecimento.

A garantia de validade do experimento também é uma preocupação da Engenharia de Software Experimental. Quatro preocupações são descritas por Travassos (2002) e Wohlin et al. (2000):

Validade de Conclusão: garante que o relacionamento entre o tratamento e os resultados do experimentos foram construídos de maneira correta. Para isso, é preciso analisar como ocorreu o processo desde a coleta das medidas até as conclusões obtidas. Devem ser considerados conceitos como a escolha do teste estatístico, a escolha do tamanho do conjunto de participantes e a confiabilidade das medidas.

Validade Interna: verifica se os resultados obtidos são realmente consequência do tratamento que foi aplicado. Os resultados podem ser influenciados por fatores que não foram medidos ou controlados durante o experimento. O foco da validade interna está nos participantes do experimento.

Validade de Construção: tem como objetivo garantir que a construção do experimento foi realizada de forma correta. Fatores que podem ameaçar a validade de construção podem ser causados tanto pelos participantes quanto pelos pesquisadores.

Validade Externa: definem as condições que limitam a generalização do experimento para a prática industrial.

2.3.2 Processo de Experimentação

Um estudo experimental não é uma tarefa fácil, pois exige esforço necessário para preparar, conduzir e analisar os resultados. Para que o experimento seja conduzido adequadamente, é necessário a definição de um processo. Na Figura 2.4 são apresentadas as cinco fases gerais que podem estar presentes em um processo de experimentação (WOHLIN et al., 2000).

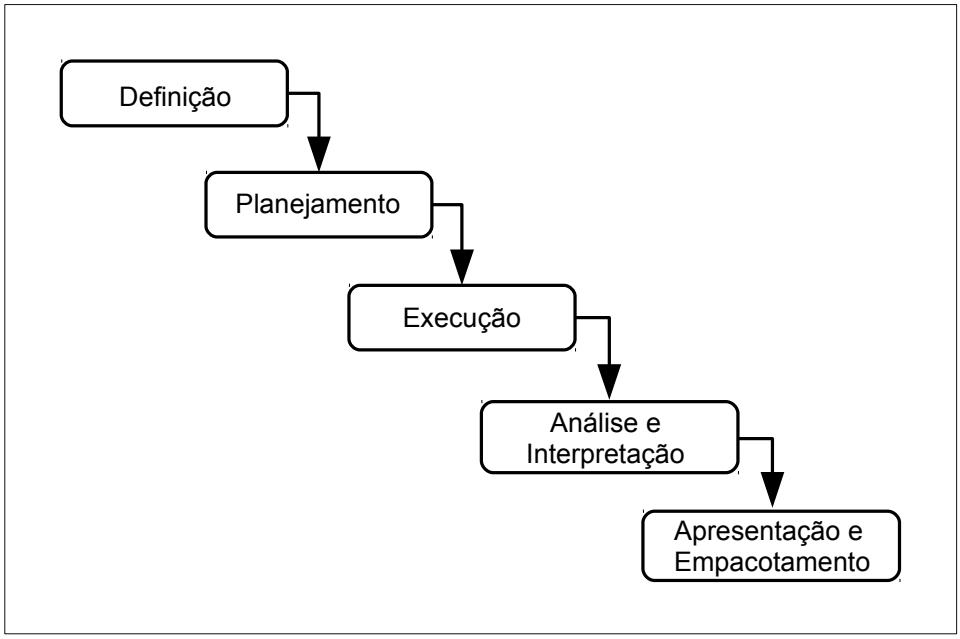

Figura 2.4: Fases gerais de um processo de experimentação (WOHLIN et al., 2000).

Cada fase é descrita a seguir (WOHLIN et al., 2000):

Definição: os pesquisadores descrevem os objetos, o contexto e o escopo do estudo. Nessa fase são descritas as principais características do experimento fornecendo uma base que posteriormente será utilizada para a formulação das hipóteses.

Planejamento: as hipóteses são formuladas de acordo com a relação de causa–efeito descrita pelos pesquisadores. As variáveis que serão analisadas são listadas e as medidas utilizadas para coletá-las são detalhadas, também apresentadas com seus valores. É definido um cronograma do experimento e o modo como os recursos serão alocados e os participantes selecionados.

Execução: para a execução, o fator que pode ser prejudicial ao experimento é o fator humano, pois os participantes devem estar cientes da metodologia utilizada e as atividades que cada um deve realizar. Se os participantes não forem bem conduzidos, há chance de prejudicar o estudo experimental (TRAVASSOS, 2002).

Análise e Interpretação: após os dados obtidos na fase de execução, geralmente precisam de um tratamento para que possam ser analisados experimentalmente. Nessa fase, métodos estatísticos são aplicados com o objetivo de eliminar informações fora da distribuição normal e agrupar os resultados de forma lógica. Com base nas informações obtidas, os resultados são analisados e as conclusões descritas. É importante que sejam documentados a descrição dos métodos estatísticos utilizados, assim como os princípios seguidos para garantir a validade do experimento.

Apresentação e Empacotamento: todas as informações relevantes a respeito da execução do experimento são documentadas. São descritas as hipóteses definidas, as atividades executadas, as dificuldades encontradas, os resultados e experiências realizados, as características dos participantes, o ambiente em que o experimento foi aplicado, entre outros.

2.3.3 Estudos Teóricos e Experimentais de Técnicas e Critérios de Teste

Diversas técnicas e critérios foram propostos para guiar a atividade de teste. Com isso, o testador deve definir qual critério ou combinação de critérios utilizar de acordo com a natureza do sistema, o tempo disponível para teste e o risco de defeitos aceitos pelo cliente. Para ajudar essa decisão, estudos teóricos e experimentais de técnicas e critérios de teste são realizados almejando comparar os critérios existentes. Segundo Delamaro, Maldonado e Jino (2007), um conjunto de teste que satisfaz todos os requisitos de um critério de teste C é chamado "adequado" a C, ou "C-adequado".

De acordo com Weyuker (1984), Rapps e J.Weyuker (1985) e Ntafos (1988), do ponto de vista de **estudos teóricos**, os critérios de teste são avaliados em dois aspectos:

1. **Relação de Inclusão:** estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Nessa hierarquia, um critério de teste C_1 inclui um critério C_2 se, para qualquer aplicação P e qualquer conjunto de testes T_1 C_1-adequado, T_1 for também C_2-adequado, e para algum conjunto de casos de teste T_2 C_2-adequado, T_2 não for C_1-adequado.

2. **Complexidade dos Critérios:** a complexidade de um critério C_1 é definida como o número máximo de casos de teste ou de elementos requeridos, no pior caso.

Em **estudos experimentais**, fatores como custo, eficácia e dificuldade de satisfação

(*strength*) são utilizados para avaliar os critérios de teste, descritos a seguir segundo Wong (1993):

Custo: esforço necessário para que o critério seja usado, o qual pode ser medido pelo número de casos de teste necessários para satisfazer o critério ou por outras métricas dependentes do critério como: o tempo necessário para executar todos os mutantes gerados, ou o tempo gasto para identificar mutantes equivalentes, caminhos e associações não executáveis, construir manualmente os casos de teste e aprender a utilizar as ferramentas de teste.

Eficácia: refere-se à capacidade que um critério possui em detectar um maior número de defeitos em relação a outro.

Dificuldade de satisfação (*strength*): refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito outro critério.

Frankl e Weyuker (1986) apresentam a relação de inclusão entre os critérios de fluxo de dados apresentado a seguir na Figura 2.5, considerando programas sem anomalias de fluxo de dados. Pode-se perceber que o critério Todos-p-Usos inclui o critério Todas-Arestas, ou seja, se um conjunto de casos de teste satisfaz o critério Todos-p-Usos, então esse conjunto também satisfaz o critério Todas-Arestas. Quando não é possível estabelecer essa ordem de inclusão para os dois critérios, pode-se dizer que esses critérios são incomparáveis; é o caso dos critérios Todas-Defs e Todos-p-Usos apresentados na Figura 2.5.

Segundo Weyuker, Weiss e Hamlet (1991), o uso de estudos teóricos para escolha de técnicas e critérios de teste podem apresentar informações de valor limitado em algumas situações práticas, e que o uso de informações probabilísticas obtidas a partir de experimentos poderiam contornar algumas dessas limitações. Nesse contexto, Wong, Mathur e Maldonado (1995) compararam experimentalmente os critérios Análise de Mutantes e Todos-Usos em relação ao custo e ao *strength*. Os resultados obtidos revelaram que o *strength* do critério Análise de Mutantes se mostrou maior do que do critério Todos-Usos, pois o conjunto de casos de teste adequado ao critério Análise de Mutantes foi também adequado ao critério Todos-Usos. Contudo, o conjunto de casos de teste adequado ao critério Todos-Usos não se mostrou adequado para o Teste de Mutação para todas as aplicações avaliadas.

Frankl, Weiss e Hu (1997) também compararam o teste de mutação com o critério Todos-Usos. Foi utilizado um conjunto inicial de nove aplicações implementados em Pascal

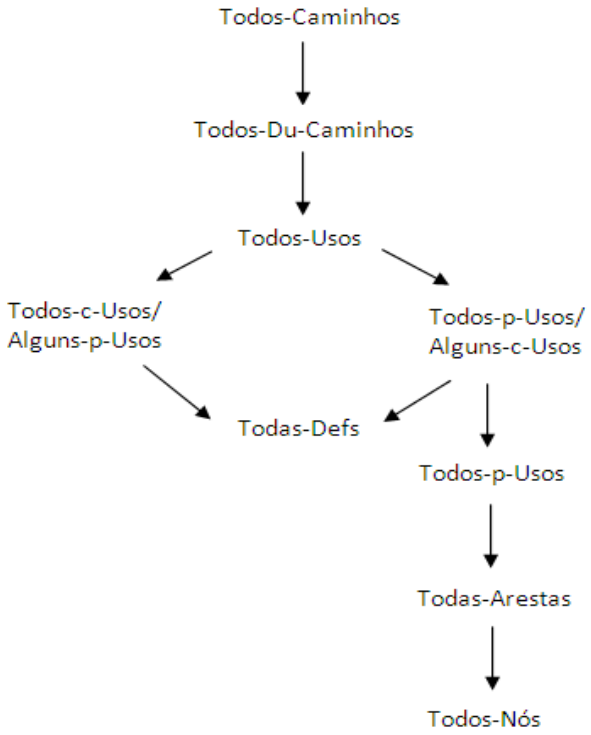

Figura 2.5: Relação de inclusão dos critérios baseados em fluxo de dados e em fluxo de controle (FRANKL; WEYUKER, 1986).

contendo defeitos naturais, mas a ferramenta de teste escolhida para auxiliar o teste de mutação foi desenvolvida para testar aplicações implementadas em Fortran. Por isso, cada aplicação foi traduzida de Pascal para Fortran com o cuidado de que sofresse o menor impacto possível. Os resultados mostraram que o teste de mutação foi mais efetivo em cinco aplicações, o critério Todos-Usos em dois e nos outros dois não foram observadas divergências.

2.3.4 Estudos Experimentais de Teste de Software no contexto Inter-Paradigma de Programação

Com o objetivo de avaliar experimentalmente medidas de custo e satisfação dos critérios de teste, estudos experimentais são aplicados comparando diferentes técnicas e critérios. Critérios que não podem ser comparados de forma teórica podem ter seu custo e eficácia avaliados experimentalmente. Bertolino (2004) disse que os estudos teóricos serviriam para levantar algumas evidências sobre em quais condições uma técnica de teste seria melhor do que outra e os resultados observados a partir de estudos experimentais serviriam para mostrar quando (e em qual contexto) tais condições são satisfeitas.

A seguir serão apresentados alguns trabalhos relacionados ao teste de software entre

paradigmas de programação diferentes que acabam envolvendo diferentes recursos de linguagem de programação. É importante avaliar o impacto de um paradigma específico sobre a atividade de teste, uma vez que alguns defeitos podem estar relacionados ao seu uso.

A Investigação de Prado (2009)

Prado (2009) comparou critérios de teste em diferentes paradigmas: procedimental e OO. Em seu trabalho, o autor apresentou um estudo experimental realizado para caracterizar e avaliar o custo e o *strength* das técnicas funcional e estrutural (baseado em fluxo de controle e em fluxo de dados) com um conjunto de 32 especificações e 64 aplicações. Cada aplicação possuía duas versões, uma escrita em linguagem C e outra em Java, extraídas e adaptadas de um livro didático de Algoritmos e Estrutura de Dados (ZIVIANI, 2005a, 2005b). Para a caracterização de propriedades estatísticas dessas aplicações, 12 métricas de implementação foram avaliadas. Os objetivos com a execução dessa pesquisa foram obter resultados iniciais sobre as questões investigadas, gerar artefatos que serviriam de base para a definição e condução de futuros experimentos e a criação de pacotes de laboratório.

A primeira fase do trabalho consistiu na criação de testes funcionais de acordo com as especificações das aplicações. Essa tarefa respondeu pela maior parte do tempo de condução do experimento, pois seu processo foi manual. As tarefas realizadas foram: a leitura e o correto entendimento da especificação; a definição de classes de equivalência e dos valores limites para cada operação especificada; a escrita dos casos de teste em linguagem natural, definindo as entradas, as saídas e os resultados esperados; a implementação dos casos de teste nas linguagens sob estudo; a verificação das assertivas; e a coleta das medidas geradas.

A fase seguinte contemplou testes estruturais para as aplicações consideradas. Nessa fase, os conjuntos de teste iniciais já estavam estabelecidos. As tarefas realizadas foram: medição do número de elementos requeridos gerados pelos critérios; análise da cobertura dos critérios pelo conjunto de testes funcional-adequado; adequação do conjunto de testes aos critérios verificados; e identificação de elementos requeridos não-executáveis. Os testes estruturais de fluxo de dados demandaram mais tempo que os de fluxo de controle, pois os números de elementos requeridos gerados eram, em média, superiores ao primeiro e a identificação das causas de não cobertura são menos diretas. Ressalta-se que foi necessário adaptar os conjuntos de testes de cada aplicação para a linguagem de cada paradigma

avaliado.

Considerando as 64 aplicações testadas e os conjuntos de testes funcional-adequados, os seguintes resultados em termos de cobertura estrutural foram observados:

- para o critério Todos-Nós, a cobertura média foi 97,65% para as versões procedimentais, e 90,36% para as versões OO;
- para o critério Todas-Arestas, a cobertura média foi 95,16% para as versões procedimentais, e 93,40% para as versões OO;
- para o critério Todos-Usos, a cobertura média foi 92,89% para as versões procedimentais, e 94,37% para as versões OO; e
- para o critério Todos-Potenciais-Usos, a cobertura média foi 90,47% para as versões procedimentais, e 92,28% para as versões OO;

A análise estatística dos dados permitiu concluir que não é possível afirmar que existe uma diferença de custo e *strength* em razão dos paradigmas, tanto para critérios estruturais de fluxo de controle quanto para critérios de fluxo de dados, considerando o domínio de aplicação analisada e o contexto na qual a investigação foi realizada. Isso significa que não foi possível demonstrar a existência dessas diferenças na investigação conduzida.

O trabalho apresentou algumas dificuldades e limitações relacionados às próprias ferramentas de teste. No caso de testes funcionais, a ferramenta CUTe (SOMMERLAD; GRAF, 2007) para testes em aplicações em linguagem C foi a única cuja operação realmente se mostrou estável e semelhante com a ferramenta JUnit para teste em aplicações Java. Para testes estruturais em C foi utilizada a ferramenta Poke-Tool (CHAIM, 1991), que impôs dificuldades com a configuração exigida para instalação e uso, falta de uma interface gráfica, falta de uma funcionalidade para adaptação do arquivo contendo a implementação dos casos de teste, além de não implementar um critério de teste (Potenciais-Usos) para o estudo. Prado (2009) ressaltou ainda a dificuldade de adequar as operações e suas respectivas funcionalidades entre os dois paradigmas.

A Investigação de Campanha (2010)

Em um trabalho similar ao conduzido por Prado (2009), Campanha comparou critérios da técnica funcional e estrutural com o critério Análise de Mutantes, e baseados em conjuntos de testes adequados entre os paradigmas procedimental e OO. Foi utilizado

o mesmo conjunto de 32 especificações (ou seja, 64 aplicações) de domínio de estrutura de dados analisado por Prado (2009). No trabalho de Campanha (2010), o teste de mutação foi aplicado com auxílio das ferramentas Proteum (DELAMARO, 1993) e MuClipse (SMITH; WILLIAMS, 2009). A avaliação do *strength* do teste de mutação em relação aos critérios pertencentes às técnicas funcional e estrutural foi executada a partir dos conjuntos de testes construído em parceria com Prado (2009).

Os testes das aplicações implementadas no paradigma procedimental foram os que gastaram mais tempo para serem aplicados. Além de terem sido testados considerando dois conjuntos de operadores de mutação diferentes (todos os operadores da Proteum e somente os operadores essenciais), foram gerados mais mutantes do que no paradigma OO, o que exigiu mais esforço para analisar cada mutante sobrevivente. Foi verificado o *strength* do teste de mutação considerando o conjunto de todos os operadores da ferramenta Proteum em relação ao conjunto de operadores essenciais definido por Barbosa, Maldonado e Vincenzi (2001). Os operadores essenciais obtiveram um escore bastante alto em relação ao conjunto de todos os operadores da Proteum. Apesar da quantidade de mutantes gerados e a quantidade de mutantes equivalentes terem sido reduzidas significativamente, a quantidade de casos de teste não seguiu a mesma proporção, sendo muito próximas entre os dois conjuntos de operadores.

O número de mutantes gerados nas aplicações implementadas no paradigma OO foi menor do que no paradigma procedimental. Os escores de mutação obtidos pelos conjuntos adequados às técnicas funcional e estrutural foram relativamente altos e em geral maiores do que no paradigma procedimental.

Um resumo dos resultados obtidos é apresentado a seguir, considerando as 64 aplicações testadas e os conjuntos de testes funcional-adequados e estrutural-adequados.

- Considerando somente testes funcionais:
  - escore médio de mutação de 86,80% para as implementações procedimentais, considerando *somente os operadores essenciais*;
  - escore médio de mutação de 90,10% para as implementações procedimentais, considerando *todos os operadores disponíveis na Proteum*;
  - escore médio de mutação de 92,90% para as implementações OO, considerando *todos os operadores disponíveis na MuClipse*;
- Considerando testes funcionais e estruturais:

- escore médio de mutação de 89,60% para as implementações procedimentais, considerando *somente os operadores essenciais*;
- escore médio de mutação de 92,20% para as implementações procedimentais, considerando *todos os operadores disponíveis na Proteum*;
- escore médio de mutação de 94,40% para as implementações OO, considerando *todos os operadores disponíveis na MuClipse*;

O escore de mutação obtido pelos conjuntos de casos de teste adequados ao paradigma OO no paradigma procedimental foi de 86,70% para operadores essenciais e 89,60% para todos os operadores. Em relação aos conjuntos de casos de testes adequados para o paradigma procedimental no paradigma OO, o escore médio foi de 95,80% para operadores essenciais e 95,90% para todos os operadores.

Resultados indicam que tanto o custo quanto o *strength* do teste de mutação é maior em programas procedimentais do que em programas OO e os resultados mostram que os conjuntos de casos de teste adequados aos critérios das técnicas funcional e estrutural no paradigma OO obtiveram, em geral, um maior escore de mutação do que no paradigma procedimental.

O trabalho apresentou algumas dificuldades e limitações como, por exemplo, a limitação da quantidade de ferramentas disponíveis para a execução do teste de mutação, a instabilidade dessas ferramentas, o conjunto limitado de operadores de mutação e a falta de padronização dos operadores de mutação.

2.4 Considerações Finais

Neste capítulo foram apresentados os fundamentos de teste de software e engenharia de software experimental. Em relação ao teste de software, foram apresentados os conceitos básicos e as características e desafios para o teste de software OO e OA. Em relação à engenharia de software experimental, foram apresentados os fundamentos e seu processo de experimentação. Também foram mostrados alguns trabalhos experimentais no contexto inter-paradigma de programação realizados por Prado (2009) e Campanha (2010), os quais inspiraram o presente estudo. Embora existam similaridades entre os objetivos, considerando particularidades em relação aos paradigmas estudados, diferentes métricas foram utilizadas para avaliar os resultados obtidos.

No próximo capítulo são apresentados o planejamento e a execução do presente estudo,

com suas principais etapas do plano elaborado e a forma de execução de cada etapa.

Capítulo 3

Planejamento e Execução do Experimento

3.1 Considerações Iniciais

Este capítulo descreve o planejamento e a execução do estudo cujo objetivo geral foi estabelecido no Capítulo 1. No planejamento são apresentados os principais pontos do plano do trabalho elaborado com suas etapas, a forma de execução de cada etapa, os artefatos e os subprodutos a serem gerados.

Como visto na Subseção 2.3.2, as etapas do processo de experimentação são: Definição, Planejamento, Execução, Análise e Interpretação, e Apresentação e Empacotamento. A etapa de Definição (Seção 3.2) segue a abordagem *Goal Question Metric* (GQM) com seus objetivos, as questões de pesquisa e as métricas utilizadas, além da elaboração da sumarização. A etapa de Planejamento (Seção 3.3), com base na definição do experimento, descreve o contexto, seleciona as variáveis dependentes e independentes, elabora o desenho do experimento, prepara o ambiente de testes e descreve as aplicações utilizadas. Após definido o Planejamento, é realizada a etapa de Execução (Seção 3.4). A Execução é uma etapa muito frágil do experimento, pois possui maior influência do fator humano. É mostrado cada passo realizado no experimento, as dificuldades encontradas e as soluções aplicadas. Uma aplicação do conjunto estudado foi selecionada para exemplificar as atividades realizadas, ilustrando todos os passos do experimento. Por fim, na Seção 3.5 são apresentadas as considerações finais. As etapas de Análise e Interpretação e de Apresentação e Empacotamento serão descritas no Capítulo 4.

