Teste estrutural de integração contextual de programas orientados a objetos e a aspectos

Bruno Barbieri de Pontes Cafeo

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 21/06/2011

Assinatura:_____

Teste estrutural de integração contextual de programas orientados a objetos e a aspectos

Bruno Barbieri de Pontes Cafeo

Orientador: **Prof. Dr. Paulo Cesar Masiero**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

USP – São Carlos
Junho de 2011

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

C129t

Cafeo, Bruno Barbieri de Pontes
Teste estrutural de integração contextual de programas orientados a objetos e a aspectos / Bruno Barbieri de Pontes Cafeo; orientador Paulo Cesar Masiero -- São Carlos, 2011.
133 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2011.

1. Teste de integração. 2. Teste estrutural. 3. Programação orientada a aspectos. 4. Programação orientada a objetos. I. Masiero, Paulo Cesar, orient. II. Título.

Agradecimentos

Agradeço inicialmente a Deus pela saúde, força e bênçãos recebidas durante esta etapa da minha vida.

Agradeço aos meus pais Ademar e Odila e ao meu irmão Danilo pelo apoio e confiança incondicionais.

Agradeço ao Prof. Dr. Paulo Cesar Masiero pela dedicação e profissionalismo durante esses anos.

Aos meus amigos do LabES: Abe, Endo, Nardi, Kátia, Fabiano, Maria, Cabeça, Frota, Alemão, Neiza, Joyce, Piu, Diogo, Harry, Draylson, Peixe, Marcão, Eduardo, Ceará, Vânia, Marcelão, Rodrigo, Vinícius; e a todos os outros que esqueci no momento mas que não são menos importantes.

Aos amigos de sempre: Bomba, Boulos, Bozoh, Ferro, Shimizu e Tiagão.

Aos amigos de Lancaster: Rachel, Júlio, Hayat, Edna, Alberto, Joost, Ruzanna e Awais.

Agradeço também a todos os professores e funcionários do ICMC, e a todas as pessoas que de alguma maneira contribuíram para a realização deste trabalho.

Agradeço à FAPESP pelo auxílio financeiro.

Resumo

Paradigmas e técnicas de desenvolvimento como a programação Orientada a Objetos (OO) e a programação Orientada a Aspectos (OA) procuram melhorar os níveis de reuso e manutenibilidade na produção de software. Contudo, com a introdução de mecanismos com maior poder de expressividade e, consequentemente, a possível introdução de novos tipos de defeitos, a utilização de linguagens OO e OA pode se tornar um obstáculo ao invés de um auxílio ao desenvolvimento de software. Para lidar com esse problema, nesta dissertação é proposta uma abordagem de teste estrutural de integração para programas orientados a objetos e a aspectos implementados em Java e AspectJ. É definido um modelo de fluxo de controle e de dados baseado no *bytecode* Java – chamado Grafo Def-Uso Contextual (ou *Contextual Def-Use graph*) – que é uma abstração formada pela integração dos grafos Def-Uso Orientados a Aspectos ($\mathcal{AODU}$) da unidade sob teste com todas as unidades que interagem direta ou indiretamente com ela até um nível de profundidade de interação máximo ou definido pelo testador. São definidos três critérios de teste: todos-nós-integrados-N_d, todas-arestas-integradas-N_d e todos-usos-integrados-N_d. Para automatizar o uso do modelo e critérios propostos, a ferramenta *JaBUTi/AJ* foi estendida. Exemplos de usos são discutidos e, por meio de um estudo experimental, uma análise de aplicabilidade da abordagem proposta é apresentada.

Abstract

Development paradigms and techniques such as Object-Oriented (OO) programming and Aspect-Oriented (AO) programming aim at improving reuse levels and maintenability in the software production. However, due to the introduction of mechanisms to support a greater power of expressiveness and, consequently, possible introduction of new type of faults, the use of OO and AO languages might become an obstacle instead of a benefit in the software development. To deal with these problems, in this dissertation is presented an integration structural testing approach for object- and aspect-oriented software based on Java and AspectJ. It is defined a control- and data-flow model based on Java bytecode – called Contextual Def-Use graph – that is an abstraction composed by the integration of Aspect-Oriented Def-Use graphs ($\mathcal{AODU}$) of the unit under testing with the units triggered by the execution of the unit under testing considering either a maximum interaction depth level or an interaction depth level previously defined by the tester. Three testing criteria are also defined: all-integrated-nodes-N_d, all-integrated-edges-N_d and all-integrated-uses-N_d. To automate the use of the model and the testing criteria, the *JaBUTi/AJ* tool was extended. Usage examples are discussed to explain the approach and an exploratory study is conducted to evaluate the applicability of the proposed approach.

Sumário

Lista de Figuras **ix**

Lista de Tabelas **xi**

1 Introdução **1**
1.1 Contextualização e Motivação 1
1.2 Objetivos 2
1.3 Organização 3

2 Teste de Software **5**
2.1 Considerações Iniciais 5
2.2 Fundamentos do Teste de Software 5
2.2.1 Terminologia e Conceitos Básicos 6
2.3 Técnicas e Critérios de Teste 8
2.3.1 Teste Funcional 8
2.3.2 Teste Estrutural 12
2.3.3 Teste Baseado em Defeitos 16
2.4 Fases da Atividade de Teste 17
2.5 Considerações Finais 18

3 Linguagens de Programação OO e OA **19**
3.1 Considerações Iniciais 19
3.2 Programação Orientada a Objetos e Java 19
3.2.1 Java 21
3.3 Programação Orientada a Aspectos e AspectJ 27
3.3.1 AspectJ 30
3.4 Considerações Finais 34

4 Teste de Programas OO e OA **35**
4.1 Considerações Iniciais 35
4.2 Fases do Teste de Software OO e OA 35
4.3 Teste de Programas Orientados a Objetos 37
4.3.1 Teste Estrutural de Unidade de Programas OO 38

4.3.2 Teste Estrutural de Integração de Programas OO 41
4.4 Teste de Programas Orientados a Aspectos 44
4.4.1 Teste Estrutural de Unidade de Programas OA 47
4.4.2 Teste Estrutural de Integração de Programas OA 52
4.4.3 Outras Abordagens de Teste de Programas OA 62
4.5 Considerações Finais 63

5 Teste Estrutural de Integração Contextual de Programas OO e OA 65
5.1 Considerações Iniciais 65
5.2 Fundamentos do Teste Estrutural de Integração Contextual 66
5.3 Modelo de Representação (Grafo $\mathcal{C}o\mathcal{DU}$) 67
5.3.1 Casos Especiais de Construção do Grafo $\mathcal{C}o\mathcal{DU}$ 78
5.4 Modelo de Fluxo de Dados 83
5.5 Critérios de Teste 87
5.5.1 Critérios de Fluxo de Controle 88
5.5.2 Critério de Fluxo de Dados 88
5.6 Estratégia Básica de Teste 94
5.7 Considerações Finais 97

6 Automação do Teste Estrutural de Integração Contextual 99
6.1 Considerações Iniciais 99
6.2 Extensão da Ferramenta *JaBUTi/AJ* 100
6.2.1 Identificação das Unidades 100
6.2.2 Construção do Grafo $\mathcal{C}o\mathcal{DU}$ 100
6.2.3 Implementação dos Critérios de Teste 101
6.2.4 Implementação do Ambiente de Teste 101
6.2.5 Esforço de Implementação 102
6.3 Exemplo de Uso da Ferramenta 102
6.3.1 A Interface da Ferramenta *JaBUTi/AJ* 103
6.3.2 Exemplo de Uso: Programa *Shape* 105
6.4 Considerações Finais 114

7 Avaliação da Abordagem de Teste Estrutural de Integração Contextual 115
7.1 Considerações Iniciais 115
7.2 Avaliação do Custo de Aplicação dos Critérios de Teste Estrutural de Integração Contextual 115
7.2.1 Caracterização do Estudo 116
7.2.2 Resultados 117
7.2.3 Limitações do Estudo 120
7.3 Considerações Finais 121