3.2 Definição

A definição do estudo foi realizada de acordo com a abordagem GQM definida por Basili, Caldiera e Rombach (1994), a qual consiste em um plano orientado a objetivo baseado em metas explícitas e bem definidas. O GQM contém a informação a respeito de um objetivo e um conjunto de questões, modelos e medidas. As questões definem as informações necessárias para atingir a meta, as medidas definem os dados a serem coletados para responder as questões, enquanto o modelo usa os dados coletados como entrada para gerar a resposta para a questão.

A seguir apresenta-se a definição do estudo deste trabalho seguindo uma estrutura proposta para definir um GQM (BASILI; CALDIERA; ROMBACH, 1994).

3.2.1 Objetivo

O experimento é composto de um objetivo primário, chamado objetivo global, e a partir deste é definido o objetivo secundário:

Objetivo Primário: Este experimento tem como objetivo principal avaliar a qualidade de conjuntos de testes com a aplicação dos critérios todas-instruções e todos-desvios da técnica estrutural a partir de conjuntos de testes adequados ao critério SFT para um paradigma de programação em específico, quando aplicado em outro paradigma. Os paradigmas considerados foram o paradigma Orientado a Objetos (OO) e o paradigma Orientado a Aspectos (OA).

Objetivo Secundário: Avaliar o esforço para adaptar conjuntos de testes funcionais quando ocorre a migração de paradigma. Novamente, foram considerados os paradigmas OO e OA.

3.2.2 Questões

As questões que motivaram este experimento foram:

- Dado um conjunto de testes adequado ao critério SFT para um software implementado em um dos paradigmas considerados (isto é, OO e OA), de acordo com critérios de seleção de casos de teste associados a uma determinada técnica, qual é a qualidade desse conjunto de testes em relação ao software implementado no outro paradigma?

- Qual o custo relacionado ao esforço da adaptação dos conjuntos de testes da aplicação dos critérios na mudança do paradigma OO para o paradigma OA e vice-versa?

3.2.3 Métricas

Para o presente estudo, foram coletadas: (i) métricas convencionais de modificação de código de casos de teste; e (ii) métricas de cobertura estrutural de código. As métricas do grupo (i) foram baseadas em Churn-LOC (NAGAPPAN; BALL, 2005), com o objetivo de comparar conjuntos de testes desenvolvidos e adaptados para ambos os paradigmas considerados[1] relacionados ao número de linhas de código modificadas de uma versão a outra. As métricas do grupo (ii) referem-se a métricas de cobertura estrutural de fluxo de controle (instruções e desvios). As descrições das métricas são apresentadas a seguir:

- ***Total-LOC***: número de linhas de código não comentadas de arquivos de código-fonte, considerando as aplicações em teste.
- ***#Classes***: número de classes da aplicação.
- ***#Aspects***: número de aspectos da aplicação.
- **Total Test Case LOC (*Total-LOC-TC*)**: número de linhas de código não comentadas das classes de teste.
- ***Churn-LOC-TC***: número de linhas de código adicionadas, modificadas e removidas de uma classe de teste considerando a versão anterior com a nova versão. Essa métrica pode ser separada em outras 3 sub-métricas:
  - **Added LOC (*ADD*)** – número de linhas de código adicionadas na nova versão de uma classe de teste;
  - **Modified LOC (*MOD*)** – número de linhas de código modificadas na nova versão de uma classe de teste em comparação com a versão anterior; e
  - **Removed LOC (*REM*)** – número de linhas de código removidas da versão anterior de uma classe de teste em comparação com a nova versão.
- **Total Test Cases (*Total-TC*)**: número de casos de teste criados para uma aplicação.

[1] Os nomes das métricas são apresentados em língua inglesa para manter consistência com um trabalho publicado em evento científico (LEVIN; FERRARI, 2014).

- **Test Requirements:** número de classes de equivalência e valores limite (nesse caso, todos relacionados ao teste funcional).
- **Code Coverage:** cobertura obtida de acordo com os critérios baseados em fluxo de controle *Todas-Instruções* e *Todos-Desvios* da técnica estrutural, a partir de conjuntos de teste adequados ao critério SFT.

3.2.4 Sumarização do GQM

A sumarização do GQM é descrita com objetos de estudo, propósito, foco de qualidade, perspectiva e contexto que são apresentados a seguir.

Objetos do estudo: Os objetos de estudo foram conjuntos de testes aplicados no paradigma OO e conjuntos de testes aplicados no paradigma OA, desenvolvidos de acordo com determinado critério da técnica de teste funcional, comparando-os com o paradigma oposto[2].

Propósito: O propósito é caracterizar e avaliar o esforço para adaptar conjuntos de testes em relação aos paradigmas considerados e a qualidade dos testes resultantes, quando aplicados no paradigma oposto.

Foco de qualidade: O foco de qualidade avaliado é o conjunto de teste, com a aplicação do custo que se refere ao esforço requerido para adaptar os conjunto de casos de teste, além das métricas tradicionais de números de requisitos de teste e cobertura estrutural obtida.

Perspectiva: A perspectiva do experimento é do ponto de vista de pesquisadores em Engenharia de Software.

Contexto: O contexto especifica o ambiente no qual o experimento está sendo realizado: o experimento foi realizado por um pesquisador, sobre um conjuntos de aplicações utilizando as ferramentas JUnit[3] e EclEmma[4] para auxiliar na execução e cálculo de cobertura de código das aplicações nos paradigmas OO e OA.

De acordo com o *template* proposto por Basili, Caldiera e Rombach (1994), a seguir apresenta-se, de forma simplificada, o modelo GQM. A Figura 3.1 é utilizada para ilustrar

[2]Comparar com o paradigma oposto significa selecionar uma aplicação com um determinado paradigma (por exemplo OO) e comparar com a aplicação de outro paradigma (no caso, OA), e vice-versa.

[3]http://www.junit.org/ – último acesso 24/10/2014

[4]http://www.eclemma.org/ – último acesso 24/10/2014

o modelo GQM definido, incluindo o objetivo, as questões de pesquisa e as métricas coletadas para respondê-las.

Analisar aplicações e seus respectivos conjuntos de casos de teste
Para o propósito de avaliar paradigmas de programação
Com respeito à dificuldade de testar aplicações desenvolvidas nesses paradigmas
Do ponto de vista de pesquisadores de Engenharia de Software
No contexto de um trabalho de mestrado em Ciência da Computação da UFS-Car.

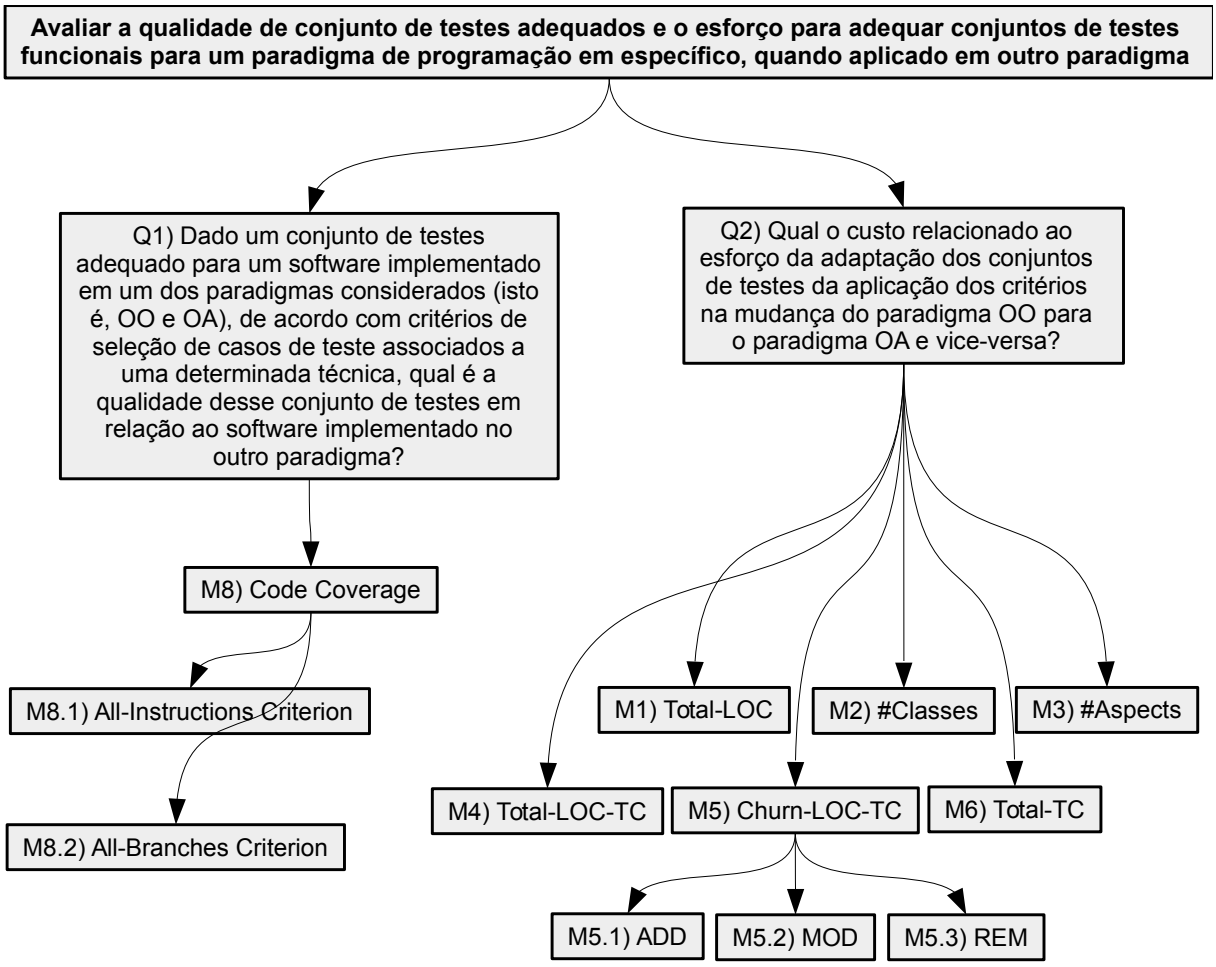

Figura 3.1: Modelo GQM.

A seguir é apresentado o planejamento do experimento, que servirá como base para conduzir e executar o presente estudo.

3.3 Planejamento

De acordo com Travassos (2002), o planejamento de um experimento serve como roteiro para a condução, execução e análise do assunto investigado. Nessa fase acontece a seleção do contexto, seleção das variáveis e o projeto do experimento. O tipo do experimento planejado foi o *quasi-experimento*, devido à falta de alguns pontos a serem considerados em um experimento do tipo controlado, pois algumas características não puderam ser identificadas (ZANNIER; MELNIK; MAURER, 2006); em particular, maior quantidade de participantes no experimento e hipóteses formuladas. Feitas as devidas observações sobre os pontos considerados para o estudo planejado, o resultado dessa fase apresenta o experimento totalmente elaborado e pronto para execução.

3.3.1 Seleção do Contexto

Foi utilizada uma amostra de 13 especificações de aplicações que possuem implementações OO (linguagem Java) e OA (linguagem AspectJ), totalizando 26 aplicações. Essa amostra foi coletada a partir de trabalhos publicados pela comunidade de pesquisa em teste de software e em estudos entre diferentes paradigmas de programação. Detalhes das aplicações serão apresentados na Subseção 3.3.5.

O estudo foi conduzido por um estudante de mestrado, abordando um problema relevante pela comunidade de pesquisa em Engenharia de Software sob o domínio de teste inter-paradigma de programação.

3.3.2 Seleção de Variáveis

Variáveis independentes

Conforme descrito no Capítulo 2, variáveis independentes podem ser manipuladas ou controladas no processo de experimentação (WOHLIN et al., 2000). Foram identificadas neste experimento as seguintes variáveis independentes:

1. Paradigma de programação em que as aplicações foram implementadas.
2. Ferramentas de teste utilizadas.
3. Técnicas e critérios de teste empregados.
4. Casos de testes implementados.

5. Especificação das aplicações.

6. Tamanho e complexidade das aplicações.

A variável a ser considerada no presente estudo é relacionada ao paradigma de programação em que as aplicações foram implementadas, constituindo-se como o único fator de interesse do experimento. Os paradigmas de programação considerados são OO e OA.

Variáveis dependentes

Variáveis dependentes consistem na tentativa de medir o efeito ou os resultados do tratamento da variável independente (WOHLIN et al., 2000). Com a adaptação dos conjuntos de testes para serem executados em aplicações implementadas em ambos os paradigmas considerados, as métricas relacionadas podem afetar as variáveis dependentes. Sendo assim, para este trabalho, as variáveis dependentes identificadas são as métricas descritas na Seção 3.2.3 deste capítulo.

3.3.3 Desenho do Experimento

Com relação às demais variáveis independentes fixas (listadas na Seção 3.3.2), foram aplicados os seguintes princípios de desenho, visando a diminuição da probabilidade de erro experimental dentro do contexto definido:

- Seleção dos critérios de teste: Os critérios foram agrupados em duas categorias: funcional e estrutural fluxo de controle. O critério que compõe a categoria funcional é o critério Teste Funcional Sistemático (LINKMAN; VINCENZI; MALDONADO, 2003). Os critérios selecionados para compor a categoria estrutural fluxo de controle são os critérios Todas-Instruções e Todos-Desvios, pois foram os critérios disponibilizados pela ferramenta de teste estrutural. O desenho escolhido é chamado de desenho cruzado (*crossed design*), pois permite a comparação dos paradigmas em relação a um mesmo tipo de critério.

- Ordem dos testes: Para a definição da ordem dos testes, as aplicações foram divididas em dois grupos. O primeiro grupo, denominado Grupo-A, incluiu metade das aplicações. A outra metade foi incluída em um segundo grupo, o Grupo-B, sendo que ambos os grupos continham implementações funcionalmente equivalentes nos paradigmas OO e OA. As Figuras 3.2, 3.3 e 3.4 mostram o procedimento da ordem dos testes. Vale ressaltar que a sétima aplicação é a aplicação Banco de Questões,

e foram utilizados somente 2 interesses diferentes para cada paradigma (devido ao maior porte de aplicação em comparação com as demais aplicações),. Para o Grupo-A foram selecionados os interesses Segurança e Auditoria, e para o Grupo-B foram selecionados os interesses Controle de Conexão e Tratamento de Exceção[5].

Na Figura 3.2 mostra-se a divisão do total de aplicações em 2 grupos, o Grupo-A e o Grupo-B. Foram selecionadas 6 aplicações nas versões OO e OA para o Grupo-A e outras 6 aplicações nas versões OO e OA para o Grupo-B. A aplicação Banco de Questões foi dividida em 2 interesses para cada grupo e foi realmente coincidente para os dois grupos como a sétima aplicação. Para o estudo foi coletado um total de 13 aplicações.

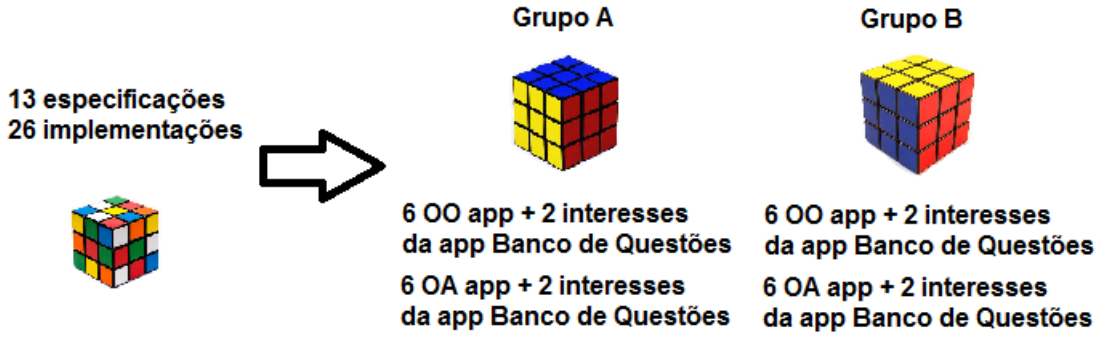

Figura 3.2: As aplicações foram divididas em 2 grupos.

Na Figura 3.3 apresenta-se os conjuntos de testes sendo criados a partir de suas específicas implementações, ou seja, para o Grupo-A tomou-se como base as aplicações OO e foram feitos conjuntos de testes relacionados somente a este paradigma. Para o Grupo-B, tomou-se como base as aplicações OA e foram feitos testes relacionados ao paradigma OA.

[5]Segurança controla o tempo ativo do usuário com o movimento do mouse e a tecla do teclado, Auditoria registra os logs das operações realizadas pelo usuário, Controle de Conexão faz as operações relacionadas a conexão do banco de dados e Tratamento de Exceção registra as exceções causadas pelo sistema.

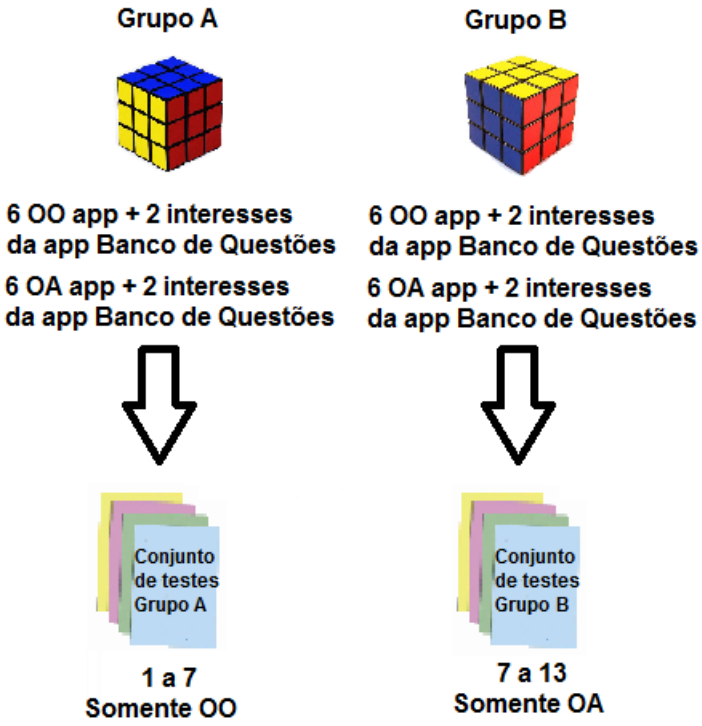

Figura 3.3: Conjunto de testes foram criados para cada grupo dentro do seu paradigma particular.

Na Figura 3.4 mostra-se a adaptação dos conjuntos de teste para a implementação oposta, dentro do seu respectivo grupo. Para o Grupo-A, foram feitas adaptações de código OO para OA, e para o Grupo-B foram feitas adaptações de código OA para OO.

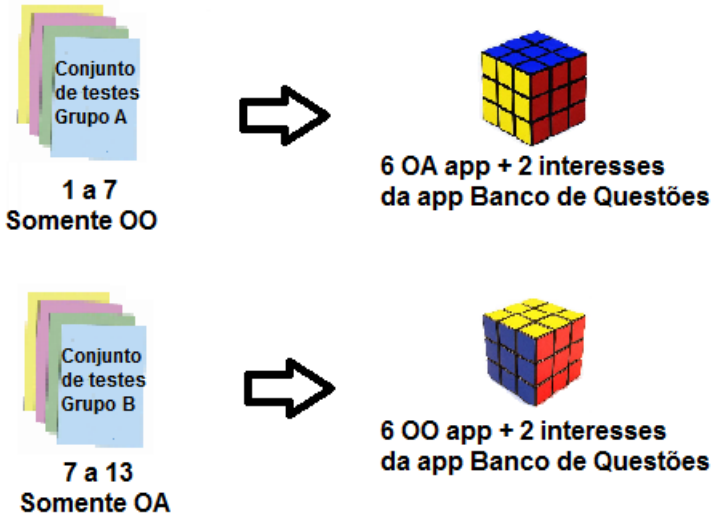

Figura 3.4: Conjunto de testes foram adaptados para as implementações opostas.

Para as demais variáveis independentes fixas não foi aplicado nenhum princípio de desenho. A variável **ferramenta de teste** terá um valor fixo e igual para os dois paradigmas.

Nos paradigmas OO e OA, o valor é determinado pelo *framework* JUnit[6] e a ferramenta EclEmma[7] para aplicação dos critérios funcionais e estruturais, respectivamente.

A **especificação** das aplicações que compõem o conjunto OO é a mesma das aplicações que compõem o conjunto OA. Ambos os conjuntos têm a mesma quantidade de aplicações nos dois paradigmas.

No presente estudo, os casos de teste foram definidos para cada aplicação e para cada tipo de critério. O conjunto de teste gerado para o teste funcional foi reaproveitado para o cálculo da cobertura estrutural das aplicações consideradas.

Os passos da estratégia de teste foram realizados dentro de um grupo em específico, ou seja, Grupo-A (ou Grupo-B), e tomou-se um dos paradigmas como ponto de partida para cada grupo. Com isso, a estratégia de teste foi dividida em seis etapas:

1. O primeiro grupo, denominado Grupo-A incluiu metade das aplicações nas versões OO e OA; e o Grupo-B, a outra metade das aplicações nas versões OO e OA, como visto na Figura 3.2;

2. Foram criados os conjuntos de teste para cada grupo baseado no critério funcional SFT, tendo como ponto de partida as aplicações OO para o Grupo-A e as aplicações OA para o Grupo-B. Esses conjuntos de teste se adequaram tanto para as aplicações OO e OA, pois foram derivados de uma única especificação, como visto na Figura 3.3;

3. Adaptou-se o conjunto de teste para a implementação do paradigma oposto, dentro do seu respectivo grupo, como visto na Figura 3.4.