8 Conclusão 123
8.1 Considerações Finais 123
8.2 Contribuições 124
8.3 Trabalhos Futuros 124

Referências 127

Lista de Figuras

3.1 O ambiente de programação Java (adaptado de Venners (1999)). 27
3.2 Compilador e interpretador Java. 28
3.3 Interesse ortogonal de atualização de tela (adaptado do trabalho de Elrad et al. (2001)). 29
3.4 Solução OO para o exemplo de atualização de tela. 30
3.5 Exemplo de código em AspectJ. 31
3.6 Destaque do conjunto de junção nomeado, definido no exemplo apresentado na Figura 3.5 nas linhas 20-22 (adaptado do livro de Laddad (2003)). . . . 32

4.1 Implementação parcial da classe `SymbolTable` (Harrold e Rothermel, 1994). 38
4.2 Grafo de chamadas de classe para a classe `SymbolTable` (Harrold e Rothermel, 1994). 39
4.3 Exemplo dos grafos $\mathcal{IG}$ e $\mathcal{DUG}$ (adaptado de Vincenzi (2004)). 42
4.4 Grafo de fluxo de controle de classe ($\mathcal{CCFG}$) para a classe `SymbolTable` (Harrold e Rothermel, 1994). 45
4.5 Exemplo para o teste de programas OA (Zhao, 2003). 48
4.6 $\mathcal{FCFG}$ para o c-aspecto `PointShadowProtocol` e para a c-classe `Point` (Zhao, 2003). 49
4.7 Exemplo de um programa OA escrito em AspectJ (Lemos, 2005). 51
4.8 Grafo $\mathcal{AODU}$ do método `affectMethod` (Lemos, 2005). 52
4.9 Código de uma aplicação que realiza soma e subtração (Franchin, 2007). . 54
4.10 Grafos $\mathcal{AODU}$ das unidades chamadora e chamada (Franchin, 2007). . . . 54
4.11 Grafo $\mathcal{PWDU}$ do par de unidades (Franchin, 2007). 55
4.12 Código em Java que calcula o número de Fibonacci. 58
4.13 Grafo $\mathcal{AODU}$ do método `fib` 58
4.14 Grafo $\mathcal{IN}1\mathcal{P}$ do método `fib`. 59
4.15 Um exemplo simples de um programa orientado a aspectos (Bernardi e Lucca, 2007). 60
4.16 O $\mathcal{IACFG}$ do programa mostrado na Figura 4.15 (Bernardi e Lucca, 2007). 61

5.1 Código-fonte parcial do programa `Shape`. 67
5.2 Interações disparadas a partir do método `area`. 71
5.3 Grafos $\mathcal{AODU}$ das unidades que formarão o $\mathcal{CoDU}$. 72

5.4 Grafo $\mathcal{CODU}$ do método `area` no nível 1 de profundidade. 72
5.5 Grafo $\mathcal{CODU}$ do método `area` no nível 2 de profundidade. 73
5.6 Grafo $\mathcal{CODU}$ do método `area` no nível 3 de profundidade. 74
5.7 Grafo $\mathcal{CODU}$ do método `area` no nível 4 de profundidade. 75
5.8 Tratamento especial dado no caso de chamadas a métodos de biblioteca. . 79
5.9 Tratamento especial dado ao $\mathcal{CODU}$ no caso de recursão. 80
5.10 Tratamento especial dado para o caso de polimorfismo (sobrescrita). 81
5.11 Representação de adendos do tipo `around` e do comando `proceed()`. . . . 82
5.12 Tratamento especial dado para chamadas múltiplas a um mesmo método. . 83
5.13 Interações dos métodos `printEntering` e `printExiting` no nível 1 de profundidade. 94
5.14 Interações dos métodos `traceEntry` e `traceExit` no nível 1 de profundidade. 95
5.15 Interações dos métodos `traceEntry` e `traceExit` no nível 2 de profundidade. 95
5.16 Interações dos adendos `before` e `after` no nível 1 de profundidade. 95
5.17 Interações dos adendos `before` e `after` no nível 2 de profundidade. 96
5.18 Interações dos adendos `before` e `after` no nível 3 de profundidade. 96
5.19 Interações do método `area` em todos os níveis de profundidade. 97

6.1 Exemplo de nomeação de uma unidade 100
6.2 Tela principal da ferramenta *JaBUTi/AJ*. 104
6.3 Exemplos de interface da ferramenta *JaBUTi/AJ*. 105
6.4 Tela de seleção de unidades da ferramenta *JaBUTi/AJ*. 106
6.5 Conjunto de casos de teste adequado para os critérios de teste de unidade, para o programa *Shape*. 106
6.6 Cobertura do programa *Shape* para os critérios de unidade implementados na *JaBUTi/AJ*. 107
6.7 Gerenciador de projeto da ferramenta *JaBUTi/AJ*. 108
6.8 Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 1 de profundidade. . . 109
6.9 Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 2 de profundidade. . . 110
6.10 Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 3 de profundidade. . . 111
6.11 Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 4 de profundidade. . . 112
6.12 Conjunto de casos de teste adicionais para a cobertura dos requisitos não cobertos do método `area` nas profundidades 2, 3 e 4. 113

7.1 *Box-plots* de distribuições coletadas com a condução do experimento. . . . 119

Lista de Tabelas

3.1 Os tipos primitivos da linguagem Java. 21
3.2 Os especificadores de acesso da linguagem Java. 23
3.3 Os modificadores da linguagem Java. 24
3.4 Categorias de pontos de junção do AspectJ (Kiczales et al., 2001b). 31
3.5 Mapeamento entre as categorias de pontos de junção e a sintaxe dos conjuntos de junção correspondente. 32

4.1 Relação entre as fases de teste e o teste de programas OO (adaptada de Domingues (2001)). 37
4.2 Resumo dos trabalhos de teste de programas OA. 46

5.1 Exemplos de sentenças Java genéricas e suas implicações no fluxo de dados (Franchin, 2007). 87
5.2 Requisitos de teste para cada profundidade de cada critério de fluxo de controle, para o método `area`. 89
5.3 Requisitos de teste para cada profundidade do critério de fluxo de dados, para o método `area`. 93

6.1 Arquivos criados/modificados na extensão da ferramenta *JaBUTi/AJ*. . . . 103
6.2 Cobertura dos requisitos do método `area` (Nível 1) após a execução do conjunto de casos de teste `StructuralTS`. 110
6.3 Cobertura dos requisitos do método `area` (Nível 2) após a execução do conjunto de casos de teste `StructuralTS`. 111
6.4 Cobertura dos requisitos do método `area` (Nível 3) após a execução do conjunto de casos de teste `StructuralTS`. 112
6.5 Cobertura dos requisitos do método `area` (Nível 4) após a execução do conjunto de casos de teste `StructuralTS`. 113

7.1 Dados coletados a partir do teste de oito programas OA. 118

Capítulo

1

Introdução

1.1 Contextualização e Motivação

Nas últimas décadas, o software adquiriu um papel de grande importância no cenário mundial. Desde então, paradigmas e linguagens de programação evoluem e são adotados com o propósito de diminuir a complexidade de desenvolvimento e aumentar a flexibilidade e compreensão dos sistemas (Parnas, 1972). Um exemplo disso é a adoção da orientação a objetos (OO) no desenvolvimento de sistemas, o que possibilitou alcançar maiores níveis de reuso e manutenibilidade (Booch, 1994, p. 16-20).

Apesar dos benefícios alcançados com a adoção da OO, algumas limitações são notadas como, por exemplo, a impossibilidade de modelar de forma localizada em um único módulo (uma classe, por exemplo) certas partes do código que implementam alguns tipos de interesses (*concerns*)[1], causando seu espalhamento por vários módulos e entrelaçamento com outros interesses e requisitos (Elrad et al., 2001). Típicos exemplos de tais interesses — chamados de interesses transversais (*crosscutting concern*) (Kiczales et al., 1997) — são: distribuição, persistência de dados, segurança e tratamento de exceções.