4. Foram executados os testes estruturais de fluxo de controle para os Grupos A e B, primeiramente nas aplicações base de cada grupo. O Grupo-A teve como ponto de partida o teste estrutural nas aplicações OO e o Grupo-B teve como ponto de partida o teste estrutural nas aplicações OA.

5. Com o conjunto de testes adaptados a implementação do paradigma oposto realizado na etapa 3, foram executados os testes estruturais de fluxo de controle para cobertura de código relacionados a esses conjuntos adaptados dos Grupos A e B, ou seja, o Grupo-A com as aplicações OA e Grupo-B com as aplicações OO.

[6]http://www.junit.org/ – último acesso 24/10/2014
[7]http://www.eclemma.org/ – último acesso 24/10/2014

6. Foram avaliadas as coberturas dos conjuntos de teste para os critérios estruturais de fluxo de controle no Grupo-A (adaptação OO → OA) e no Grupo-B (adaptação OA → OO). As porcentagens de cobertura atingidas foram anotadas no formulário de testes de cada aplicação.

3.3.4 Instrumentação

A instrumentação do trabalho foi elaborada em três etapas: criar objetos do experimento, gerar *guidelines* e instrumentar as medidas.

Tabela 3.1: Descrição da aplicação Chess.

Nome:	Chess (ALVES et al., 2011)
`Requisitos funcionais:`	
`Peão:`	- movimenta-se apenas uma casa para frente. Apenas no movimento inicial o peão pode ser movido duas casas para frente. O peão captura outras peças apenas na diagonal para frente. Caso a casa desejada para o movimento do peão esteja ocupada, o mesmo fica impossibilitado de avançar.
`Rei:`	- movimenta-se apenas uma casa em qualquer direção. O rei captura outras peças deslocando-se apenas uma casa para qualquer direção. Caso a casa desejada para o movimento do rei esteja ocupada, o mesmo fica impossibilitado de avançar.
`Torre:`	- movimenta-se quantas casas disponíveis quiser em linha reta. A torre captura outras peças deslocando-se apenas em linha reta. Caso a casa desejada para o movimento da torre esteja ocupada, o mesmo fica impossibilitado de avançar naquela casa.
`Bispo:`	- movimenta-se quantas casas disponíveis quiser na diagonal. O bispo captura outras peças deslocando-se apenas em diagonal. Caso a casa desejada para o movimento do bispo esteja ocupada, o mesmo fica impossibilitado de avançar naquela casa.
`Rainha:`	- movimenta-se quantas casas disponíveis quiser na diagonal e em linha reta. A rainha captura outras peças deslocando-se apenas em diagonal e linha reta. Caso a casa desejada para o movimento da rainha esteja ocupada, a mesma fica impossibilitado de avançar naquela casa.
`Cavalo:`	- movimenta-se quantas casas disponíveis no formato em L; 2 casas na vertical e 1 na horizontal ou vice-versa. O cavalo captura outras peças deslocando-se apenas em formato em L. Caso a casa desejada para o movimento do cavalo esteja ocupado, o mesmo fica impossibilitado de avançar naquela casa.
`Requisito não funcional:`	
Mensagem de erro:	Mostra uma mensagem de erro caso algum movimento da peça tenha sido realizado de forma inválida.

A etapa de criação dos objetos do experimento consistiu em um conjunto de aplicações às quais foram aplicadas os tratamentos do experimento. Houve a necessidade de preparar

previamente essas aplicações e foi preciso separar as implementações referentes a cada especificação descrita. É apresentado um exemplo na Tabela 3.1 da descrição da aplicação Chess (jogo de xadrez) contendo o nome da aplicação, os requisitos funcionais e não funcionais.

Tabela 3.2: Cabeçalho da planilha de teste funcionais da aplicação Chess.

Nome da aplicação:	Chess
`Fonte:`	Alves, P.; Santos, A; Figueiredo, E; Ferrari, F. C. How do Programmers Learn AOP? - An Exploratory Study of Recurring Mistakes. In: 5th Latin-American Workshop on Aspect-Oriented Software Development (LA-WASP), São Paulo - Brasil, 2011
`Descrição do sistema:`	Jogo de xadrez contendo uma interface gráfica
`Requisitos do sistema:`	As peças do xadrez podem se movimentar e capturar outras peças nas casas de acordo com suas específicas regras. O sistema exibe uma mensagem de erro caso haja um movimento inválido.

Após a descrição das especificações, foi feita uma planilha de testes funcionais para cada aplicação. O cabeçalho dessa planilha, exemplificado na Tabela 3.2, mostra o nome da aplicação, a fonte, a descrição da aplicação e os requisitos da aplicação simplificados. É apresentado um exemplo na Tabela 3.3 de uma especificação da aplicação Chess da planilha de teste funcional. São apresentadas as condições de entrada e saída, as classes válidas e inválidas, e os valores limite.

3.3.5 Preparação

Para que o processo de execução do experimento pudesse ser realizado, foi necessária a preparação do ambiente de testes. Após ser feita a definição e o planejamento para a execução dos testes, o primeiro passo consistiu em planejar as ferramentas para que pudessem ser usada ao longo do experimento.

O Ambiente de Testes

O ambiente construído para a avaliação dos conjuntos de teste nos paradigmas OO e OA foram:

- Sistema operacional Linux, distribuição Ubuntu - versão 13.10;

Tabela 3.3: Exemplo de uma especificação da aplicação Chess da planilha de teste funcional.

`Peça: Bispo`			
Condição de entrada	Classe Válida	Classe Inválida	Valores Limite
Movimentar: número da linha e coluna que se encontra a peça e o número da linha e coluna do destino.	(C1) a casa está livre, o bispo anda qualquer número de casas na diagonal, mas não pode saltar sobre as peças	(I1) casa está ocupada, o bispo vai para a mesma posição na diagonal	(B1) bispo anda 1 posição na diagonal (B2) bispo anda 7 posições na diagonal
Capturar: número da linha e coluna que se encontra a peça e o número da linha e coluna do destino.	(C2) qualquer posição diagonal (se o inimigo ocupa essa casa)		
Condição de saída	Classe Válida	Classe Inválida	
Peça do oponente é capturada	(O1) a peça capturada é removida; bispo ocupa a posição		
Mensagem de erro	(O2) mensagem de erro é mostrada quando o bispo se move para uma posição inválida		

- IDE Eclipse - versão 4.3.1 (Kepler Service Release 1)[8];
- *Framework* JUnit 3.7.2 para a aplicação dos testes funcionais[9];
- Ferramenta de desenvolvimento AspectJ - versão 2.2.4[10];
- EclEmma para a aplicação dos testes estruturais - versão 2.2.1[11];
- Plugin Metrics para coleta das medidas de implementação das aplicações (LOC) - versão 1.3.6[12]; e
- Ferramenta Meld Diff Viewer - versão 1.3.0 para comparação de caracteres entre arquivos[13].

[8] http://www.eclipse.org/ide/ – último acesso 24/10/2014
[9] http://www.junit.org/ – último acesso 24/10/2014
[10] https://www.projects.eclipse.org/projects/tools.ajdt – último acesso 24/10/2014
[11] http://www.eclemma.org/ – último acesso 24/10/2014
[12] http://www.metrics.sourceforge.net/ – último acesso 24/10/2014
[13] http://www.meldmerge.org/ – último acesso 24/10/2014

Artefatos Gerados

Após a instalação e configuração do ambiente, foram gerados alguns artefatos das aplicações utilizadas no experimento na etapa da geração dos *guidelines*, descritos a seguir.

- Descrição da aplicação e suas funcionalidades: nesse documento é descrito como deve ser o funcionamento da aplicação e são detalhados os requisitos;
- Planilha de testes funcionais: documento que descreve a aplicação com o nome da aplicação, fonte de onde a aplicação foi coletada, descrição básica da aplicação, tabela contendo as condições de entrada, classes válidas e inválidas para o particionamento de equivalência e seus respectivos valores limite, e as condições de saída com as classes válidas e inválidas;
- Documento de casos de teste: documento feito para conduzir a execução do teste funcional com o nome da aplicação, casos de teste definidos pela combinação dos critérios Particionamento de Equivalência e Análise do Valor Limite para cada condição de entrada representada anteriormente, requerendo ao menos dois casos de teste para cada partição conforme as diretrizes do critério SFT.
- Diretório das aplicações: diretório contendo as aplicações originais divididas em dois sub-diretórios: Java e AspectJ;
- Diretório das aplicações com os testes: contém 2 sub-diretórios separados em grupos (Grupo-A e Grupo-B), onde o diretório do Grupo-A contém as aplicações testadas no paradigma OA que foram adaptadas de testes de aplicações OO (OO $\rightarrow$ OA), e o diretório do Grupo-B são as aplicações testadas no paradigma OO que foram adaptadas de testes de aplicações OA (OA $\rightarrow$ OO). Há um outro diretório que contém os testes da aplicação Banco de Questões, onde se encontra os aspectos das aplicações OA com a sua respectiva aplicação OO, e são separadas novamente em 2 grupos (Grupo-A e Grupo-B), onde no Grupo-A permanece os testes utilizando os aspectos da versão OA que foram adaptados da versão OO (OO $\rightarrow$ OA), e no Grupo-B permanece os testes utilizando as aplicações em Java da versão OO adaptadas da versão OA utilizando aspectos (OA $\rightarrow$ OO).
- Documento de cobertura estrutural das aplicações: documento para registrar as porcentagens de código atingidas pelo teste estrutural. Contém o nome da aplicação, o total de instruções e desvios (somente para as funcionalidades selecionadas em

ambos os paradigmas), instruções e desvios cobertos e faltantes de cada aplicação, e a porcentagem de cobertura total de instruções e desvios.

- Documento de medidas de implementação: artefato gerado na etapa de instrumentação das medidas para observar a diferença ou a igualdade de LOC entre os testes em ambos os paradigmas, tanto para OO → OA como para OA → OO. É exibido na Tabela 3.4 um exemplo na aplicação Chess para o esforço na migração dos testes de OO para OA desse documento.

Timing.aj

```
1  //Copyright (c) Xerox Corporation 1998-2002.  All rights reserved.
2
3  package telecomAO;
4
5  public aspect Timing {
6
7      public long Customer.totalConnectTime = 0;
8
9      public long getTotalConnectTime(Customer cust) {
10         return cust.totalConnectTime;
11     }
12
13     private Timer Connection.timer = new Timer();
14     public Timer getTimer(Connection conn) { return conn.timer; }
15
16     after (Connection c): target(c) && call(void Connection.complete()) {
17         getTimer(c).start();
18     }
19
20     pointcut endTiming(Connection c): target(c) &&
21         call(void Connection.drop());
22
23     after(Connection c): endTiming(c) {
24         getTimer(c).stop();
25         c.getCaller().totalConnectTime += getTimer(c).getTime();
26         c.getReceiver().totalConnectTime += getTimer(c).getTime();
27     }
28 }
```

Figura 3.5: Um exemplo de requisitos não executáveis que são gerados pela ferramenta EclEmma na aplicação Telecom - Construtor default do bytecode

Na Figura 3.5 é mostrado o aspecto Timing que implementa o interesse de cronometragem entre chamadas telefônicas da aplicação Telecom, e as linhas de código cobertas pelos testes foram as linhas 10, 14, 17, 18, 24, 25, 26 e 27. Com a execução dos testes, a ferramenta Eclemma gerou alguns requisitos que foram desconsiderados pela inspeção manual de código, pois quando os aspectos são instrumentados, a ferramenta pode gerar alguns requisitos não executáveis, como visto nas linhas 1 e 5, o que não representa, de fato, um requisito existente a ser coberto.

A Tabela 3.4 é definida pelos campos nome da aplicação, LOC de toda a aplicação (LOC código fonte), LOC's pertinentes do código fonte testado (LOC relevante fonte),

Tabela 3.4: Exemplo de documento de medidas de implementação.

Nome da aplicação	LOC código fonte	LOC relevante fonte	Nome classe teste	TOTAL LOC TC	LOC SetUp e TearDown
Chess (base: OO)	1155	464	BishopTest	45	2
TOTAL OO	1155	464		45	2
TOTAL OA	945	773		47	2

Nome TC	LOC TC individual	LOC ADD	LOC REM	LOC MOD	LOC TC adaptado
testBishopMovement_C1	4	0	0	0	4
testBishopMovement_C1	8	0	0	0	8
testBishopMovement_C2	6	0	0	0	6
testBishopMovement_I1	5	2	0	1	7
TOTAL OO	23				
TOTAL OA		2	0	1	25

nome das classes de teste, a quantidade de LOC para cada classe de teste (TOTAL LOC TC), LOC SetUp (quantidade de LOC que será incluída no começo da execução do caso de teste) e TearDown (quantidade de LOC que será incluída antes do final da execução do caso de teste) de cada classe de teste (LOC SetUp e TearDown), nome do caso de teste seguido por "C ou I + *número*" onde C é uma classe válida, I é uma classe inválida e *número* é o número do caso de teste definido na planilha de testes funcionais (Nome TC), LOC de cada caso de teste (LOC TC individual), LOC adicionado após a adaptação de paradigma (LOC ADD), LOC removido (LOC REM), LOC modificado (LOC MOD) e LOC total adaptado (LOC TC adaptado).

Descrição das Aplicações

Como já mencionado, as aplicações foram coletadas de trabalhos publicados pela comunidade de pesquisa em teste de software com aplicações de mesma especificação para ambos os paradigmas (isto é, OO e OA), com implementações do mesmo conjunto de funcionalidades. A amostra considerada para o estudo é composta por 13 aplicações de cada paradigma.

Foi coletado um conjunto de aplicações de *Design Patterns* (DP) nomeadas de ShowWindow, AbstractObject, ExpressionInterpreter, ObjectRequest, SensitiveCharacter e RecordValue

Tabela 3.5: Aplicações que foram utilizadas no estudo e sua descrição.

Nome da aplicação Grupo-A	Descrição	Total LOC OO / OA	#Classes	#Classes / #Aspects
1. ShowWindow (DP)	Aplicação que Implementa o design pattern Abstract Factory para a criação de uma interface inicial que permite que o usuário escolha uma factory e gera um novo GUI com os elementos que a respectiva factory fornece.	90 / 97	4	4 / 1
2. Boolean	Representa fórmulas booleanas e as imprime em dois estilos diferentes, com termos AND, OR, XOR, NOT e variáveis.	301 / 316	12	10 / 2
3. AbstractObject (DP)	Desacopla uma abstração de sua implementação para que os dois objetos possam variar independentemente, implementando o design pattern Bridge.	76 / 82	6	6 / 1
4. Chess	Jogo de xadrez contendo uma GUI.	1155 / 945	13	13 / 1
5. ExpressionInterpreter (DP)	Implementa o design pattern Interpreter como um intérprete para uma linguagem de expressões booleanas.	118 / 126	8	8 / 1
6. VendingMachine	Simula uma máquina na qual o cliente insere moedas, a fim de obter bebidas.	209 / 245	9	9 / 1
7. Banco de Questões	Gerencia coleções de questões para provas construídas por professores.	6447 / 6479	27	27 / 5
TOTAL		**8396 / 8290**	**79**	**77 / 12**
Nome da aplicação Grupo-B	**Descrição**	**Total LOC OO / OA**	**#Classes**	**#Classes / #Aspects**
7. Banco de Questões	Gerencia coleções de questões para provas construídas por professores.	6447 / 6479	27	27 / 5
8. ATM-log	Gerencia conta de banco.	496 / 519	12	11 / 1
9. ObjectRequest (DP)	Implementa o design pattern ChainOfResponsability para a criação de uma interface GUI para representar a relação entre os objetos para processar várias solicitações diferentes (ou seja, para recuperar objetos pelo mecanismo de herança).	96 / 150	5	5 / 2
10. SensitiveCharacter (DP)	Gera caracteres em letras maiúsculas e minúsculas e mostra na tela de acordo com uma variedade de opções, implementando o design pattern Flyweight.	44 / 61	4	4 / 2
11. RecordValue (DP)	Grava qualquer valor na memória e recupera esse valor. O aplicativo confere se este valor dado pelo usuário está ou não na memória, implementando o design pattern Memento.	29 / 64	2	3 / 2
12. ShopSystem	Simula operações de comércio eletrônico.	360 / 381	10	8 / 8
13. Telecom	Calcula e relata as taxas e duração de chamadas de telefone (chamadas locais e interurbanas).	186 / 197	8	8 / 2
TOTAL		**7658 / 7581**	**68**	**66 / 22**

extraídas do trabalho de Hannemann e Kiczales (2002) e as demais aplicações ShopSystem (BARTSCH, 2007), Boolean (PRECHELT et al., 2001), Banco de Questões (CHA-

GAS; OLIVEIRA, 2009), ATM-log (ALVES et al., 2011), Chess (ALVES et al., 2011), Telecom (The Eclipse Foundation, 2013) e VendingMachine (LIU; CHANG, 2008).

É apresentada na Tabela 3.5 uma breve descrição das aplicações coletadas com o nome da aplicação, descrição de funcionalidade, total de linhas de código das aplicações OO e OA, e a quantidade de classes para aplicações OO e a quantidade de classes e aspectos das aplicações OA.

3.4 Execução

A execução foi a etapa que demandou mais tempo na condução do estudo. Os testes funcionais responderam pela maior parte do tempo de condução, tendo em vista que todo o processo é manual, sendo apenas a verificação das assertivas auxiliadas pelas ferramentas. Dentre as tarefas exigidas para a realização dos testes funcionais: a criação de um documento das especificações; o correto entendimento da especificação; a definição de classes de equivalência e dos valores limites para cada operação especificada; a escrita dos casos de teste em linguagem natural, definindo-se as entradas, saídas e resultados esperados; a implementação dos casos de teste nas linguagens sob estudo; a verificação das assertivas; e a coleta das medidas geradas. Para a criação dos conjuntos de testes seguindo o critério SFT, houve uma colaboração com outro pesquisador do grupo de pesquisas.

O procedimento para a criação dos conjuntos de testes é apresentado a seguir.

3.4.1 Criação dos Conjuntos de Testes

Para realizar o teste funcional, foi criado um documento com os casos de teste relacionados à planilha de teste funcional, juntamente com o código de teste. Exemplos são exibidos na Figura 3.6 e na Tabela 3.6, sendo descritos a seguir.

Na Figura 3.6 é mostrado um exemplo de como foram feitos os testes utilizando-se o *framework* JUnit. No caso de teste (testBishopMovement_C1()) é feita a verificação de uma funcionalidade do movimento de uma peça de xadrez da aplicação Chess e ao final é comparado se a posição final é legal para a peça Bispo. Após a execução do teste, o resultado é informado na planilha de testes funcionais (Tabela 3.6).

```
1  public void testBishopMovement_C1(){
2          startRow = 6; startColumn = 5;
3          desRow = 4; desColumn = 3;
4
5          assertTrue(bishop.legalMove(startRow, startColumn, desRow, desColumn,
6          cellMatrix.getPlayerMatrix()));
7
8          startRow = 6; startColumn = 5;
9          desRow = 4; desColumn = 7;
10
11         assertTrue(bishop.legalMove(startRow, startColumn, desRow, desColumn,
12         cellMatrix.getPlayerMatrix()));
13 }
```

Figura 3.6: Exemplo de caso de teste.

Tabela 3.6: Exemplo de algumas funcionalidades da aplicação Chess do documento de caso de teste.

Funcionalidade	Caso de teste <dado de teste, resultado esperado>	Resultado
`BishopTest`		
testBishopMovement_C1	<bishop.legalMove(sr, sc, dr, dc, getPlayerMatrix(), true>	true
testBishopMovement_C2	<bishop.legalMove(sr, sc, dr, dc, getPlayerMatrix(), true>	true
testBishopMovement_C3	<bishop.legalMove(sr, sc, dr, dc, getPlayerMatrix(), true>	true
testBishopMovement_I1	<"mensagem de erro", bishop.strErrorMsg>	bishop.strErrorMsg

A Tabela 3.6 mostra um exemplo para identificar os testes de algumas funcionalidades da aplicação Chess com os parâmetros dado de teste, resultado esperado e o resultado do teste.

3.4.2 Procedimentos para Coleta de Métricas de LOC

Após a tarefa de criação dos casos de teste de cada grupo de aplicações, os testes tiveram que ser adaptados para o seu paradigma oposto com as implementações equivalentes. Para a adaptação desses testes, algumas aplicações necessitaram de modificações no código em teste devido a diferentes recursos de linguagem. Por exemplo, para recuperar um objeto dentro de um aspecto em uma aplicação OA, pode-se utilizar o método estático aspectOf(), mas tal recurso não existe na linguagem Java. Com isso, é necessária a adaptação para o código OO com os recursos disponíveis da linguagem para que os conjuntos de testes sejam adequados, mantendo a equivalência funcional. Isto foi feito para todas aplicações, tanto para o Grupo-A (OO $\rightarrow$ OA) quanto para o Grupo-B (OA $\rightarrow$ OO).

Aplicação das métricas nas classes de testes

Em geral, para a conversão dos casos de teste entre aplicações OO e OA, houve uma maior semelhança sintática entre os casos de teste. Quando a conversão dos casos de teste não puderam seguir a semelhança estática devido aos recursos de linguagem disponíveis em cada paradigma, a semelhança semântica do caso de teste foi preservada. Os valores das entradas deveriam ser iguais sempre que possível. Caso não fossem, elas deveriam ser minimamente correspondentes, ou seja, deveriam ser suficientes para que executassem o mesmo comportamento esperado da operação.