Por outro lado, extensões do paradigma OO, como a programação orientada a aspectos (POA) (Kiczales et al., 1997), tentam solucionar as limitações do paradigma OO com o

[1]Segundo Dijkstra (1976), interesses são partes do sistema que o projetista decide tratar separadamente.

propósito de obter uma maior modularidade no software em situações práticas em que a orientação a objetos e padrões de projetos não fornecem o apoio adequado.

A POA é uma técnica de desenvolvimento contemporâneo que procura melhorar o nível de modularização de interesses transversais em um sistema. Essa técnica introduz o *aspecto* como um novo conceito de unidade com o propósito de encapsular as ações executadas pelos interesses transversais. Uma vez implementados, os aspectos são combinados com os outros módulos do sistema em um processo chamado de entrelaçamento de aspectos (do inglês, *aspect weaving*) (Kiczales et al., 1997) para a produção do sistema executável. O aumento da modularidade no desenvolvimento implica em sistemas com uma maior manutenibilidade, alta reusabilidade e maior facilidade de evolução (Coady e Kiczales, 2003; Laddad, 2003; Mortensen et al., 2010).

Contudo, devido à introdução de mecanismos que apoiam um maior poder de expressividade e, consequentemente, a possível introdução de novos tipos de defeitos, a utilização da POA pode se tornar um obstáculo ao invés de um auxílio no desenvolvimento de software (Alexander et al., 2004). Por isso, atividades de garantia de qualidade de software têm sido introduzidas ao longo de todo o processo de desenvolvimento com o objetivo de minimizar a ocorrência de problemas. Dentre as várias técnicas para garantia de qualidade de software, a atividade de teste é uma das mais utilizadas, já que as outras atividades são insuficientes para descobrir os defeitos introduzidos ao longo do processo de desenvolvimento (Pressman, 2005).

Dessa forma, para que a POA se consolide como uma técnica de desenvolvimento aplicável na prática profissional, são necessárias abordagens de teste de software específicas que considerem os novos defeitos introduzidos (Alexander, 2003; Bodkin, 2003; Ferrari et al., 2008). Alguns pesquisadores propuseram abordagens de teste estrutural de programas OO e OA (Bernardi e Lucca, 2007; Franchin, 2007; Lemos, 2005, 2009; Neves, 2009; Zhao, 2002, 2003), porém nota-se que o teste estrutural de integração para programas OA ainda não foi completamente explorado. Esta dissertação trata essa questão, propondo uma abordagem de teste estrutural de integração específica para programas OA.

1.2 Objetivos

O objetivo desta dissertação é propor uma abordagem de teste estrutural de integração para programas OA que permita que uma unidade possa ser testada com todas as unidades com as quais ela interage direta e indiretamente, seja a chamada de um método, seja o entrecorte de um aspecto. De forma específica, os objetivos desta dissertação são: (1) propor uma abordagem de teste de integração de programas OO e OA com a definição de um modelo de representação das unidades integradas e a definição de critérios de

teste com base no modelo definido; (2) automatizar a abordagem proposta por meio do desenvolvimento de uma extensão da ferramenta *JaBUTi/AJ* (estendida da ferramenta *JaBUTi* (Vincenzi, 2004)) e que apoie a aplicação dos critérios de teste definidos na abordagem, bem como a utilização do modelo de representação proposto; e (3) definir e conduzir estudos que visem a fornecer evidências da aplicabilidade da abordagem com a avaliação de custo de aplicação dos critérios de teste estrutural de integração contextual para programas OO e OA.

1.3 Organização

No Capítulo 2 é apresentada uma revisão bibliográfica sobre teste de software, em que são abordados os fundamentos do teste de software, a terminologia e os conceitos pertinentes, tais como critérios e fases de teste. No Capítulo 3 é apresentada uma introdução à programação orientada a objetos e orientada a aspectos. Além disso, a linguagem Java e a linguagem AspectJ são brevemente discutidas. No Capítulo 4 apresenta-se uma revisão bibliográfica sobre teste de software de programas orientados a objetos e de programas orientados a aspectos, com ênfase no teste estrutural. No Capítulo 5 é apresentada uma abordagem de teste estrutural de integração contextual para programas OO e OA e no Capítulo 6 é apresentada a implementação da abordagem em uma ferramenta de teste. No Capítulo 7 é apresentado um estudo de avaliação da abordagem proposta do ponto de vista do custo de aplicação dos critérios propostos. Finalmente, no Capítulo 8 são apresentadas as conclusões, as principais contribuições deste trabalho e os trabalhos futuros relacionados com esta dissertação.

Capítulo

2

Teste de Software

2.1 Considerações Iniciais

Neste capítulo é apresentada uma revisão bibliográfica sobre a atividade de teste de software. Na Seção 2.2 são abordados os fundamentos do teste de software, a terminologia e conceitos relevantes. Na Seção 2.3 são apresentados os critérios de teste funcional, estrutural — baseado em fluxo de controle e em fluxo de dados — e baseado em mutação. Ênfase é dada ao teste estrutural. Por fim, na Seção 2.4 são discutidas as fases do teste de software e na Seção 2.5 são apresentadas as considerações finais sobre o capítulo.

2.2 Fundamentos do Teste de Software

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar das técnicas, ferramentas e métodos empregados, erros no produto ainda podem ocorrer (Maldonado et al., 2004). Para que tais erros sejam descobertos antes de o software ser liberado para utilização, atividades agregadas sob o nome de **Garantia de Qualidade de Software** têm sido introduzidas ao longo de todo o processo de desenvolvimento, entre elas atividades de VV&T – Verificação, Validação e Teste. **Verificação** se refere ao conjunto de atividades que garante que o software implementa corretamente uma função específica (Pressman, 2005). **Validação** se refere a um conjunto de atividades diferentes que garante que o software construído corresponde aos requisitos do cliente (Pressman,

2005). Por fim, **Teste** se refere ao processo de execução de um programa com a finalidade de encontrar erros (Myers et al., 2004).

Atividades de VV&T não se restringem ao produto final. Ao contrário, podem e devem ser conduzidas durante todo o processo de desenvolvimento de software por meio de um amplo conjunto de atividades de garantia de qualidade, que inclui revisões técnicas formais, auditoria de qualidade e configuração, monitoramento de desempenho, simulação, estudo de viabilidade, revisão da documentação, revisão da base de dados, análise de algoritmos, teste de desenvolvimento, teste de usabilidade, teste de qualificação e teste de instalação (Wallace e Fujii, 1989). Estas atividades são comumente classificadas em atividades estáticas e dinâmicas. As **atividades estáticas** são aquelas que não requerem a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas (Delamaro et al., 2007). As **atividades dinâmicas** são aquelas que se baseiam na execução de um programa ou modelo (Delamaro et al., 2007).

Entre as técnicas de verificação e validação, a atividade de teste — que é uma atividade dinâmica — é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades (Maldonado, 1991). Salienta-se ainda que mesmo sendo apontada como uma das atividades mais onerosas no desenvolvimento de software (Myers et al., 2004; Pressman, 2005), conhece-se muito menos sobre teste de software do que sobre outras características e/ou atividades do desenvolvimento de software (Myers et al., 2004).

Apesar de ser impossível provar que um programa está correto por meio de testes – para praticamente qualquer programa (Myers et al., 2004) –, a sua utilização fornece evidências da conformidade com as funcionalidades especificadas e, desta forma, uma atividade de teste bem realizada aumenta a confiança no produto, além de auxiliar no entendimento dos artefatos testados (Lemos, 2005).

2.2.1 Terminologia e Conceitos Básicos

Dada a grande importância do teste de software, esforços vêm sendo feitos para padronizar algumas terminologias comumente utilizadas. O padrão IEEE 610.12-1990 (IEEE, 1990) define alguns termos e conceitos:

- Defeito (*fault*): passo, processo ou definição de dado incorreto, como por exemplo, uma instrução ou comando incorreto;

- Engano (*mistake*): ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador;

- Erro (*error*): diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e

- Falha (*failure*): produção de uma saída incorreta com relação à especificação. Um erro pode levar a uma falha.