Para a coleta de métricas de LOC foi utilizado o Plugin Metrics[14] e a ferramenta Meld Diff Viewer[15]. O Plugin Metrics trata-se de um plugin da IDE Eclipse para contar linhas de código em diferentes níveis (método, classe, aplicação, etc.) e a Meld Diff Viewer trata-se de uma ferramenta para comparação de caracteres entre arquivos, com o objetivo de comparar e destacar as diferenças entre eles.

Na Figura 3.7 é apresentado um exemplo de como um caso de teste da aplicação Chess foi adaptado da implementação OO para OA, e na Figura 3.8 é mostrado um exemplo de como um caso de teste da aplicação ObjectRequest foi migrado da implementação OA para OO. As linhas modificadas estão destacadas em negrito.

Exemplo de um caso de teste - Aplicação Chess OO:

```
1  public void testBishopMovement_nSquaresDiagonal_I1(){
2          startRow = 7; startColumn = 5;
3          desRow = 5; desColumn = 3;
4
5          assertFalse(bishop.legalMove(startRow, startColumn, desRow,
6          desColumn, cellMatrix.getPlayerMatrix() ));
7          assertEquals("Bishop can only move along diagonal lines", bishop.strErrorMsg);
8  }
```

Exemplo de um caso de teste - Aplicação Chess OA:

```
1  public void testBishopMovement_nSquaresDiagonal_I1(){
2          startRow = 7; startColumn = 5;
3          desRow = 5; desColumn = 3;
4
5          assertFalse(bishop.legalMove(startRow, startColumn, desRow,
6          desColumn, cellMatrix.getPlayerMatrix()));
7          ErrorMsg em = ErrorMsg.aspectOf();
8          String output = em.getErrorMsg();
9          assertEquals("Bishop can only move along diagonal lines", output);
10 }
```

Figura 3.7: Exemplo de um caso de teste para a aplicação Chess.

O caso de teste da aplicação Chess apresentado na Figura 3.7 faz a verificação se um

[14]http://www.metrics.sourceforge.net/ – último acesso 24/10/2014
[15]http://www.meldmerge.org/ – último acesso 24/10/2014

movimento de uma peça de xadrez (de um bispo, no caso), é legal ou não conforme as regras básicas de um jogo de xadrez. Na assertiva são preenchidos os dados de teste e o resultado esperado, ou seja, o objeto passado pelo testador como parâmetro (a mensagem de erro, no caso) com base na especificação e comparado com a respectiva mensagem esperada a ser exibida com a recuperação do objeto corrente no software. Se o movimento da peça for ilegal, as mensagens de erro devem ser equivalentes e então o teste é executado com sucesso.

Para a migração de OO para OA, considerando o caso de teste nomeado testBishop-Movement_nSquaresDiagonal_I1 da Figura 3.7, as linhas 7 e 8 foram adicionadas na ordem para extrair a informação que é encapsulada no aspecto `ErrorMsg`. Além disso, a linha 9 foi modificada quando comparada ao mesmo caso de teste do Chess OO. Nesse exemplo, a contagem da métrica *ADD* seria 2 e a contagem da métrica *MOD* seria 1, respectivamente. Essa análise foi seguida também para a métrica *REM* e outras métricas coletadas no estudo.

Na Figura 3.8 apresenta-se um exemplo de caso de teste da aplicação ObjectRequest de como foi adaptado da versão OA para OO. O caso de teste de nome testShiftMask_C1 faz a verificação se uma interface é exibida corretamente com as devidas teclas do teclado pressionadas. O presente exemplo mostra a interface quando a tecla shift é pressionada.

Para a adaptação de OA para OO, as linhas 7 e 8 do caso de teste implementado em OA foram removidas para se adaptarem à implementação OO e as linhas 4, 5, 16 e 21 do caso de teste implementado em OO foram modificadas. O método estático aspectOf() não foi utilizado no caso de teste da implementação OO devido à inexistência desse recurso na linguagem Java. Nesse exemplo, a contagem da métrica *REM* seria 2 e a contagem da métrica *MOD* seria 4. Novamente, essa análise foi seguida também para a métrica *ADD* e outras métricas coletadas no estudo.

A coleta das métricas foi realizada para o Grupo-A como base nas aplicações OO para OA e para o Grupo-B como base nas aplicações OA para OO de forma a seguir:

1. Elaboração de um conjunto de teste com base nas aplicações de seu respectivo grupo (Grupo-A com base nas aplicações OO e Grupo-B com base nas aplicações OA);

2. Adaptação do conjunto de teste elaborado na etapa anterior para a implementação oposta;

3. Comparação entre os caracteres dos casos de testes em especificações OO e OA da mesma aplicação utilizando a ferramenta Meld Diff Viewer;

Exemplo de um caso de teste - Aplicação ObjectRequest OA:

```
1  public void testShiftMask_C1(){
2
3          Frame frame = new Frame("Chain of Responsibility pattern example");
4          Panel panel = new Panel();
5          Button button = new Button("Click here");
6
7          ClickChain.aspectOf().setSuccessor(button, panel);
8          ClickChain.aspectOf().setSuccessor(panel, frame);
9
10         frame.getContentPane().add(panel);
11         panel.add(button);
12
13         PrintStream origOut = System.out;
14         ByteArrayOutputStream allOutput = new ByteArrayOutputStream();
15         PrintStream out = new PrintStream(allOutput);
16         System.setOut(out);
17
18         ActionEvent ae = new ActionEvent(button, 1001, button.getActionCommand(), 1);
19         button.doClick(new Click(ae));
20
21         String saida = allOutput.toString().trim();
22         System.setOut(origOut);
23
24         assertEquals("Button is asked to accept the request. Button is handling the event.", saida);
25 }
```

Exemplo de um caso de teste - Aplicação ObjectRequest OO:

```
1  public void testShiftMask_C1(){
2
3          Frame frame = new Frame("Chain of Responsibility pattern example");
4          Panel panel = new Panel(frame);
5          Button button = new Button("Click here", panel);
6
7          frame.getContentPane().add(panel);
8          panel.add(button);
9
10         PrintStream origOut = System.out;
11         ByteArrayOutputStream allOutput = new ByteArrayOutputStream();
12         PrintStream out = new PrintStream(allOutput);
13         System.setOut(out);
14
15         ActionEvent ae = new ActionEvent(button, 1001, button.getActionCommand(), 1);
16         button.handleClick(new Click(ae));
17
18         String saida = allOutput.toString().trim();
19         System.setOut(origOut);
20
21         assertEquals("Button is asked to handle the request. Button handles the request.", saida);
22 }
```

Figura 3.8: Exemplo de um caso de teste para a aplicação ObjectRequest.

4. Comparação entre a quantidade de linhas dos testes em ambas as especificações utilizando o Plugin Metrics;

5. Anotação das linhas de código de casos de teste adicionadas, removidas e modificadas tendo como base as aplicações de ambos os paradigmas.

3.4.3 Procedimentos para Coleta de Métricas de Cobertura

Para a coleta de métricas de cobertura foi utilizada a ferramenta EclEmma[16]. Trata-se de uma ferramenta para computar a cobertura estrutural de código Java utilizada com a IDE Eclipse. Mesmo que a instrumentação dos *bytecodes* em Java seja rápida e consideravelmente confiável, algumas compilações de *bytecodes* a partir de código OA podem produzir resultados sem total confiança em relação à condições implícitas que podem ser inseridas ou omitidas pela sua compilação.

Quando o compilador padrão do AspectJ ajc[17] identifica os possíveis pontos de junção na aplicação, alguns deles só podem ser resolvidos em tempo de execução, com isso, são adicionados alguns *resíduos* no *bytecode* para realizar as checagens dinâmicas necessárias. Um exemplo de *resíduo* pode ser uma instrução condicional if que verifica o tipo de um determinado parâmetro e que é colocada antes da chamada de método correspondente ao adendo, quando a execução do adendo depende do tipo de parâmetro (HILSDALE; HUGUNIN, 2004)

Tendo-se em vista esses problemas, foi realizada uma tarefa adicional para verificação de código em todas as aplicações OA na forma de depuração, na qual o pesquisador teve o trabalho de executar passo-a-passo a aplicação para garantir maior precisão nas coberturas computadas pela ferramenta EclEmma, verificando assim se o código fonte estava sendo realmente coberto ou não pelo teste e analisando as possíveis entradas de resíduos e requisitos incorretos gerados pela ferramenta.

Na etapa de teste estrutural, foram aproveitados os conjuntos de testes pelo critério SFT da técnica funcional que já estavam estabelecidos. As tarefas exigidas nessa etapa foram: análise de cobertura dos critérios Todas-Instruções e Todos-Desvios da técnica estrutural na aplicação do paradigma corrente e na aplicação do paradigma oposto.

A ferramenta EclEmma computa valores de instruções e desvios cobertos ou não pelo teste em questão. Depois que o teste é escrito, é preciso executá-lo no modo de cobertura de código para teste estrutural. Segue um exemplo (Figura 3.9) da utilização da ferramenta com a aplicação Chess na versão OO.

[16]http://www.eclemma.org/ – último acesso 24/10/2014
[17]https://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html – último acesso 24/10/2014

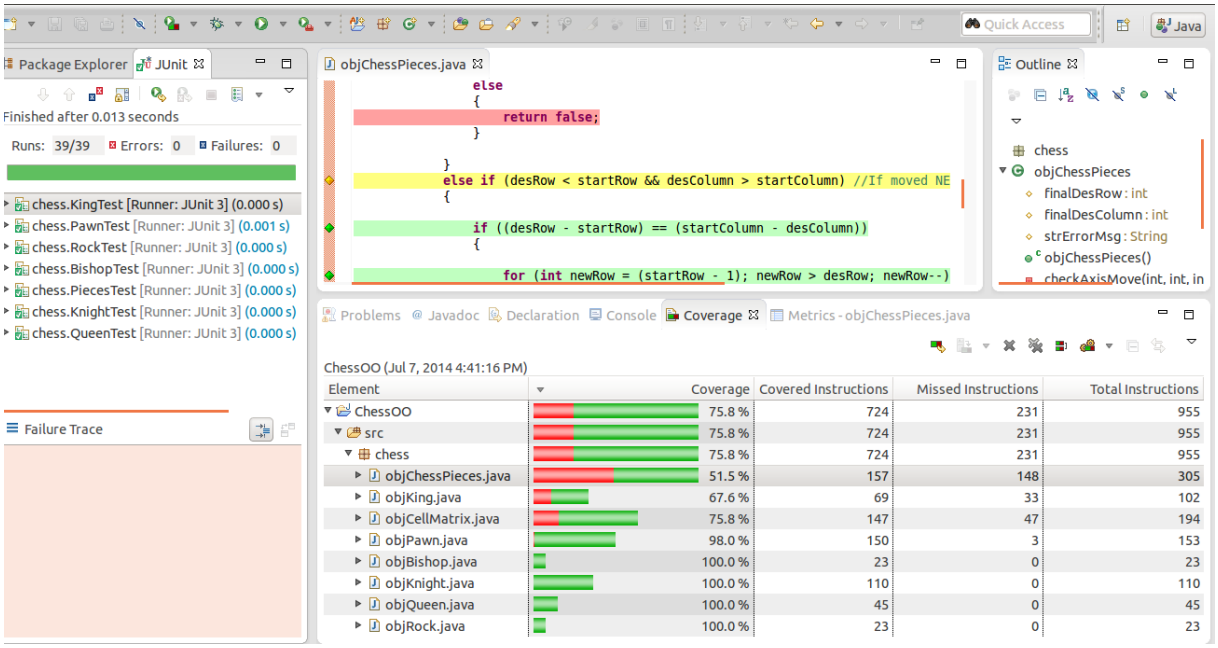

Figura 3.9: Aplicação Chess em teste estrutural.

Aplicação do Critério Todas-Instruções e Todos-Desvios

Após a adaptação dos conjuntos de testes para todas as aplicações dos dois paradigmas, iniciou-se a aplicação dos critérios Todas-Instruções e Todos-Desvios. Devido a criação do conjunto de casos de teste para a técnica funcional com outro estudante de mestrado, houve a possibilidade das aplicações serem testadas de forma aleatória, sendo que cada estudante testou aplicações em ordens diferentes. As aplicações foram divididas em 2 grupos entre os estudantes para que cada um fizesse os testes funcionais primeiramente em um paradigma em específico para depois adaptá-los no paradigma oposto, invertendo a ordem do paradigma realizados inicialmente para cada estudante.

Para a aplicação dos critérios Todas-Instruções e Todos-Desvios foram utilizados os mesmos conjuntos de teste iniciais feitos para o teste funcional, e em seguida foram feitos os seguintes passos:

1. Aplicação do critério Todas-Instruções e Todos-Desvios nas aplicações OO para o Grupo-A (ou no sentido contrário, seguindo de aplicações OA para o Grupo-B) com o auxílio da ferramenta EclEmma;

2. Aplicação do critério Todas-Instruções e Todos-Desvios nas aplicações OA para o Grupo-B (ou em aplicações OO para o Grupo-A) - EclEmma;

3. Inspeção de código OA coberto de forma manual sob o critério Todas-Instruções e

Todos-Desvios (depuração).

Os resultados de instruções e desvios gerados[18] pela ferramenta EclEmma da aplicação Banco de Questões entre os interesses implementados em OO e OA serviram para avaliar e comparar os resultados desses tipos de critério no mesmo contexto sob diferentes portes de sistemas.

3.5 Considerações Finais

Neste capítulo foram apresentados o Planejamento e a Execução do Experimento. Para o planejamento foram apresentados a definição do estudo de acordo com a abordagem GQM, a seleção do contexto, a seleção de variáveis, o projeto do experimento, a preparação do ambiente de testes e descrição das aplicações utilizadas. Para a execução foram apresentados o procedimento para a criação dos conjuntos de testes para o critério SFT da técnica funcional com exemplos de códigos de teste, os procedimentos utilizados para coletar as métricas de linhas de código e a aplicação dessas métricas, os procedimentos utilizados para coletar as métricas de cobertura de código, com a aplicação dos critérios Todas-Instruções e Todos-Desvios da técnica estrutural para os Grupos A e B.

Com o entendimento do planejamento e a execução do experimento mostrados anteriormente, serão apresentadas no próximo capítulo a discussão, análise e a interpretação dos resultados das métricas de LOC e de cobertura, que permitirão responder as questões definidas no GQM.

[18] O termo "gerados" significa a quantidade de instruções e desvios que foram desenvolvidos pela ferramenta na interceptação entre o código base e o aspecto

Capítulo 4

Discussão, Análise e Interpretação dos Resultados

4.1 Considerações Iniciais

Neste capítulo são apresentados os resultados obtidos após a execução de todos os testes e avaliações em todas as aplicações. São mostrados os resultados das métricas de LOC (Seção 4.2) e os resultados da aplicação do teste estrutural em ambos os grupos de aplicações (Seção 4.3) separados pelos critérios *Todas-Instruções* e *Todos-Desvios*.

Além disso, neste capítulo são descritos os passos realizados para a avaliação das questões de pesquisa com os resultados das métricas de LOC e de cobertura de código para o Grupo-A e o Grupo-B (Seção 4.4), e os resultados da avaliação definida no Capítulo 3. Apresenta-se também uma análise sobre a geração de requisitos estruturais de teste com duas aplicações (ShopSystem e Banco de Questões), nas versões OO e OA (Seção 4.4.3). Na Seção 4.5 é apresentada uma comparação entre os resultados obtidos de trabalhos relacionados com o presente estudo, e por fim, as considerações finais são apresentadas na Seção 4.6.

4.2 Resultados das Métricas de LOC

Nesta seção será tratada a Questão de Pesquisa 2, que está relacionada ao objetivo secundário definido na Seção 3.2.1. A questão tratada é analisar qual o custo relacionado ao esforço da adaptação dos conjuntos de testes da aplicação dos critérios na mudança do paradigma OO para o paradigma OA e vice-versa. Para isso, algumas métricas foram definidas para conseguir analisar o esforço da adaptação dos conjuntos de testes, como o

Total-LOC da aplicação, *#Classes*, *#Aspects*, *Total-LOC-TC*, *Churn-LOC-TC* (dividida em *ADD*, *MOD* e *REM*), Requisitos de Teste (*Test Requirements*) com as classes de equivalência e seus valores limite, e o total de casos de teste (*Total TC*).

A Tabela 4.1 exibe as métricas de todas as aplicações, juntamente com a aplicação Banco de Questões que foi baseada somente em dois interesses pra cada paradigma.

4.2.1 Resultados do Grupo-A

Em relação às métricas de LOC coletadas para os conjuntos de testes do Grupo-A, três aplicações não necessitaram de modificação nos conjuntos de teste para o paradigma OA. São as aplicações ShowWindow, AbstractObject e ExpressionInterpreter. As diferenças mais expressivas em relação ao aumento de LOC do conjunto de casos de teste (*TOTAL-LOC-TC*) foram observadas nas aplicações Boolean e Chess, com 27,6% e 7,5% de aumento, respectivamente.

Tratando-se de valores de *Churn-LOC-TC*, os testes para as aplicações Boolean, VendingMachine e Banco de Questões mostraram valores expressivos considerando métricas particulares. Por exemplo, para a aplicação Boolean da implementação OO havia 29 linhas de código do conjunto de testes em OO. Com a migração para o paradigma OA, o total de linhas de código de teste aumentou para 37 linhas, isto é, 27,6% a mais de código de teste OA. As aplicações VendingMachine e Banco de Questões tiveram mais modificações de código de teste, de acordo com os valores contabilizados para a métrica *MOD*. Para a aplicação VendingMachine, 5 linhas foram modificadas, de um total de 41 linhas de código de teste OO, isto é, 12,2%. Para a aplicação Banco de Questões, 6 linhas foram modificadas de um total de 47, resultando em 12,8% de modificação de código de teste OO para OA. Ressalta-se que ambas as implementações realizam a mesma especificação; sendo assim, os requisitos de teste e o número do total de casos de teste (*Total-TC*) foram os mesmos utilizados para ambos os paradigmas em todas as aplicações.

Em média, para o Grupo-A houve 5,7% de adição de código, 4,5% de modificação e 0% de remoção para adaptar conjuntos de testes ao paradigma OA. Os resultados obtidos mostraram pequenas diferenças em relação ao esforço na adaptação dos conjuntos de testes adequados para os paradigmas OO e OA. A respeito disso, parece haver uma tendência das aplicações do Grupo-A relacionada à necessidade de mais adições e modificações de linhas de código na adaptação de conjunto de testes do paradigma OO para o paradigma OA, mas é importante ressaltar que os valores das médias podem não ser tão relevantes. Esses resultados são discutidos mais adiante neste capítulo (Seção 4.4.1).

Tabela 4.1: Métricas de LOC coletadas pelos conjunto de testes.

Grupo A: OO - OA

Nome da aplicação	Total LOC TC	%. diff. tam.	Churn LOC TC						Test Requirements		
			ADD	%ADD	MOD	%MOD	REM	%REM	Classes equiv.	Valores limite	Total TC
1. ShowWindowOO	20								4	1	4
ShowWindowOA	20	0%	0	0%	0	0%	0	0%			
2. BooleanOO	29								7	3	7
BooleanOA	37	+27,6%	8	27,6%	1	3,4%	0	0%			
3. AbstractObjectOO	76								16	2	16
AbstractObjectOA	76	0%	0	0%	0	0%	0	0%			
4. ChessOO	281								28	13	39
ChessOA	302	+7,5%	21	7,5%	8	2,8%	0	0%			
5. ExpressionInterpreterOO	47								48	2	48
ExpressionInterpreterOA	47	0%	0	0%	0	0%	0	0%			
6. VendingMachineOO	41								9	10	10
VendingMachineOA	43	+4,9%	2	4,9%	5	12,2%	0	0%			
7. BancoQuestõesOO	47								5	3	8
BancoQuestõesOA	47	0%	0	0%	6	12,8%	0	0%			
MÉDIA		+5,7%		5,7%		4,5%		0%			

Grupo B: OA - OO

Nome da aplicação	Total LOC TC	%. diff. tam.	Churn LOC TC						Test Requirements		
			ADD	%ADD	MOD	%MOD	REM	%REM	Classes equiv.	Valores limite	Total TC
8. ATM-logOA	111								9	5	15
ATM-logOO	111	0%	0	0%	4	3,6%	0	0%			
9. ObjectRequestOA	108								6	0	6
ObjectRequestOO	96	-11,1%	0	0%	18	16,7%	12	11,1%			
10. SensitiveCharacterOA	36								4	4	4
SensitiveCharacterOO	36	0%	2	5,5%	4	11,1%	2	5,5%			
11. RecordValueOA	31								2	2	3
RecordValueOO	31	0%	0	0%	8	25,8%	0	0%			
12. ShopSystemOA	256								22	35	30
ShopSystemOO	256	0%	0	0%	0	0%	0	0%			
13. TelecomOA	257								12	16	23
TelecomOO	244	-5%	0	0%	15	5,8%	13	5%			
7. BancoQuestõesOA	50								5	5	6
BancoQuestõesOO	54	+8%	4	8%	2	4%	0	0%			
MÉDIA		-1,2%		1,9%		9,6%		3,1%			

4.2.2 Resultados do Grupo-B

Em relação às métricas de LOC coletadas para os conjuntos de testes das aplicações do Grupo-B, apenas para uma aplicação (ShopSystem) não houve a necessidade de modificação no conjunto de testes para o paradigma OA. Em relação ao Grupo-A, resultados do Grupo-B mostraram expressivas diferenças para LOC do conjunto de casos de teste (*TOTAL-LOC-TC*) em duas aplicações: ObjectRequest e Telecom. Porém, seguindo tendência contrária, observou-se 11,1% e 5% de diminuição de código de teste OO, respectivamente, quando comparado com o paradigma base (OA, neste caso).