De uma forma geral, os erros são classificados em: **erros computacionais** – o erro provoca uma computação incorreta mas o caminho executado (sequências de comandos) é igual ao caminho esperado; e **erros de domínio** — o caminho efetivamente executado é diferente do caminho esperado, ou seja, um caminho errado é selecionado (Maldonado et al., 2004).

Um ponto crucial na atividadade de teste é o projeto e a avaliação da qualidade de um conjunto de casos de teste utilizado para testar um determinado programa P. Um **caso de teste** consiste de um par ordenado $(d, S(d))$, no qual d é um elemento de um dado domínio $D(d \in D)$ e $S(d)$ é a saída esperada para uma dada função quando d é utilizado como entrada. Uma verificação completa de P poderia ser obtida testando P com um conjunto de casos de teste T que inclui todos os elementos do domínio. Entretanto, como geralmente o conjunto de elementos do domínio é infinito ou muito grande, torna-se praticamente impossível testar todos os elementos do domínio e, dessa forma, torna-se conveniente escolher um subconjunto para ser testado.

Além disso, a atividade de teste é permeada por uma série de limitações (Howden, 1986; Rapps e Weyuker, 1985). Em geral, os seguintes problemas são indecidíveis: dado dois programas, se eles são equivalentes; dados duas sequências de comandos (caminhos) de um programa, ou de programas diferentes, se eles computam a mesma função; e dado um caminho, se ele é executável ou não, ou seja, se existe um conjunto de dados de entrada que leva à execução desse caminho. Outra limitação fundamental é a correção coincidente – o programa pode apresentar, coincidentemente, um resultado correto para um item particular de um dado $d \in D$, ou seja, um particular item de dado ser executado, satisfazer a um requisito de teste e não revelar a presença de um erro.

Desta forma, para selecionar e avaliar conjuntos de casos de teste é fundamental a utilização de critérios de teste que auxiliam o testador e fornecem um método para a avaliação de conjuntos de casos de teste e uma base para a seleção de casos de teste. No primeiro caso os critérios de adequação servem para evidenciar a suficiência da atividade de teste e no segundo caso para ajudar na construção de casos de teste (Frankl e Weyuker, 2000).

2.3 Técnicas e Critérios de Teste

Como dito anteriormente, um programa em teste P deve ser executado com todos os elementos de $D(P)$ para que seja possível afirmar que nesse programa P não existem defeitos. No entanto, tal abordagem é infactível devido à cardinalidade de $D(P)$ (Delamaro et al., 2007).

Assim, é importante selecionar apenas um subconjunto reduzido de $D(P)$ para a execução de P, mas que tenha alta probabilidade de revelar a presença de defeitos caso eles existam. Com isso, surge a ideia de "subdomínios" de teste. Um **subdomínio** de $D(P)$ é um subconjunto do domínio de entrada que contém dados de teste que se comportam de maneira semelhante em um dado programa P.

O "teste de subdomínios" procura estabelecer quais são os subdomínios a serem utilizados e, então, selecionam-se os casos de teste de cada subdomínio. De acordo com o tipo de informação utilizada para estabelecer os subdomínios são definidas as **técnicas de teste**: funcional, estrutural e baseada em defeitos. Para identificar os subdomínios são estabelecidas certas "regras" para definir quando um dado de teste deve estar no mesmo subdomínio ou não. Tais regras são conhecidas como **critérios de teste**.

2.3.1 Teste Funcional

Segundo Delamaro et al. (2007), teste funcional é uma técnica utilizada para se projetarem casos de teste na qual o programa ou sistema é considerado uma caixa preta e, para testá-lo, são fornecidas entradas e avaliadas as saídas geradas para verificar se estão em conformidade com os objetivos especificados. Os detalhes de implementação quando se usa esta técnica não são considerados, e o software é avaliado segundo o ponto de vista do usuário.