A respeito das métricas de *Churn-LOC-TC*, as aplicações ObjectRequest, SensitiveCharacter e RecordValue mostraram valores expressivos considerando métricas específicas. Para a aplicação ObjectRequest, 18 linhas foram modificadas e 12 foram removidas de um total de 108 linhas de código de teste implementadas para a implementação OA, isto é, 16,7% e 11,1%, respectivamente. Para a aplicação SensitiveCharacter, 4 linhas foram modificadas de um total de 36 linhas de código de teste OA, resultando em 11,1% de modificação quando os casos de teste foram adaptados para o paradigma OO. Em relação ao conjunto de testes da aplicação RecordValue, o código de teste foi o mais modificado dentre todas as aplicações, com 8 linhas modificadas de um total de 31 linhas de código de teste OA, isto é, 25,8%. Novamente, ressalta-se que ambas as implementações realizam a mesma especificação; dessa forma, mantiveram-se os número de requisitos de teste e de casos de teste em todas as aplicações.

Em média, para o Grupo-B houve 1,9% de adição de código, 9,6% de modificação e 3,1% de remoção para adaptar conjuntos de teste ao paradigma OO. Os resultados do Grupo-B indicam apenas pequenas diferenças em relação ao esforço na adaptação dos conjuntos de testes para os paradigmas OA e OO. Para as aplicações do Grupo-B foi necessário realizar mais modificações e remoções de linhas de código de teste durante a migração do paradigma OA para o paradigma OO, mesmo que os valores das médias possam não ser tão significantes. Esses resultados são discutidos mais adiante na Seção 4.4.1.

4.3 Resultados das Métricas de Cobertura

Nesta seção será tratada a Questão de Pesquisa 1, que está relacionada ao objetivo primário definido na Seção 3.2.1. A questão tratada está relacionada à qualidade de conjuntos de testes adaptados para um software implementado em um dos paradigmas considerados, de acordo com critérios de seleção de casos de teste associados a uma deter-

minada técnica em relação ao software implementado no outro paradigma. Para responder essa questão, as métricas de cobertura de código *Todas-Instruções* e *Todos-Desvios* serão empregadas.

A análise dos resultados para as aplicações de pequeno porte é apresentada na Seção 4.3.1, enquanto a análise dos resultados referentes à aplicação Banco de Questões é analisada na Seção 4.3.2.

4.3.1 Resultados para as Aplicações de Pequeno Porte

Os resultados de cobertura de instruções e desvios para as aplicações de pequeno porte são apresentados na Tabela 4.2 e ilustrados nos gráficos apresentados nas Figuras 4.1 e 4.3 (Grupo-A) e nas Figuras 4.2 e 4.4 (Grupo-B).

Com base nas aplicações menores, os resultados apresentados na Tabela 4.2 indicam que há pouca diferença em cobertura quando os critérios *Todas-Instruções* e *Todos-Desvios* são aplicados em aplicações implementadas nos paradigmas OO e OA. No Grupo-A, os conjuntos de testes alcançaram uma porcentagem média de cobertura de instruções de 90,5% e 89,5% para implementações OO e OA, respectivamente. Para cobertura de desvios do mesmo grupo, as médias foram de 77,9% para OO e 78,9% para OA. No Grupo-B, os valores para os diferentes paradigmas também se mostraram próximos: 86,3% de cobertura de instruções para implementações OO e 84,9% para OA, e 66,5% e 67,2% de cobertura de desvios para implementações OO e OA, respectivamente.

Em seguida são mostrados os gráficos relacionados à cobertura de instruções (Figuras 4.1 e 4.2) e desvios (Figuras 4.3 e 4.4) das aplicações menores para o Grupo-A e para o Grupo-B.

Em relação à cobertura de instruções do Grupo-A (Figura 4.1), apenas os requisitos de teste das aplicações ShowWindow e AbstractObject foram cobertos totalmente com os conjuntos de testes adaptados do paradigma OO para o paradigma OA. As aplicações Boolean, Chess e VendingMachine tiveram poucas diferenças em cobertura de instruções, sendo que as coberturas mais altas na versão OA comparada à versão OO foram das aplicações Chess e VendingMachine. Apenas a aplicação ExpressionInterpreter teve uma maior diferença no resultado, com 92,0% de cobertura de instruções na versão OO e 85,5% na versão OA. Entretanto, não foi possível identificar uma diferença significativa entre as coberturas computadas.

Em relação à cobertura de instruções do Grupo-B (Figura 4.2), somente a aplicação

Tabela 4.2: Cobertura de instruções e desvios das aplicações menores.

Grupo A: OO - OA						
Nome da Aplicação	**%Instruções Cobertas**	**# Instruções**	**# Instruções Cobertas / Faltantes**	**%Desvios Cobertos**	**# Desvios**	**# Desvios Cobertos / Faltantes**
1. ShowWindowOO	100 %	134	134 / 0	n/a	n/a	n/a
ShowWindowOA	100 %	147	147 / 0	n/a	n/a	n/a
2. BooleanOO	87,5 %	431	377 / 54	66,7 %	24	16 / 8
BooleanOA	85,5 %	532	455 / 77	70,8 %	24	17 / 7
3. AbstractObjectOO	100 %	120	120 / 0	100 %	4	4 / 0
AbstractObjectOA	100 %	151	151 / 0	100 %	4	4 / 0
4. ChessOO	75,8 %	955	724 / 231	63,8 %	232	148 / 84
ChessOA	76,8 %	964	740 / 224	65,3 %	248	162 / 86
5. ExpressionInterpreterOO	92,0 %	225	207 / 18	71,4 %	14	10 / 4
ExpressionInterpreterOA	85,5 %	290	248 / 42	78,6 %	14	11 / 3
6. VendingMachineOO	87,9 %	321	282 / 39	87,5 %	16	14 / 2
VendingMachineOA	89,1 %	366	326 / 40	80,0 %	5	4 / 1
MÉDIA OO	90,5 %			77,9 %		
MÉDIA OA	89,5 %			78,9 %		
Grupo B: OA - OO						
Nome da Aplicação	**%Instruções Cobertas**	**# Instruções**	**# Instruções Cobertas / Faltantes**	**%Desvios Cobertos**	**# Desvios**	**# Desvios Cobertos / Faltantes**
8. ATM-logOA	72,7 %	326	237 / 89	71,4 %	14	10 / 4
ATM-logOO	80,8 %	271	219 / 52	83,3 %	12	10 / 2
9. ObjectRequestOA	76,7 %	257	197 / 60	68,8 %	16	11 / 5
ObjectRequestOO	77,7 %	157	122 / 35	66,7 %	18	12 / 6
10. SensitiveCharacterOA	82,5 %	120	99 / 21	87,5 %	8	7 / 1
SensitiveCharacterOO	85,4 %	82	70 / 12	75,0 %	8	6 / 2
11. RecordValueOA	100 %	112	112 / 0	n/a	n/a	n/a
RecordValueOO	100 %	44	44 / 0	n/a	n/a	n/a
12. ShopSystemOA	85,7 %	1581	1355 / 226	75,6 %	41	31 / 10
ShopSystemOO	82,6 %	872	720 / 152	73,8 %	80	59 / 21
13. TelecomOA	91,8 %	477	438 / 39	100 %	20	20 / 0
TelecomOO	91,6 %	381	349 / 32	100 %	20	20 / 0
MÉDIA OA	84,9 %			67,2 %		
MÉDIA OO	86,3 %			66,5 %		

RecordValue teve 100% de cobertura de requisitos nas versões OO e OA. De seis aplicações, três delas (ATM-log, ObjectRequest e SensitiveCharacter) resultaram em uma maior cobertura de instruções na versão OO. As aplicações ShopSystem e Telecom tiveram maior cobertura na versão OA, com diferenças de 3,1% e 0,2%, respectivamente. Com exceção da aplicação Atm-log, com 72,7% de cobertura na versão OA e 80,8% na versão OO, o restante das aplicações apresentaram 3,1% de diferença máxima de cobertura de instruções

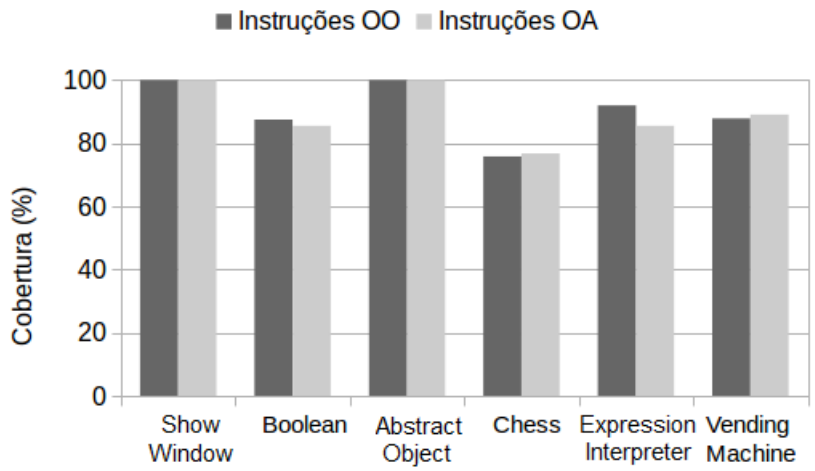

Figura 4.1: Cobertura de Instruções do Grupo-A das aplicações menores.

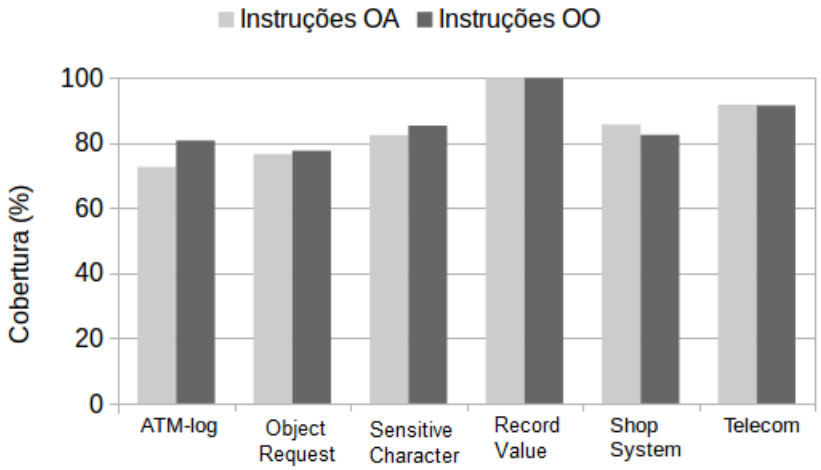

Figura 4.2: Cobertura de Instruções do Grupo-B das aplicações menores.

quando os dois paradigmas são considerados.

Em relação à cobertura de desvios do Grupo-A (Figura 4.3), somente o código da aplicação AbstractObject foi totalmente coberto. Os códigos das aplicações Boolean, Chess, ExpressionInterpreter se mostraram mais cobertos nas implementações OA, apresentando uma maior diferença de cobertura na aplicação ExpressionInterpreter, com 71,4% em OO e 78,6% em OA. A aplicação VendingMachine foi a única que teve mais desvios cobertos na implementação OO do que em OA, com a maior diferença de cobertura dentre todas as aplicações, com 7,5%. A aplicação ShowWindow foi a única que não apresentou requisitos de teste em relação ao critério *Todos-Desvios*.

Em relação à cobertura de desvios do Grupo-B (Figura 4.4), o código da aplicação

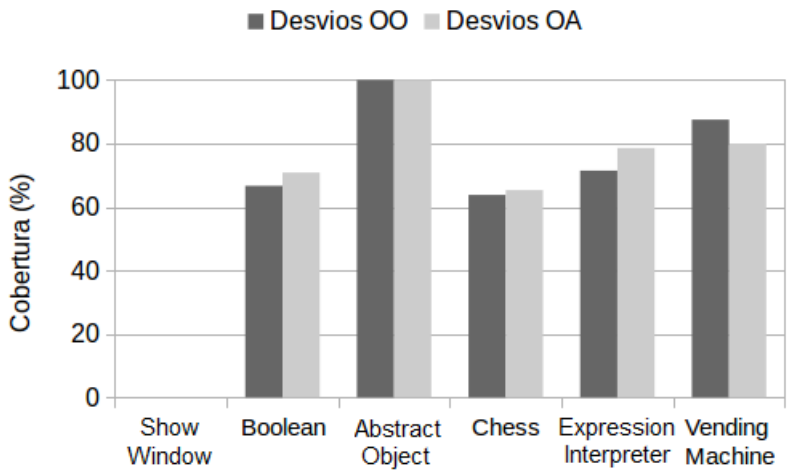

Figura 4.3: Cobertura de Desvios do Grupo-A das aplicações menores.

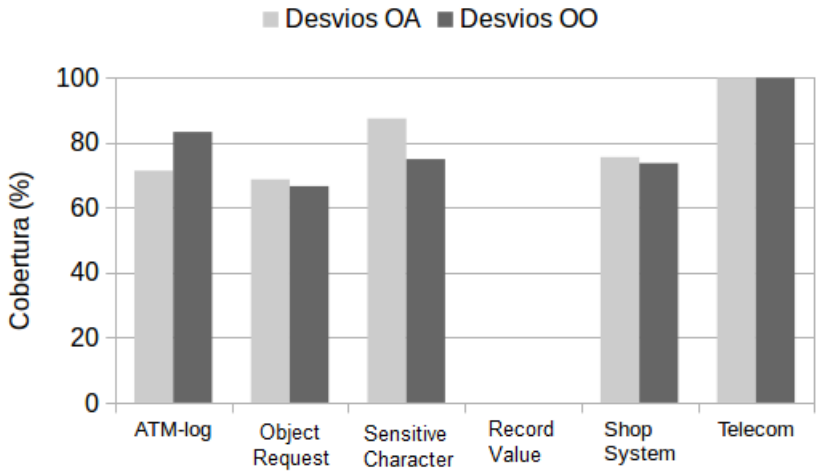

Figura 4.4: Cobertura de Desvios do Grupo-B das aplicações menores.

Telecom foi o único que teve 100% de cobertura nas versões OO e OA. As aplicações ATM-log e SensitiveCharacter tiveram maiores diferenças entre os resultados para esse critério, sendo ATM-log com 71,4% na versão OA e 83,3% em OO, e SensitiveCharacter com 87,5% em OA e 75,0% em OO. As aplicações ObjectRequest e ShopSystem tiveram resultados de cobertura de desvios um pouco mais altos em implementações OA do que em implementações OO, com diferenças de 2,1% e 1,8%, respectivamente. A aplicação RecordValue foi a única que não tinha desvios a serem cobertos.

Alguns gráficos de caixa, também conhecidos como gráficos de *box plot*, foram construídos para mostrar a variação das porcentagens de instruções e desvios obtidas pelos grupos A e B para as aplicações de pequeno porte.

Para o Grupo-A, são mostradas na Figura 4.5 as variações dos dados baseados nas instruções e desvios apresentados. Para o Grupo-B, essas informações são mostradas na Figura 4.6.

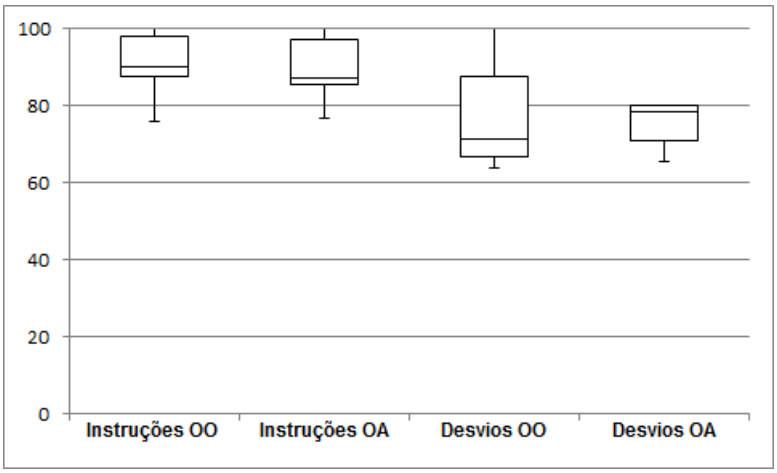

Figura 4.5: Box Plot do Grupo-A das aplicações menores.

Para o *box plot* do Grupo-A (Figura 4.5), pode-se verificar que algumas características permaneceram semelhantes tanto em coberturas de instruções OO e OA como em dispersão dos resultados. Em relação à cobertura de desvios, há algumas diferenças perceptíveis. Mesmo com a maior taxa de cobertura de código OO com respeito a desvios, os dados obtidos no paradigma OO apresentaram-se mais dispersos em comparação com o paradigma OA, e a mediana dos resultados obtidos em desvios para o paradigma OA foi superior a OO. Há maior semelhança em instruções entre os paradigmas OO e OA do que em desvios. Os resultados de desvios do paradigma OO se mostraram superiores ao do paradigma OA.

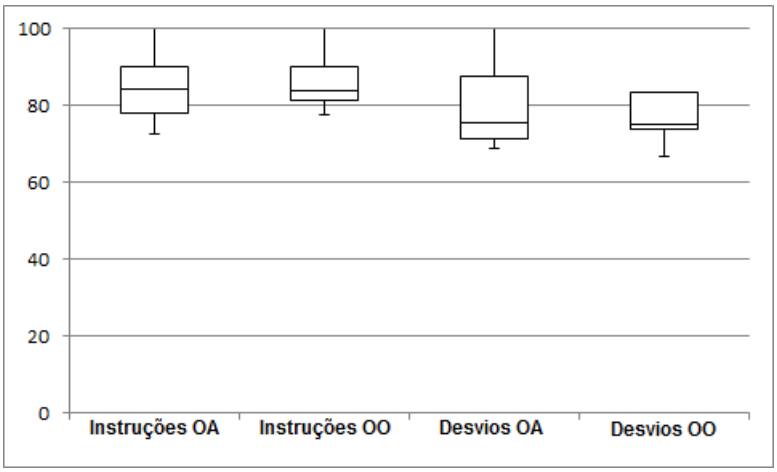

Figura 4.6: Box Plot do Grupo-B das aplicações menores.

Para as aplicações pequenas do Grupo-B, os resultados ilustrados na Figura 4.6 indicam

que o paradigma OO foi levemente superior ao OA, com menos dispersão entre os dados, porém os valores de suas medianas entre o paradigma OO e OA ficaram praticamente iguais. Em relação a desvios, os resultados do paradigma OA conseguiram ser levemente superiores do que em OO, porém com uma maior dispersão dos dados. Os dados se mostraram semelhantes em relação à dispersão entre as instruções e desvios para ambos os paradigmas.

Como conclusão, com base nas médias de cobertura de instruções e desvios resultantes exibidas na Tabela 4.2, tratando-se somente de instruções, as coberturas das aplicações OO superaram as coberturas das aplicações OA com uma diferença de 1% para o Grupo-A e 1,7% para o Grupo-B. Já com relação a desvios, as coberturas das aplicações OA superaram as coberturas das aplicações OO com uma diferença de 1% para o Grupo-A e 0,7% para o Grupo-B. Com essas pequenas diferenças de resultados obtidos, pode-se dizer que não há indícios de que um determinado paradigma considerado no estudo é melhor do que o outro, simplesmente por conseguir cobrir no máximo 1,7% de código a mais com conjuntos de testes adaptados para os paradigmas. Vale ressaltar que esses resultados foram obtidos a partir de aplicações de pequeno porte.

Na seção a seguir são apresentados os resultados do teste estrutural da aplicação Banco de Questões.

4.3.2 Resultados para a Aplicação Banco de Questões

Os resultados de cobertura de instruções e desvios para a aplicação Banco de Questões são apresentados na Tabela 4.3. Para o Grupo-A, os gráficos gerados são representados nas Figuras 4.7 e 4.9 com os interesses Segurança e Auditoria. Para o Grupo-B, os gráficos gerados são ilustrados nas Figuras 4.8 e 4.10 com os interesses Controle de Conexão e Tratamento de Exceção.

Ressalta-se que se esperava que a cobertura de código dessa aplicação fosse baixa, pois os conjuntos de testes foram criados somente para cobrir alguns interesses específicos da aplicação que se encontravam espalhados ou entrelaçados no restante do código da aplicação. A principal análise a ser feita é a avaliação da diferença de cobertura alcançada quando os dois paradigmas são comparados, especificamente nos módulos (classes e aspectos) que tinham alguma relação com os interesses considerados.

Com base na aplicação Banco de Questões, os resultados apresentados na Tabela 4.3 indicam que há uma substancial diferença em cobertura quando os critérios *Todas-Instruções*

Tabela 4.3: Cobertura de instruções e desvios da aplicação Banco de Questões.

Grupo A: OO - OA						
Nome da Aplicação	**%Instruções Cobertas**	**# Instruções**	**# Instruções Cobertas / Faltantes**	**%Desvios Cobertos**	**# Desvios**	**# Desvios Cobertos / Faltantes**
1. TimeOO	72,2 %	2329	1681 / 648	44,6 %	112	50 / 62
TimeOA	25,8 %	7971	2059 / 5912	3,8 %	1028	39 / 989
2. LoggingOO	46,3 %	605	280 / 325	23,1 %	26	6 / 20
LoggingOA	22,4 %	2015	451 / 1564	4,8 %	228	11 / 217
MÉDIA OO	59,2 %			33,9 %		
MÉDIA OA	24,1 %			4,3 %		
Grupo B: OA - OO						
Nome da Aplicação	**%Instruções Cobertas**	**# Instruções**	**# Instruções Cobertas / Faltantes**	**%Desvios Cobertos**	**# Desvios**	**# Desvios Cobertos / Faltantes**
1. ConnectionOA	11,3 %	3384	381 / 3003	2,7 %	413	11 / 402
ConnectionOO	24,9 %	977	243 / 734	15,4 %	26	4 / 22
2. ExceptionOA	5,5 %	7500	409 / 7091	2,7 %	308	8 / 300
ExceptionOO	16,2 %	2199	357 / 1842	5,0 %	126	6 / 120
MÉDIA OA	8,4 %			2,7 %		
MÉDIA OO	20,6 %			10,2 %		

e *Todos-Desvios* são aplicados nos específicos interesses das implementações OO e OA.

No Grupo-A, os conjuntos de testes alcançaram uma porcentagem média de cobertura de instruções de 59,2% e 24,1% para implementações OO e OA, respectivamente. Para cobertura de desvios do mesmo grupo, as médias foram de 33,9% para OO e 4,3% de cobertura de código para a implementação OA. Para o Grupo-B, os valores de porcentagens médias de cobertura entre os diferentes paradigmas também se mostraram bastante diferentes: 20,6% de cobertura de instruções para implementações OO e 8,4% para implementações OA, e 10,2% e 2,7% de cobertura de desvios para implementações OO e OA, respectivamente.

Em seguida são mostrados os gráficos relacionados à cobertura de instruções (Figuras 4.7 e 4.8) e desvios (Figuras 4.9 e 4.10) dos interesses selecionados da aplicação Banco de Questões para o Grupo-A e para o Grupo-B.

Em relação à cobertura de instruções (Figura 4.7) do Grupo-A, foram testados dois interesses: Segurança (representado por Time) e Auditoria (representado por Logging). Observou-se que a cobertura de código para instruções na implementação OO foi bastante superior do que na implementação OA, tendo como a maior diferença de cobertura o interesse Segurança com 72,2% na implementação OO e 25,8% na implementação OA.

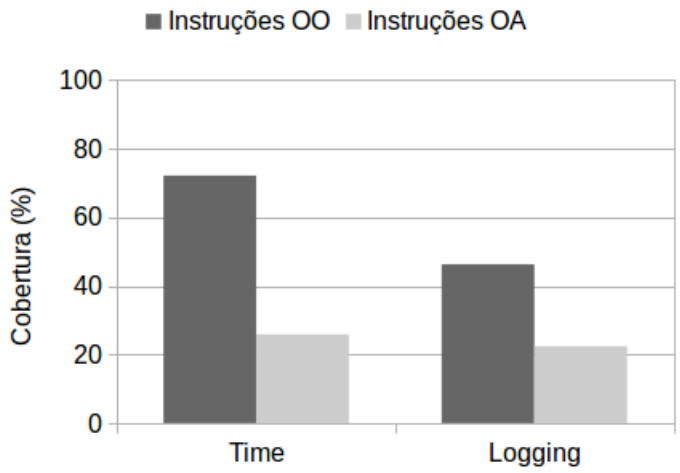

Figura 4.7: Cobertura de Instruções do Grupo-A da aplicação Banco de Questões.

Para o interesse Logging, 46,3% das instruções dos módulos considerados foram cobertas, enquanto somente 22,4% foram cobertas na implementação OA.

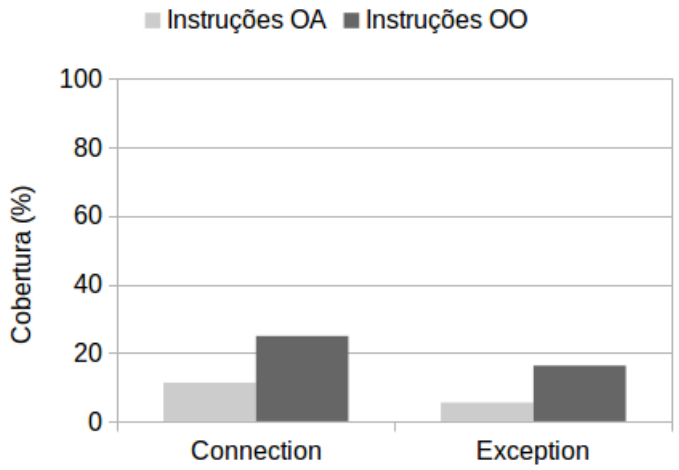

Figura 4.8: Cobertura de Instruções do Grupo-B da aplicação Banco de Questões.

Para o Grupo-B, foram testados outros dois interesses: Controle de Conexão (representado por Connection) e Tratamento de Exceção (representado por Exception). Em relação à cobertura de instruções (Figura 4.8), resultou em porcentagens de coberturas bastante superiores na implementação OO quando comparada com a implementação OA. O interesse de maior diferença de cobertura foi Tratamento de Exceção (Exception), com 5,5% na implementação OA e 16,2% na implementação OO. Com a execução dos testes para o interesse Controle de Conexão foram cobertos 11,3% de instruções na implementação OA e 24,9% na implementação OO.

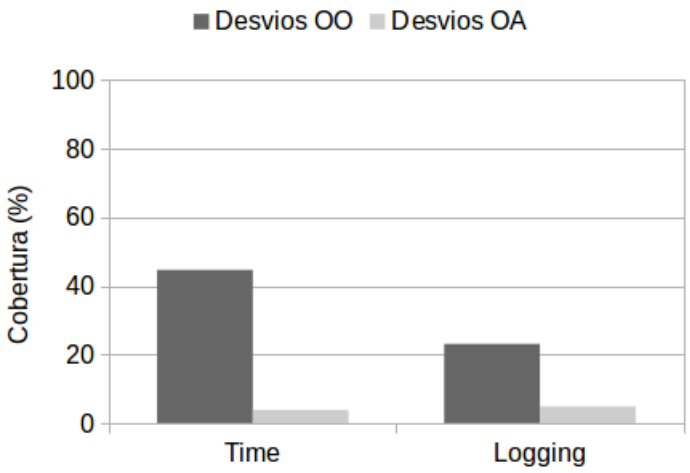

Figura 4.9: Cobertura de Desvios do Grupo-A da aplicação Banco de Questões.

Em relação à cobertura de desvios da aplicação Banco de Questões para o Grupo-A (Figura 4.9), foi observada uma maior diferença nos resultados de cobertura de requisitos de teste entre os paradigmas OO e OA tratando-se dos interesses Segurança e Auditoria. Na implementação OO, teve-se como resultado uma maior porcentagem de cobertura de código do que na implementação OA. Por exemplo, módulos relacionados ao interesse Segurança (representado por Time) tiveram uma cobertura de 44,6% de cobertura de desvios pelos casos de teste na implementação OO e 3,8% na implementação OA.

Para o interesse Auditoria (representado por Logging), a cobertura de desvios nos módulos relacionados foi de 23,1% na implementação OO e 4,8% na implementação OA.

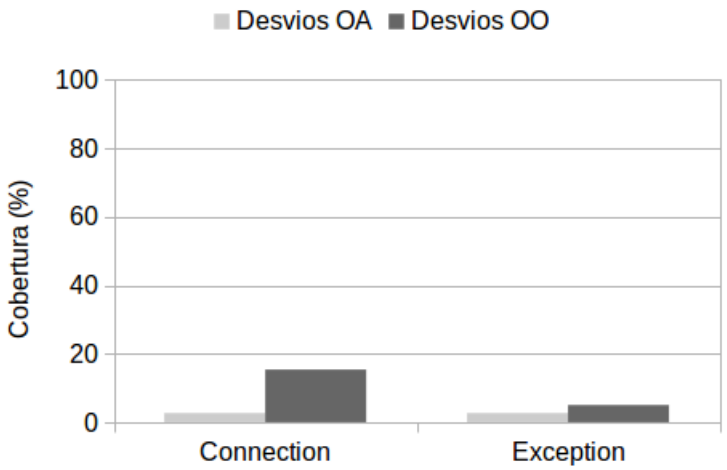

Figura 4.10: Cobertura de Desvios do Grupo-B da aplicação Banco de Questões.

Para o Grupo-B da aplicação Banco de Questões, em relação à cobertura de desvios

(Figura 4.10), foram cobertos muito mais requisitos de teste na implementação OO do que na implementação OA. Para o interesse Controle de Conexão computou-se 2,7% de cobertura na implementação OA e 15,4% de cobertura na implementação OO. Para o interesse Tratamento de Exceção, foram cobertos 2,7% de desvios na implementação OA e 5,0% na implementação OO.

Alguns gráficos de *box plot* também foram construídos para mostrar a variação das porcentagens de instruções e desvios obtidas pelos grupos A e B para a aplicação Banco de Questões. Para a aplicação Banco de Questões do Grupo-A, as variações dos dados baseados nas instruções e desvios são mostradas na Figura 4.11 e para o Grupo-B, na Figura 4.12.

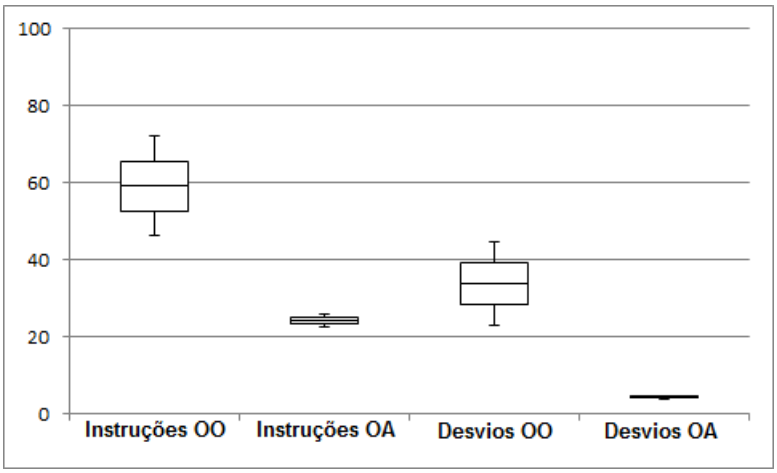

Figura 4.11: Box Plot do Grupo-A da aplicação Banco de Questões.

Para o Grupo-A (Figura 4.11), houve bastante diferença entre a cobertura de código atingida entre os paradigmas OO e OA tanto pelo critério Todas-Instruções quanto pelo critério Todos-Desvios. Para instruções, o código OO foi mais coberto pelos conjuntos de testes comparando-se com a cobertura do código OA. Situação semelhante ocorreu para a cobertura de desvios, com o código da aplicação OO mais coberto do que na aplicação OA, mesmo que a dispersão dos dados em instruções e desvios OO seja maior do que na aplicação OA. Pelos conjuntos de testes executados foi possível cobrir mais código na implementação OO do que na implementação OA, tanto em instruções como em desvios.

Para o Grupo-B (Figura 4.12), os resultados de cobertura de instruções e desvios para a implementação OO foram maiores em comparação à implementação OA. Em geral, houve pouca dispersão entre os dados, tanto para instruções quanto para desvios entre os dois paradigmas considerados.

Em relação à aplicação Banco de Questões pôde-se perceber nas Figuras 4.11 e 4.12 que houve uma diferença expressiva entre os dados em relação aos paradigmas OO e OA.

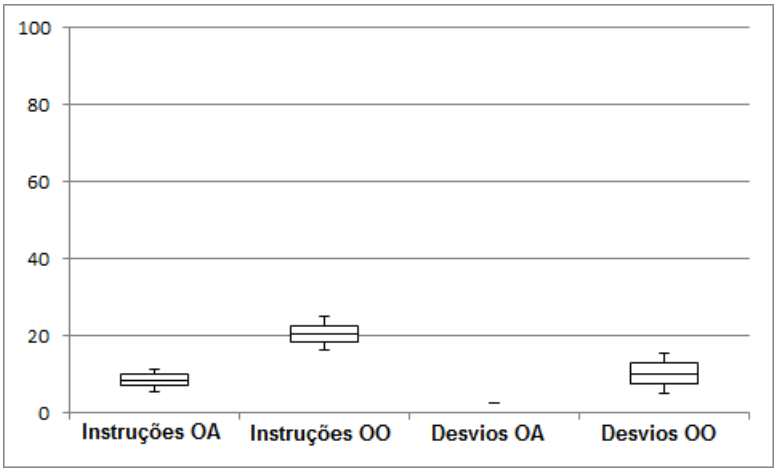

Figura 4.12: Box Plot do Grupo-B da aplicação Banco de Questões.

Os resultados obtidos pela implementação OA foram bastante inferiores aos resultados para a implementação OO, tanto em instruções quanto em desvios.

Como conclusão, as coberturas de código da aplicação Banco de Questões, para o conjunto de interesses considerados, mostraram-se inferiores na implementação OA do que na implementação OO. Observou-se também que a quantidade de requisitos de teste computados pela ferramenta EclEmma foi maior para os módulos considerados na implementação OA do Banco de Questões, pois alguns *pointcuts* foram definidos pelo desenvolvedor de forma genérica. Como consequência, menos código OA foi atingido pelo conjunto de teste, enfraquecendo assim a cobertura computada. Na Seção 4.4.3 discute-se a questão da geração de requisitos de teste computados pela ferramenta utilizada.

4.4 Análise e Discussão dos Resultados

Nesta seção são apresentadas a análise e a discussão dos resultados obtidos para as aplicações do Grupo-A e do Grupo-B. Para a Questão de Pesquisa 1 será discutida a cobertura estrutural atingida e para a Questão de Pesquisa 2 será discutido o esforço de migração de conjunto de teste entre os paradigmas, conforme definidas na Seção 3.2.1.

4.4.1 Esforço de Migração de Conjuntos de Teste entre Paradigmas

Para o Grupo-A, os resultados obtidos revelaram que, na média, adaptar conjuntos de teste OO para implementações OA necessitaram de 5,7% na adição de código de teste, 4,5% de modificação e 0% de remoção. Com esses resultados tem-se indícios de que migrar

conjuntos de teste baseados em implementações OO para implementações OA requer mais incremento e modificação de LOC de conjunto de testes do que remoção.

Para o Grupo-B, mais modificações e remoções precisaram ser feitas. Na média, as linhas de código dos conjuntos de testes foram 9,6% modificadas e 3,1% removidas para se alinharem às implementações OO, enquanto somente 1,9% das linhas de código de teste foram adicionadas. Com esses resultados, há indícios de que migrar conjuntos de teste baseados em implementações OA para implementações OO requer mais decremento e modificação de LOC de conjunto de testes do que adição.

Para ambos os grupos de aplicações (Grupo-A e Grupo-B), as porcentagens e os valores absolutos relacionados à adaptação dos conjuntos de teste foram relativamente baixos, pois a quantidade de linhas de código dos testes também foi baixa devido ao porte de aplicações utilizadas no experimento. Por outro lado, em aplicações de grande porte, com a possibilidade de testar mais funcionalidades complexas, os valores obtidos poderão ter um maior impacto, ou seja, o esforço na adaptação de conjunto de testes entre os paradigmas considerados poderá ser bem maior.

Algumas análises foram feitas a partir dos resultados obtidos:

1. Menos código é escrito para testes em aplicações OO, especialmente porque os casos de teste para implementações OA exigiram mais código específico para expor informações de contexto para construir assertivas JUnit;

2. O código de teste para aplicações OO é mais reutilizável, o que é refletido pelas médias das métricas *MOD* e *REM*. Essas médias indicam intervenções recorrentes em conjuntos de teste para sistemas OA, a fim de adaptá-los para implementações OO.

Para embasar as análises feitas a partir dos resultados, apresenta-se na Figura 4.13 um exemplo de um caso de teste da aplicação Chess. Ressalta-se que a aplicação em questão foi inicialmente testada em sua implementação OO; em seguida, os casos de testes foram adaptados para serem executados na implementação OA.

No primeiro caso de teste (OO), a assertiva (linha 15) utiliza diretamente o atributo `srtErrorMsg` do objeto `pawn` para a comparação definida. No segundo caso de teste (OA), nota-se que devido a um recurso da linguagem AspectJ, é usado o método `aspectOf()` para a recuperação do objeto que exibe a mensagem de erro. Assim sendo, há a necessidade de migrar o caso de teste com a adição dessas linhas de código específicas e recuperar o

Exemplo de um caso de teste - Aplicação Chess OO:

```
1  public void testPawnMovement_I4(){
2
3          startRow = 6; startColumn = 5;
4          desRow = 6; desColumn = 6;
5
6          assertFalse(pawn.legalMove(startRow, startColumn, desRow, desColumn,
7          cellMatrix.getPlayerMatrix(), player1 ));
8
9          startRow = 5; startColumn = 5;
10         desRow = 6; desColumn = 5;
11
12         assertFalse(pawn.legalMove(startRow, startColumn, desRow, desColumn,
13         cellMatrix.getPlayerMatrix(), player1 ));
14
15         assertEquals("Ilegal move", pawn.strErrorMsg);
16 }
```

Exemplo de um caso de teste - Aplicação Chess OA:

```
1  public void testPawnMovement_I4(){
2
3          startRow = 6; startColumn = 5;
4          desRow = 6; desColumn = 6;
5
6          assertFalse(pawn.legalMove(startRow, startColumn, desRow, desColumn,
7          cellMatrix.getPlayerMatrix(), player1 ));
8
9          startRow = 5; startColumn = 5;
10         desRow = 6; desColumn = 5;
11
12         assertFalse(pawn.legalMove(startRow, startColumn, desRow, desColumn,
13         cellMatrix.getPlayerMatrix(), player1 ));
14
15         ErrorMsg em = ErrorMsg.aspectOf();
16         String saida = em.getErrorMsg();
17         assertEquals("Ilegal move", saida);
18 }
```

Figura 4.13: Exemplo de um caso de teste para a aplicação Chess.

objeto com os recursos disponíveis da linguagem AspectJ (linhas 15 e 16), sempre que for necessário recuperar o objeto. Por fim, a assertiva original é adaptada para manipular os objetos instanciados (linha 17).

O exemplo apresentado na Figura 4.13 foi recorrente nas aplicações testadas no Grupo-A, fornecendo indícios de que a primeira conclusão destacada anteriormente é verdadeira, pois menos código de teste é escrito para aplicações OO. Como também visto na Tabela 4.1, no Grupo-A foi preciso adicionar mais código de teste OA para se alinhar ao paradigma OA, com uma porcentagem de 5,7% no aumento de linhas. Em relação ao Grupo-B, foi preciso diminuir mais código de teste em aplicações OA para se alinhar ao paradigma OO.

Para a segunda análise, as aplicações do Grupo-A obtiveram 4,5% de modificações de código de teste e 0% de remoções. Para as aplicações do Grupo-B, 9,6% das linhas do conjunto de testes foram modificadas e 3,1% foram removidas. Com isso, há maiores indícios de que os códigos de testes das aplicações OA são mais modificados e removidos, na

maior parte, para se adaptar às implementações OO, enquanto o Grupo-A não necessitou sequer de remoções de códigos de testes OO para se alinhar ao paradigma OA.

A análise em relação à segunda questão de pesquisa definida na Seção 3.2.2, em particular, dois pontos merecem destaque: (1) o código de teste para implementações OO pode ser mais reutilizado e é mais fácil de ser implementado, requerendo menos linhas de código; sua adaptação para o paradigma OA é mais simples, pois consiste principalmente em adições de instruções específicas da linguagem OA alvo (no caso deste estudo, AspectJ); e (2) o código de testes para implementações OA requer maior intervenção do programador para adaptá-lo para o paradigma OO, tendo em vista que modificações e remoções de instruções requerem uma melhor compreensão do código para evitar que defeitos sejam introduzidos no código de teste.

4.4.2 Cobertura Estrutural nos Paradigmas Considerados

Em relação às aplicações pequenas, para o Grupo-A, a média de instruções cobertas foi de 90,5% para as implementações OO e 89,5% para as implementações OA. Em relação à média de desvios, 77,9% foram cobertos para implementações OO e 78,9% cobertos para implementações OA. Para o Grupo-B, as instruções foram cobertas, em média, 84,9% para as implementações OA e 86,3% para implementações OO. Tratando-se de desvios, a média de cobertura foi de 67,2% para implementações OA e 66,5% para implementações OO.

Pode-se perceber que não houve uma diferença expressiva entre a porcentagem de cobertura relacionada às aplicações pequenas entre os paradigmas OO e OA. Para o Grupo-A, 1% a mais foi coberto para as aplicações OO com respeito às instruções, e 1% a mais foi coberto para as aplicações OA com relação aos desvios. Para o Grupo-B em cobertura de instruções, 1,4% a mais foi coberto para as aplicações OO, enquanto para desvios, 0,7% a mais foi coberto para as aplicações OA.

Diferentemente dos resultados das aplicações menores, os testes executados no sistema Banco de Questões resultaram em maior média de cobertura de instruções e desvios em todos os interesses na implementação OO. Para o Grupo-A, 35,1% a mais foi coberto de código na implementação OO tratando-se de instruções, enquanto a cobertura de desvios na implementação OO foi 29,6% superior. Tratando-se do Grupo-B, 12,2% a mais de instruções e 7,5% a mais de desvios foram cobertos na implementação OO.

Com essa diferença dos resultados para a aplicação Banco de Questões, é preciso uma maior atenção em alguns pontos que devem ser analisados. Como mencionado na Seção

3.3, a análise de cobertura foi realizada somente para os módulos que eram relacionados aos interesses específicos considerados. Com isso, os casos de teste foram projetados para esses interesses, não exercitando partes substanciais dos módulos envolvidos.

Como conclusão para a primeira questão de pesquisa (definida na Seção 3.2.2), a cobertura de código para as aplicações pequenas foi alta e os valores se apresentaram bem semelhantes em relação aos paradigmas OO e OA. Com isso, não se pôde perceber valores de cobertura discrepantes entre os paradigmas considerados, portanto, não há como garantir que um conjunto de teste escrito em uma determinada linguagem de programação é melhor do que o outro para esse porte de aplicações. Em outras palavras, não se pode afirmar que um conjunto de testes executado em uma implementação OO tem mais qualidade do que o mesmo conjunto adaptado para o paradigma OA e vice-versa. Porém, quando se analisam os resultados para uma aplicação de maior porte, no caso Banco de Questões, o conjunto de testes escrito para a implementação OO ou migrado para ela apresentou-se com maior qualidade em termos de cobertura obtida, pois as quantidades de instruções e desvios foram bem superiores às respectivas coberturas na implementação OA.

Pôde-se perceber que a quantidade de requisitos de teste estrutural na implementação OA foi bem maior do que na implementação OO, particularmente para a aplicação Banco de Questões. Ressalta-se que essa essa diferença também está presente para as aplicações de pequeno porte quando ambos os paradigmas são considerados. Sendo assim, na próxima seção apresenta-se uma análise que envolve uma comparação de requisitos estruturais gerados para as implementações OO e OA. Exemplos das aplicações ShopSystem e Banco de Questões são apresentadas para ambos os paradigmas.

4.4.3 Análise Adicional: Requisitos de Teste Estrutural

Motivado pelos resultados relacionados à cobertura estrutural considerando ambos os paradigmas investigados, uma análise adicional foi realizada para avaliar a quantidade de requisitos de teste gerados pela ferramenta EclEmma. Exemplos das aplicações ShopSystem e Banco de Questões foram utilizados nessa análise.

Inicialmente, ressalta-se que em uma aplicação AspectJ o compilador padrão da linguagem (ajc) combina os módulos principais (classes) e transversais (aspectos) em um sistema final. A manipulação desses módulos é realizada em nível de *bytecode*, basicamente consistindo na inserção de invocação de um ou mais *advices* sempre que uma condição definida em um *pointcut* é satisfeita. Nota-se que, em linhas gerais, cada *advice*

é compilado para um método da linguagem Java, permitindo tal invocação. A combinação de aspectos e classes, além de introduzir as chamadas aos *advices*, também pode deixar resíduos no código compilado (*bytecodes*), que são expressões que são avaliadas em tempo de execução para decidir se um *advice* de fato será ou não executado.

Como os requisitos de teste estrutural (instruções e desvios) computados pela ferramenta EclEmma advêm da análise dos *bytecodes* da aplicação e não do código fonte original, tal computação é a provável causa da grande quantidade de requisitos estruturais computados para alguns casos (alguns exemplos são mostrados nas Tabelas 4.4 e 4.5).

Para a aplicação ShopSystem, detalha-se na Tabela 4.4 as quantidades de requisitos para os critérios *Todas-Instruções* e *Todos-Desvios* computados pela ferramenta EclEmma. Para a aplicação Banco de Questões, as quantidades de instruções e desvios computados é apresentada na Tabela 4.5. A seguir é feita uma análise da possível causa da quantidade dessas gerações de requisitos.

Tabela 4.4: Aplicação do Teste Estrutural na aplicação ShopSystem - Detalhado por classes e aspectos.

Classes OO	Classes OA	Instruções OO	Instruções OA	Desvios OO	Desvios OA
ShopManager	ShopManager	293	783	26	2
AccountTransaction	AccountTransaction	56	56	2	2
ShoppingCart	ShoppingCart	120	120	8	8
Account	Account	56	56	2	2
AuthenticationManager		158		14	
AccountManager	AccountManager	45	45	2	2
Product	Product	21	21	0	0
AuthorisationManager	AuthorisationManager	94	126	26	16
LoggingManager		7		0	
User	User	22	22	0	0
	aspectos.AuthorisationManagerAdmin		72		2
	aspectos.AuthorisationManagerUser		72		2
	aspectos.LogonStatus		69		2
	aspectos.AspectPrecedence		25		0
	aspectos.CheckArguments		46		3
	aspectos.LoggingManagerLogoff		22		0
	aspectos.LoggingManagerLogon		21		0
	aspectos.LoggingManagerShopManager		25		0
		872	**1581**	**80**	**41**

Como pode ser visto na Tabela 4.4, a aplicação ShopSystem, em sua implementação OO, possui 10 classes, para as quais foram computadas 872 instruções e 80 desvios. Com

a mudança para o paradigma OA, a EclEmma computou 1581 instruções e 41 desvios, considerando as 8 classes e 8 aspectos presentes na aplicação. O que pode se observar é a grande diferença de instruções e desvios que são gerados entre os diferentes paradigmas com funcionalidades equivalentes em ambas as aplicações. Para instruções, a ferramenta EclEmma computou quase duas vezes mais requisitos para a implementação OA, enquanto a mesma ferramenta computou duas vezes mais desvios para a implementação OO (detalhes são apresentados na Tabela 4.4).

Na Tabela 4.5 são apresentados os quatro interesses considerados da aplicação Banco de Questões, sendo eles: Segurança, Auditoria, Controle de Conexão e Tratamento de Exceção.

Para o interesse Segurança do Grupo-A, 10 classes Java estavam relacionadas ao interesse para a implementação OO e 10 classes Java com mais 3 aspectos estavam relacionadas ao interesse para a implementação OA. Em relação à quantidade de instruções, foram computados 3 vezes mais requisitos para a implementação OA. Em relação a desvios, foram computados 9 vezes mais requisitos para a mesma implementação.

No interesse Tratamento de Exceção do Grupo-B, 5 classes Java e 3 aspectos estavam relacionadas ao interesse para o paradigma OA e 6 classes Java estavam relacionadas ao interesse para o paradigma OO. Em relação a instruções, foram geradas 3 vezes mais para a implementação OA e em relação a desvios foram gerados 2 vezes mais para a implementação OA.

Em geral, para a aplicação Banco de Questões, a ferramenta estrutural computou um total de 6110 instruções e 290 desvios na implementação OO e na implementação OA computou 20870 instruções e 1977 desvios. Pode-se perceber a grande diferença que os paradigmas têm em relação à geração de requisitos de teste (instruções e desvios). Para instruções, computou-se aproximadamente 3 vezes mais requisitos na implementação OA, enquanto esse número foi 7 vezes superior com relação a desvios.

Analisando-se as características dos elementos de programação OA empregados nas aplicações testadas, elenca-se a seguir duas possíveis razões para os elevados números de requisitos de teste extraídos das aplicações compiladas:

- A generalidade do aspecto para facilitar a evolução do sistema sem quebrar *pointcuts*[1]; e
- A estratégia adotada pelo desenvolvedor para criar aspectos (e suas partes internas)

[1]Quebrar *pointcuts* significa não conseguir capturar os *join points* desejados devido a particularidades da convenção de nomes empregados na elaboração dos *pointcuts*.

Tabela 4.5: Aplicação do Teste Estrutural na aplicação Banco de Questões - Detalhado por classes e aspectos.

Banco de Questões	Classes OO	Classes OA	Instruções OO	Instruções OA	Desvios OO	Desvios OA
Segurança	negocio.Usuario	negocio.Usuario	372	1.369	26	180
	apresentacao.FrmLogin	apresentacao.FrmLogin	437	1.238	24	160
	apresentacao.FrmPrincipal	apresentacao.FrmPrincipal	1.132	4.169	50	560
	apresentacao.Iniciar	apresentacao.Iniciar	49	98	2	8
	apresentacao.FrmIniciar	apresentacao.FrmIniciar	171	865	6	118
	funcoes.PacManKeyListener	funcoes.PacManKeyListener	11	11	0	0
	apresentacao.funcoes.Start	apresentacao.funcoes.Start	25	8	4	0
	apresentacao.funcoes.Clock	apresentacao.funcoes.Clock	27	8	0	0
	funcoes.MyAdapter	funcoes.MyAdapter	9	11	0	0
	dados.Banco	dados.Banco	96	32	0	0
		aspectos.TratamentoExcecao		101		0
		aspectos.Seguranca		27		0
		aspectos.ControleConexao		34		2
			2329	**7971**	**112**	**1028**
Auditoria	negocio.Disciplina	negocio.Disciplina	253	966	14	124
	negocio.Tema	negocio.Tema	215	792	12	100
	dados.Banco	dados.Banco	96	32	0	0
	dados.Logging	aspectos.Auditoria	41	75	0	2
		aspectos.ControleConexao		49		2
		aspectos.TratamentoExcecao		101		0
			605	**2015**	**26**	**228**
Controle de Conexão	negocio.Disciplina	negocio.Disciplina	253	966	0	124
	negocio.Tema	negocio.Tema	215	792	0	100
	negocio.Usuario	negocio.Usuario	372	1.369	26	180
	dados.Banco	dados.Banco	96	32	0	0
	dados.Logging	aspectos.Auditoria	41	75	0	2
		aspectos.ControleConexao		49		3
		aspectos.TratamentoExcecao		101		4
			977	**3.384**	**26**	**413**
Tratamento de Exceção	negocio.Disciplina	negocio.Disciplina	253	966	0	124
	negocio.Prova	negocio.Prova	1.222	4.116	100	0
	negocio.Tema	negocio.Tema	215	792	0	0
	negocio.Usuario	negocio.Usuario	372	1.369	26	180
	dados.Banco	dados.Banco	96	32	0	0
	dados.Logging	aspectos.Auditoria	41	75	0	2
		aspectos.ControleConexao		49		2
		aspectos.TratamentoExcecao		101		0
			2.199	**7.500**	**126**	**308**

utilizando mecanismos da linguagem AspectJ.

Ambas as razões estão relacionadas com a definição de *pointcuts* com amplo alcance (conhecidos como *pointcuts* fracos (MORTENSEN, 2005; ANBALAGAN; XIE, 2008)), ou

seja, *pointcuts* que selecionam um número elevado de *join points*, ativando o *advice* cuja execução de fato é resolvida em tempo de execução.

```
1  pointcut printStackTrace(): execution(* *.*(..)) && !within(TratamentoExcecao);
2
3  declare soft: Exception: withincode(* *.*(..)) && !within(TratamentoExcecao);
4
5  void around() : execution(void *.*(..)) && !within(TratamentoExcecao){
6      try{
7          proceed();
8      }
9      catch (Exception ex){
10         System.err.println("Exceção: "+ex.getCause());
11     }
12 }
```

Figura 4.14: Exemplo de implementação do interesse Tratamento de Exceção da aplicação Banco de Questões.

O exemplo de *pointcut* mostrado na Figura 4.14, chamado `printStackTrace` e extraído da aplicação Banco de Questões, captura a execução de qualquer método da aplicação, com exceção dos métodos definidos dentro do aspecto intitulado `TratamentoExcecao`. O advice around será ativado assim que uma exceção ocorrer na execução de todos os métodos capturados pelo *pointcut* `printStackTrace`. Quando ativado, o *advice* exibe a mensagem de erro da exceção lançada, permitindo que sistema continue em funcionamento.

```
1  pointcut LiberaBanco() : execution(public static * negocio.*.*(..));
2
3      after(): LiberaBanco() {
4          for (int i = 0; i < numConexoes; i++)
5              Banco.fechar();
6          numConexoes = 0;
7      }
```

Figura 4.15: Exemplo de implementação do interesse Controle de Conexão da aplicação Banco de Questões.

Um outro exemplo de *pointcut* de amplo alcance é mostrado na Figura 4.15, agora relacionado ao interesse Conexão da aplicação Banco de Questões. O *pointcut* em questão é chamado de `LiberaBanco` e captura a execução de todos os métodos das classes do pacote `negocio` que têm em sua assinatura os modificadores public static. Quando essa situação ocorre, o *advice* after é acionado, o qual é invocado no código base após a execução do método capturado para o fechamento da conexão do banco de dados. A quantificação de um elevado número de *join points* utilizada para esse *pointcut* é a similar à do exemplo apresentado na Figura 4.14.

Devido à generalidade dos *pointcuts* mencionados anteriormente, muitos requisitos de teste estrutural foram gerados pela ferramenta EclEmma na implementação OA. Dessa forma, a cobertura de código atingida para a implementação OA foi menor do que a implementação OO. Ressalta-se que não se trata de uma limitação da ferramenta EclEmma, pois a mesma é destinada à instrumentação e cálculo de cobertura de código objeto Java (isto é, *bytecodes*). Outras ferramentas de teste estrutural para programas Java adotam estratégia similar para instrumentar o código em teste, o que resultaria igualmente em uma grande quantidade de requisitos de teste para as aplicações OA testadas neste trabalho.

4.5 Comparação com Trabalhos Relacionados

Conforme mencionado nas Considerações Finais do Capítulo 2 deste documento,para a realização deste trabalho houve a inspiração em dois estudos, a investigação de Prado (2009) e a investigação de Campanha (2010).

Como visto na Seção 2.3.4, Prado (2009) apresentou um estudo experimental realizado para avaliar o custo de aplicação e a dificuldade de satisfação de critérios de teste, comparando dois paradigmas: OO e procedimental. Para a condução do estudo foram utilizados critérios funcionais e estruturais para um domínio de 32 especificações de Estrutura de Dados. Primeiramente foram executados os testes funcionais, e aproveitando-se dos testes já estabelecidos, os testes estruturais foram executados.

Para a análise da dificuldade de satisfação (*strenght*) entre os paradigmas, tomou-se como base a métrica Porcentagem de Cobertura/Programa no paradigma oposto, analisados com conjuntos de testes adequados aos paradigmas.

Os resultados obtidos por Prado (2009) pela análise de *strenght* encontram-se na Tabela 4.6. Vale ressaltar que os resultados de cobertura estrutural das aplicações de Prado (2009) mostraram-se superiores ao do presente estudo devido possivelmente à grande base de documentação das aplicações disponível nos livros de Ziviani (2005a, 2005b). Baseado em testes estatísticos, os resultados do trabalho de Prado (2009) não permitiram que o autor refutasse a hipótese de que há diferença entre o custo e a dificuldade de satisfação entre os paradigmas considerados.

Para o trabalho de Campanha (2010), foi utilizado o mesmo conjunto de testes adequados de aplicações utilizadas por Prado (2009), comparando critérios da técnica funcional e estrutural com o critério Análise de Mutantes. Os paradigmas considerados foram novamente o procedimental e o OO. Para o estudo de Campanha (2010), foram considerados

Tabela 4.6: Comparação dos resultados com trabalhos relacionados.

Investigação de (PRADO, 2009)			
Paradigma Procedimental		**Paradigma OO**	
Todos-Nós	**Todas-Arestas**	**Todos-Nós**	**Todas-Arestas**
Cobertura de código: 97,65%	95,16%	90,36%	93,40%
Investigação de (CAMPANHA, 2010)			
Paradigma Procedimental		**Paradigma OO**	
Operadores Essenciais	**Todos Operadores**	**Operadores Essenciais**	**Todos Operadores**
Escore de mutação: 86,70%	89,60%	95,80%	95,90%
Investigação do presente estudo			
Paradigma OO		**Paradigma OA**	
Grupo A – OO → OA			
Todas-Instruções	**Todos-Desvios**	**Todas-Instruções**	**Todos-Desvios**
Cobertura de código: 90,50%	77,90%	89,50%	78,90%
Grupo B – OA → OO			
Todas-Instruções	**Todos-Desvios**	**Todas-Instruções**	**Todos-Desvios**
Cobertura de código: 86,30%	66,50%	84,90%	67,20%

conjuntos de teste adequados aos critérios Análise de Valor Limite (técnica funcional) e o Todos-Potenciais-Usos (técnica estrutural). Resultados indicam que tanto o custo quanto o *strength* do teste de mutação é maior em programas procedimentais do que em programas OO. Os resultados também mostram que os conjuntos de casos de teste adequados aos critérios das técnicas funcional e estrutural no paradigma OO obtiveram, em geral, um maior escore de mutação do que no paradigma procedimental, como pode ser visto na Tabela 4.6.

Para o presente estudo foram coletadas 13 especificações, concretizadas em 26 implementações divididas igualmente entre os paradigmas OO e OA (Grupo-A e Grupo-B). Primeiramente foram feitos testes funcionais utilizando o critério Teste Funcional Sistemático para todas as aplicações. Para o Grupo-A foi tomado como base somente as implementações OO, pois os conjuntos de testes criados para essas implementações posteriormente seriam migrados para as respectivas implementações OA, e vice-versa para o Grupo-B. Isso se deu para analisar o quão difícil seria migrar conjuntos de testes de um paradigma a outro em relação ao números de linhas de código que constituíam os casos de teste para cada paradigma.

A ênfase do estudo é reutilizar conjuntos de testes desenvolvidos com base em uma determinada técnica de teste (funcional, no caso), com as devidas adaptações de código

de teste entre diferentes paradigmas, mas sem incrementos ou decrementos de casos de testes para garantir cobertura com respeito a outras técnicas e critérios. Os conjuntos de testes foram reutilizados para a técnica de teste estrutural no presente estudo.

Em relação à cobertura estrutural das aplicações menores, não foi observada diferença expressiva para permitir que se afirmasse que a cobertura de código em um paradigma é melhor do que no outro, no caso, OO e OA. Os resultados são sumarizados na Tabela 4.6. Em relação à métrica LOC de conjuntos de testes adaptados de um paradigma a outro, pelo Grupo-A, necessitaram de mais adição e modificação de código de teste para que as implementações OO se adaptassem às implementações OA. Já para o Grupo-B, houve necessidade de maior modificação e remoção de código de teste para que as implementações OA se adaptassem às implementações OO.

Para o presente estudo, é importante ressaltar que foram coletadas aplicações variadas e sem documentação pronta das especificações. Sendo assim, foi preciso criar uma documentação contendo todas as especificações das aplicações. A escassez de documentação pode ser um dos motivos que levaram a resultados ligeiramente inferiores de cobertura estrutural quando comparados aos resultados de Prado (2009) e Campanha (2010).

Tabela 4.7: LOC das aplicações com os casos de teste gerados.

Investigação de (PRADO, 2009)	**LOC Procedimental**	**LOC OO**	**Casos de testes**
	2114	1292	339
Investigação de (CAMPANHA, 2010)	**LOC Procedimental**	**LOC OO**	**Casos de testes**
	2114	1292	349
Investigação do presente estudo	**LOC OO**	**LOC OA**	**Casos de testes**
	3160	3183	205

Os números de linhas de código fonte das aplicações OO e procedimentais coletadas por Prado (2009) e Campanha (2010), juntamente com os casos de teste gerados[2], são apresentados na Tabela 4.7 para comparação com o presente estudo relacionado somente às aplicações menores.

Para os números de casos de teste gerados, no trabalho de Prado (2009) e Campanha (2010) houve uma média de 1 caso de teste para cada 4 linhas de código e para o presente estudo, 1 caso de teste para cada 15 linhas de código, todos para o paradigma OO. Pode-se perceber uma maior complexidade do caso de teste escrito para o presente estudo do que para os trabalhos relacionados.

[2]Foram adicionados 10 casos de teste para o trabalho de Campanha (2010) em comparação com o trabalho de Prado (2009) para os mesmos conjuntos de aplicações.

4.6 Considerações Finais

Neste capítulo foram discutidos os resultados obtidos das métricas de LOC e das métricas de cobertura de código para os dois grupos de aplicações testadas. As métricas de LOC foram utilizadas para medir a dificuldade na adaptação de conjunto de testes entre paradigmas diferentes. As coberturas de requisitos estruturais foram utilizadas para avaliar a qualidade de conjuntos de testes entre os paradigmas OO e OA, aproveitando-se dos conjuntos de testes utilizados da técnica funcional. Foram construídos alguns gráficos para melhor visualização dos resultados obtidos de cobertura estrutural e também gráficos *box plot* para mostrar a variação das porcentagens de instruções e desvios cobertos para os dois grupos de aplicações.

Em relação ao esforço na migração de testes entre os paradigmas considerados, o código de teste OO pode ser mais reutilizável e pode ser implementado com mais facilidade, comparado aos testes implementados em OA. Em relação à qualidade dos conjuntos de teste entre os paradigmas OO e OA, não se pôde garantir que um conjunto de teste escrito em uma determinada linguagem de programação é melhor do que o outro sob o domínio das aplicações menores. Em um domínio de maior aplicação como a aplicação Banco de Questões, a qualidade do conjunto de teste da implementação OO foi melhor do que a da implementação OA.

No próximo capítulo a conclusão e trabalhos futuros são apresentados.

Capítulo 5

Conclusão e Trabalhos Futuros

As atividades de Verificação e Validação (V&V) de software são fundamentais para ajudar a aumentar a qualidade do software. A qualidade de software é um dos principais requisitos a serem considerados durante o desenvolvimento. Para que o software atinja um nível de aceitação de qualidade é preciso seguir algumas atividades de V&V, e uma dessas atividades é o teste de software, cujo objetivo é revelar defeitos presentes nos artefatos produzidos (MYERS; BADGETT; SANDLER, 2011).

Paradigmas de programação são propostos e evoluídos para superar algumas deficiências de alguns paradigmas já existentes. A criação da Programação Orientada a Objetos (POO), na década de 70, surgiu com o objetivo de tentar suprir deficiências do paradigma procedimental, por exemplo, a fim de fornecer um mecanismo para isolar os dados da forma como eram manipulados com o uso de encapsulamento e ocultação da informação. Na década de 90 foi proposta a Programação Orientada a Aspectos (POA) para possibilitar uma melhor modularização de interesses transversais, a fim de que os interesses não ficassem espalhados pelos diversos módulos do sistema ou entrelaçados aos demais interesses.

Embora os paradigmas OO e OA tenham conseguido resolver alguns problemas, o emprego de qualquer um desses paradigmas para o desenvolvimento de software pode introduzir novos defeitos. Testes funcionais e estruturais mostram-se bastante importantes para a avaliação da qualidade do software ou mesmo do conjunto de testes utilizado para testá-lo, considerando esses paradigmas.

Na teoria há um certo consenso de que testar software OA é mais difícil do que testar software OO (FERRARI et al., 2013), porém essa hipótese ainda não havia sido avaliada na prática. O presente estudo foi motivado por essa inexistência de avaliação prática da

dificuldade de se testar software OA quando comparado com o teste de software OO. Além disso, motivou a pesquisa na reutilização de conjuntos de testes quando há uma migração de paradigma, tanto de software OO → OA como de software OA → OO, podendo assim avaliar se a mencionada hipótese se confirmaria.

Para o presente estudo, analisou-se a aplicação do critério Teste Funcional Sistemático da Técnica Funcional para aplicações funcionalmente equivalentes entre os paradigmas OO e OA, considerando a criação de conjuntos de testes adequados para os paradigmas com o objetivo de identificar o esforço na adaptação de conjuntos de testes de implementações OO para OA (e vice-versa). Foi analisada também a qualidade desses conjuntos de testes adaptados com a aplicação dos critérios Todas-Instruções e Todos-Desvios da Técnica Estrutural, analisando a cobertura de código atingida em ambos os paradigmas.

Para a condução do experimento foi utilizado um conjunto de 13 especificações, com implementações funcionalmente equivalentes em OO e OA. As aplicações foram separadas em dois grupos, Grupo-A e Grupo-B, e foram construídos conjuntos de testes adequados ao critério SFT, tomando como base as implementações OO e OA, respectivamente para cada grupo. Foi feita a adaptação do conjunto de testes de cada grupo para sua implementação oposta, ou seja, para a implementação no outro paradigma considerado. A partir de métricas definidas, foi analisado o esforço na migração dos conjuntos de testes adequados para os paradigmas. Com os mesmos conjuntos de testes adaptados foi possível analisar a qualidade desses conjuntos pela cobertura de código atingida em ambos os paradigmas, para ambos os grupos (Grupo-A e Grupo-B).

Resultados indicam que o conjunto de testes de aplicações OO migradas para OA necessitaram de mais incremento de LOC, e de aplicações OA para OO necessitaram de mais decremento e modificação de LOC. Em relação à cobertura de código estrutural, não houve uma diferença expressiva entre as aplicações pequenas para os paradigmas OO e OA do Grupo-A e do Grupo-B. Em relação à aplicação Banco de Questões, de maior porte, a cobertura de código estrutural para a implementação OO foi superior à cobertura da implementação OA pelos conjuntos de testes adaptados aos paradigmas.

Em consequência das grandes diferenças de resultados para a aplicação Banco de Questões, pôde-se perceber a grande quantidade de requisitos gerados pela ferramenta de teste estrutural para a implementação OA devido à implementação de *pointcuts* fracos. Portanto, não foi possível cobrir de forma substancial os requisitos de teste com testes feitos somente para interesses específicos.

5.1 Contribuições

Pode-se destacar como principais contribuições deste trabalho:

- Identificação do esforço na migração de conjunto de testes para se adequar aos paradigmas orientado a objetos e orientado a aspectos.
- Comparação da qualidade do conjunto de testes adequados ao critério SFT entre os paradigmas orientado a objetos e orientado a aspectos.
- Reutilização de código de teste, com as devidas adaptações, no contexto inter-paradigma (OO $\leftrightarrow$ OA).
- A definição de um arcabouço de artefatos e resultados experimentais como os documentos criados para conduzir os testes nas aplicações e os resultados obtidos para cada grupo de aplicações.

5.2 Limitações

Algumas limitações do trabalho são:

- A utilização de poucas aplicações funcionalmente equivalentes implementadas em OO e OA.
- A não evolução dos conjuntos de teste funcionais para se tornarem adequados aos critérios estruturais empregados devido a limitações tecnológicas.
- As hipóteses do experimento não foram formuladas, portanto não foram realizadas as análises estatísticas dos resultados obtidos. Essa análises não foram realizadas devido ao tamanho do conjunto de aplicações não ser suficientemente grande.

5.3 Trabalhos Futuros

Dentre as atividades que podem ser realizadas para dar continuidade ao trabalho e contribuir para a melhoria do mesmo, destacam-se:

- Considerar tanto outros domínios de aplicações para o experimento como aplicações de grande porte, dessa forma diminuindo a restrição da generalização dos resultados.

- A redefinição do estudo envolvendo mais linguagens de programação para comparar a qualidade de conjuntos de testes entre diferentes paradigmas e com um maior número de participantes para a execução do experimento.

- Trabalhar com diferentes critérios e técnicas de teste na criação de conjunto de testes adequados e avaliar o custo e o *strength* entre as diferentes técnicas de teste.

- Investigar outras formas de medir (métricas) as modificações feitas nos casos de teste quando um teste de uma implementação foi migrado para outra, criando diretrizes para as migrações.

Referências Bibliográficas

ALEXANDER, R. T.; BIEMAN, J. M.; ANDREWS, A. A. *Towards the Systematic Testing of Aspect-Oriented Programs*. Fort Collins/Colorado - USA, 2004.

ALVES, P. et al. How do programmers learn AOP? an exploratory study of recurring mistakes. In: *Proceedings of the 5th Latin American Workshop on Aspect-Oriented Software Development (LAWASP)*. São Paulo/SP - Brazil: Brazilian Computer Society, 2011. p. 131–140.

ANBALAGAN, P.; XIE, T. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In: *Proceedings of the 19 th International Symposium on Software Reliability Engineering (ISSRE)*. Seattle/WA - USA: IEEE Computer Society, 2008. p. 239–248. ISBN 978-0-7695-3405-3. ISSN 1071-9458.

BARBEY, S.; STROHMEIER, A. The problematics of testing object-oriented software. In: *Proceedings of the 2nd Conference on Software Quality Management*. Edinburgh - Scotland/UK: [s.n.], 1994. p. 411–426.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Toward the Determination of Sufficient Mutant Operators for C. *The Journal of Software Testing, Verification and Reliability*, John Wiley & Sons, v. 11, n. 2, p. 113–136, 2001.

BARTSCH, M. *Empirical Assessment of Aspect-Oriented Programming and Coupling Measurement in Aspect-Oriented Systems*. Tese (Doutorado) — University of Reading, 2007.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Goal Question Metric Paradigm. John Wiley & Sons, 1994.

BERTOLINO, A. The (im)maturity level of software testing. In: . New York, NY, USA: ACM, 2004. v. 29, n. 5, p. 1–4. ISSN 0163-5948.

BINDER, R. V. *Testing Object-Oriented Systems: Models, Patterns and Tools*. 1st. ed. Reading/MA - USA: Addison Wesley, 1999. ISBN 978-0201809381.

BOOCH, G. *Object-Oriented Analysis and Design with Applications*. 2nd.. ed. Redwood City/CA - USA: Addison Wesley, 1994.

BUDD, T. A. Mutation Analysis: Ideas, Example, Problems and Prospects, chapter Computer Program Testing. North-Holand Publishing Company, Amsterdam, p. 129–148, 1981.

CAFEO, B. B. P.; MASIERO, P. C. Contextual integration testing of object-oriented and aspect-oriented programs: A structural approach for Java and AspectJ. In: *Proceedings of the 25th Brazilian Symposium on Software Engineering (SBES)*. São Paulo/SP - Brazil: IEEE Computer Society, 2011. p. 214–223. ISBN 978-1-4577-2187-8.

CAMPANHA, D. N. *Teste de mutação nos paradigmas procedimental e OO: uma avaliação no contexto de estrutura de dados*. Dissertação (Mestrado) — ICMC-USP, 2010.

CAPRETZ, L. F. A brief history of the object-oriented approach. *SIGSOFT Software Engineering Notes*, ACM Press, v. 28, n. 2, 2003. ISSN 0163-5948.

CECCATO, M.; TONELLA, P.; RICCA, F. Is AOP code easier or harder to test than OOP code? In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*. Chicago/IL - USA: [s.n.], 2005.

CHAGAS, J. D. E.; OLIVEIRA, M. V. G. *Programação orientada a objetos versus programação orientada a aspectos. Um estudo de caso comparativo através de um banco de questões*. São Cristóvão - Brazil, 2009.

CHAIM, M. L. *Poke-Tool - uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados*. Dissertação (Mestrado) — DCA/FEEC/UNICAMP, Campinas - SP, 1991.

COELHO, R. et al. Assessing the impact of aspects on exception flows: An exploratory study. In: *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*. Paphos - Cyprus: Springer-Verlag, 2008. p. 207–234 (LNCS v.5142). ISBN 978-3-540-70591-8.

COLANZI, T. E. *Uma abordagem integrada de desenvolvimento e teste de software baseada na UML*. Dissertação (Mestrado) — ICMC-USP, 1999.

DELAMARO, M. E. *Proteum - um ambiente de teste baseado na análise de mutantes*. Dissertação (Mestrado) — ICMC/USP, São Carlos, SP, 1993.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. [S.l.: s.n.], 2007. ISBN 978-8535226348.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, 1978.

ELRAD, T. et al. Discussing Aspects of AOP. *Communications of the ACM*, ACM Press, v. 44, n. 10, p. 33–38, 2001.

FERRARI, F. C. *A contribution to the fault-based testing of aspect-oriented software*. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), São Carlos/SP - Brasil, 2010.

FERRARI, F. C. et al. Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice. In: *Proceedings of the 24th Brazilian Symposium on Software Engineering (SBES)*. Salvador/BA - Brazil: IEEE Computer Society, 2010. p. 50–59. ISBN 978-0-7695-4273-7.

FERRARI, F. C. et al. Difficulties for testing aspect-oriented programs: A report based on practical experience on structural and mutation testing. In: *Proceedings of the 7th Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*. Brasília/DF - Brazil: Brazilian Computer Society, 2013. p. 12–17. ISSN 2178-6097.

FERRARI, F. C.; RASHID, A.; MALDONADO, J. C. Towards the practical mutation testing of AspectJ programs. *Science of Computer Programming*, Elsevier North-Holland, Inc., Amsterdam - The Netherlands, v. 78, n. 9, p. 1639–1662, 2013. ISSN 0167-6423.

FILMAN, R. E.; FRIEDMAN, D. Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns - held in conjunction with OOPSLA*. Minneapolis - USA: [s.n.], 2000. p. 21–35.

FRANKL, F. G.; WEYUKER, E. J. *Data flow testing in the presence of unexecutable paths*. [S.l.], 1986. 4–13 p.

FRANKL, P. G.; WEISS, S. N.; HU, C. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 38, n. 3, p. 235–253, 1997. ISSN 0164-1212.

FRANKL, P. G.; WEYUKER, E. J. Testing Software to Detect and Reduce Risk. *Journal of Systems and Software*, Elsevier Science Inc., v. 53, n. 3, p. 275–286, 2000. ISSN 0164-1212.

HANNEMANN, J.; KICZALES, G. Design pattern implementation in Java and AspectJ. In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Seattle/WA - USA: ACM Press, 2002. p. 161–173. ISBN 1-58113-471-1.

HARROLD, M. J. Testing: A roadmap. In: *Proceedings of the Conference on the Future of Software Engineering - held in conjunction with ICSE*. Limerick - Ireland: ACM Press, 2000. p. 61–72. ISBN 1-58113-253-0.

HARROLD, M. J.; ROTHERMEL, G. Performing data flow testing on classes. In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 1994. (SIGSOFT '94), p. 154–163. ISBN 0-89791-691-3.

HILSDALE, E.; HUGUNIN, J. Advice weaving in aspectj. In: *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2004. (AOSD '04), p. 26–35. ISBN 1-58113-842-3.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. New York/NY - USA, 1990.

KAMIN, S. N.; MICKUNAS, M. D.; REINGOLD, E. M. *An introduction to computer science using Java*. [S.l.]: McGraw-Hill, 2002. ISBN 978-0071122320.

KICZALES, G. et al. Aspect-Oriented Programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Jyväskylä - Finland: Springer-Verlag, 1997. p. 220–242 (LNCS v.1241). ISBN 978-3-540-63089-0. ISSN 0302-9743.

KICZALES, G.; MEZINI, M. Aspect-oriented programming and modular reasoning. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005. (ICSE '05), p. 49–58. ISBN 1-58113-963-2.

KIM, S. woo; CLARK, J. A.; MCDERMID, J. A. Assessing test set adequacy for objectoriented programs using. In: *Class Mutation, 28 JAIIO: Symposium on Software Technology (SoST'99*. [S.l.: s.n.], 1999. p. 72–83.

LEMOS, O. A. L. *Teste de Programas Orientados a Aspectos: Uma Abordagem Estrutural para AspectJ*. Dissertação (Mestrado) — ICMC/USP, São Carlos/SP - Brasil, 2005.

LEMOS, O. A. L. et al. Testing aspect-oriented programming pointcut descriptors. In: *Proceedings of the 2nd Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with ISSTA*. Portland/Maine - USA: ACM Press, 2006. p. 33–38. ISBN 1-59593-415-4/06/0007.

LEMOS, O. A. L.; FRANCHIN, I. G.; MASIERO, P. C. Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for Java. *Science of Computer Programming*, Elsevier North-Holland, Inc., Amsterdam - The Netherlands, v. 74, n. 10, p. 861–878, 2009. ISSN 0167-6423.

LEMOS, O. A. L.; MALDONADO, J. C.; MASIERO, P. C. Structural unit testing of AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*. Chicago/IL - USA: [s.n.], 2005.

LEMOS, O. A. L.; MASIERO, P. C. A pointcut-based coverage analysis approach for aspect-oriented programs. *Information Sciences*, Elsevier Science Inc., Amsterdam - The Netherlands, v. 181, n. 13, p. 2721–2746, 2011. ISSN 0020-0255.

LEMOS, O. A. L. et al. Unit testing of aspect-oriented programs. In: *Proceedings of the 18th Brazilian Symposium on Software Engineering (SBES)*. Brasília/DF - Brazil: [s.n.], 2004. p. 55–70.

LEMOS, O. A. L. et al. Control and data flow structural testing criteria for aspect-oriented programs. *The Journal of Systems and Software*, Elsevier Science Inc., v. 80, n. 6, p. 862–882, 2007. ISSN 0164-1212.

LEMOS, O. A. L. et al. Visualization, analysis, and testing of Java and AspectJ programs with multi-level system graphs. In: *Proceedings of the 27th Brazilian Symposium on Software Engineering (SBES)*. Brasília/DF - Brazil: IEEE Computer Society, 2013. p. 49–58. ISBN 978-0-7695-5165-4.

LEVIN, T. G.; FERRARI, F. C. Is it difficult to test aspect-oriented software? Preliminary empirical evidence based on functional tests. In: *Proceedings of the 11th Workshop on Software Modularity (WMod)*. Maceio/AL - Brazil: Brazilian Computer Society, 2014. p. 15–26. ISSN 2178-6097.

LINKMAN, S.; VINCENZI, A. M. R.; MALDONADO, J. C. An evaluation of systematic functional testing using mutation testing. In: *Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering (EASE)*. Keele - UK: [s.n.], 2003. p. 1–15.

LIU, C.-H.; CHANG, C.-W. A state-based testing approach for aspect-oriented programming. *Journal of Information Science and Engineering*, Institute of Information Science, Academia Sinica, Taiwan, v. 24, n. 1, p. 11–31, 2008.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese (Doutorado) — DCA/FEE, Universidade Estadual de Campinas (UNICAMP), Campinas, SP - Brasil, 1991.

MALDONADO, J. C. et al. *Aspectos Teóricos e Empíricos de Teste de Cobertura de Software*. [S.l.], 1998.

MALDONADO, J. C. et al. *Introdução ao Teste de Software*. 2000. Minicurso apresentado no 14^o Simpósio Brasileiro de Engenharia de Software (SBES 2000). João Pessoa, PB/Brasil.

MCDANIEL, R.; MCGREGOR, J. D. *Testing polymorphic interactions between classes*. [S.l.], 1994.

MCEACHEN, N. Distributing classes with woven concerns: an exploration of potential fault scenarios. In: *In Proceedings of International Conference on Aspect-Oriented Software Development*. [S.l.]: ACM Press, 2005. p. 192–200.

MORTENSEN, M. An approach for adequate testing of aspectj programs. In: *In 2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD*. [S.l.: s.n.], 2005.

MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3nd. ed. [S.l.]: John Wiley & Sons, 2011. ISBN 978-1-118-03196-4.

NAGAPPAN, N.; BALL, T. Use of relative code churn measures to predict system defect density. In: *Proceedings of the 27^{th} International Conference on Software Engineering (ICSE)*. St. Louis, MO, USA: IEEE Computer Society, 2005. p. 284 – 292.

NTAFOS, S. C. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, IEEE Press, v. 14, n. 6, p. 868–874, 1988.

PRADO, M. P. *Um estudo de caracterização e avaliação de critérios de testes estruturais entre os paradigmas procedimental e OO*. Dissertação (Mestrado) — ICMC-USP, 2009.

PRECHELT, L. et al. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Computer Society, v. 27, p. 1134 – 1144, 2001. ISSN 0098-5589.

PRESSMAN, R. S. *Software engineering - A Practitioner's Approach*. 6th.. ed. New York/NY - USA: McGraw-Hill, 2005. ISBN 0072853182".

PRESSMAN, R. S. *Software engineering: A Practitioner's Approach*. 7th.. ed. New York/NY - USA: McGraw-Hill, 2010. ISBN 978-0-07-337597-7.

RAPPS, S.; J.WEYUKER, E. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, IEEE Press, v. 11, n. 4, p. 367–375, 1985.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: *Proceedings of the 6th International Conference on Software Engineering*. IEEE Computer Society Press, 1982. (ICSE '82), p. 272–278. Disponível em: <http://dl.acm.org/citation.cfm?id=800254.807769>.

SAVITCH, W. J. *Java: An introduction to computer science and programming*. 2th.. ed. [S.l.]: Prentice-Hall, 2001. ISBN 0-13-031697-0.

SMITH, B. H.; WILLIAMS, L. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 14, n. 3, p. 341–369, 2009. ISSN 1382-3256.

SMITH, M. D.; ROBSON, D. J. Object-oriented programming - the problems of validation. VI International Conference on Software Maintenance - ICSM'90, Washington, DC, EUA, p. 272–281, 1990.

SOMMERLAD, P.; GRAF, E. Cute: C++ unit testing easier. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. New York, NY, USA: ACM, 2007. (OOPSLA '07), p. 783–784. ISBN 978-1-59593-865-7.

The Eclipse Foundation. *AspectJ Documentation*. 2013. Online. `http://www.eclipse.org/aspectj/docs.php` - último acesso 24/07/2014.

TRAVASSOS, G. *Introdução à engenharia de software experimental*. Brasil, 2002.

van Deursen, A.; MARIN, M.; MOONEN, L. *A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw*. Amsterdam - The Netherlands, 2005.

VINCENZI, A. M. R. *Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação*. Tese (Doutorado) — ICMC/USP, São Carlos, SP - Brasil, 2004.

VINCENZI, A. M. R. et al. Establishing structural testing criteria for java bytecode. *Software: Practice and Experience*, John Wiley & Sons, v. 36, n. 14, p. 1513–1541, 2006.

WEYUKER, E. J. The Complexity of Data Flow Criteria for Test Data Selection. *Information Processing Letters*, Elsevier North-Holland, Inc., Amsterdam - The Netherlands, v. 19, n. 2, p. 103–109, 1984. ISSN 0020-0190.

WEYUKER, E. J. Using failure cost information for testing and reliability assessment. *ACM Transactions on Software Engineering and Methodology*, ACM Press, v. 5, n. 2, p. 87–98, 1996.

WEYUKER, E. J.; WEISS, S. N.; HAMLET, R. G. Comparison of program testing strategies. *4th Symposium on Software Testing, Analysis and Verification*, British Columbia - Canada, p. 1–10, 1991.

WOHLIN, C. et al. *Experimentation in Software Engineering: an Introduction*. [S.l.]: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

WONG, W. E. *On Mutation and Data Flow*. Tese (Doutorado) — Department of Computer Science, Purdue University, West Lafayette/IN - USA, 1993.

WONG, W. E.; MATHUR, A. P.; MALDONADO, J. C. *Mutation versus all-users: An empirical evaluation of cost, stregth and effectiveness*. [S.l.: s.n.], 1995. 258–265 p. ISBN 978-0-387-34848-3.

XIE, T.; ZHAO, J. Perspectives on automated testing of aspect-oriented programs. In: *Proceedings of the 3rd Workshop on Testing Aspect Oriented Programs (WTAOP)*. Vancouver/British Columbia - Canada: ACM Press, 2007. p. 7–12.

ZANNIER, C.; MELNIK, G.; MAURER, F. On the success of empirical studies in the international conference on software engineering. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. Shanghai, China: ACM, 2006. p. 341–350. ISBN 1-59593-375-1.

ZHANG, S.; ZHAO, J. On identifying bug patterns in aspect-oriented programs. In: *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*. Beijing - China: IEEE Computer Society, 2007. p. 431–438. ISBN 0-7695-2870-8. ISSN 0730-3157.

ZHAO, C.; ALEXANDER, R. T. Testing Aspect-Oriented Programs as Object-Oriented Programs. *Proceedings of the 3rd workshop on Testing aspect-oriented programs*, ACM, p. 23–27, 2007.

ZHAO, J. Data-flow-based unit testing of aspect-oriented programs. In: *Proceedings of the 27th Annual International Conference on Computer Software and Applications*. Washington, DC, USA: IEEE Computer Society, 2003. (COMPSAC '03), p. 188–. ISBN 0-7695-2020-0. Disponível em: <http://dl.acm.org/citation.cfm?id=950785.950862>.

ZIVIANI, N. *Projeto de algoritmos com implementações em java e c++*. [S.l.]: Thomson, 2005. ISBN 8522105251.

ZIVIANI, N. *Projeto de algoritmos com implementações em pascal e c*. 2th.. ed. [S.l.]: Thomson, 2005. ISBN 978-8522110506.

Glossário

AODU – *Aspect-Oriented Def-Use*

DUG – *Def-Use Graph*

GFC – *Grafo de Fluxo de Controle*

POA – *Programação Orientada a Aspectos*

POO – *Programação Orientada a Objetos*

SFT – *Systematic Functional Testing*