O teste funcional envolve dois passos principais: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software (Pressman, 2005). Portanto, é de suma importância a qualidade das especificações de software para este tipo de teste, já que as funções em questão são identificadas a partir delas.

Uma das maneiras de testar um software utilizando essa técnica e, em principio, encontrar todos os defeitos, é submeter o programa a todas as entradas possíveis. Essa abordagem, chamada de **teste exaustivo**, é inviável, pois o domínio de entrada pode ser infinito ou muito extenso, levando a atividade de teste a consumir muito tempo. Por isso, utilizam-se critérios de teste, entre os quais, os mais conhecidos da técnica funcional são: particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito.

Vale ressaltar ainda que, como a técnica funcional e seus critérios de teste requerem a especificação do produto para derivar os requisitos de teste, é possível aplicá-los a qualquer programa, seja ele procedimental ou orientado a objetos, ou a componentes de software, uma vez que o código-fonte não é usado. Por outro lado, de acordo com Roper (1995), como os critérios funcionais se baseiam apenas na especificação, eles não podem assegurar que partes críticas e essenciais do código tenham sido cobertas.

Particionamento em classes de equivalência

Considerando-se que o teste exaustivo é, em geral, impossível de ser aplicado, durante a atividade de teste o testador fica restrito a usar um subconjunto de todas as possíveis entradas do programa, sendo que esse subconjunto deve ter uma grande probabilidade de encontrar erros (Myers et al., 2004).

Com o objetivo de apoiar a determinação desse subconjunto, o particionamento em classes de equivalência sugere particionar o domínio de entrada de um programa em um número finito de classes de equivalência de tal forma que se possa assumir que o teste de um valor representativo de cada classe é equivalente ao teste de qualquer outro valor da classe. Isto é, se um caso de teste pertencente a uma classe de equivalência detectar um erro, todos os outros casos de teste na mesma classe de equivalência devem encontrar o mesmo erro. Inversamente, se um caso de teste não detectar um erro, espera-se que nenhum outro caso de teste na mesma classe de equivalência também o detecte. Além disso, alguns autores consideram, além do domínio de entrada, também a partição do domínio de saída, identificando neste possíveis alternativas que poderiam determinar classes de equivalência no domínio de entrada (Copeland, 2003; Roper, 1995).

Segundo Myers et al. (2004), o projeto de casos de teste com a utilização do critério de particionamento em classes de equivalência é conduzido em dois passos:

1. Identificação das classes de equivalência: a partir da especificação do programa ou sistema, as classes de equivalência são identificadas tomando-se cada condição de entrada e particionando-as em classes de equivalência válidas e inválidas. Caso seja observado que as classes de equivalência se sobrepõem ou que os elementos de uma mesma classe não devem se comportar da mesma maneira, elas devem ser reduzidas a fim de separá-las e torná-las distintas (Delamaro et al., 2007).

2. Identificação dos casos de teste: uma vez identificadas as classes de equivalência, deve-se determinar os casos de teste, escolhendo-se um elemento de cada classe, de forma que cada novo caso de teste cubra o maior número de classes válidas possível. Para as classes inválidas devem ser gerados casos de teste exclusivos, uma vez que

um elemento de uma classe inválida pode mascarar a validação do elemento de outra classe inválida (Copeland, 2003).

De acordo com Delamaro et al. (2007), a força deste critério está na redução que ele possibilita no tamanho do domínio de entrada e na criação de dados de teste baseados unicamente na especificação, sendo adequado para aplicações em que as variáveis de entrada podem ser identificadas com facilidade e assumem valores específicos. No entanto, Delamaro et al. (2007) ressaltam que o critério não é tão facilmente aplicável quando o domínio de entrada é simples e seu processamento é complexo pois, embora a especificação possa sugerir que um grupo de dados seja processado de forma idêntica, na prática isso pode não acontecer. Além disso, a técnica não fornece diretrizes para a determinação dos dados de teste e para encontrar combinações entre eles que permitam cobrir as classes de equivalência de maneira mais eficiente (Roper, 1995).

Análise do valor limite

De acordo com Myers et al. (2004), a experiência mostra que casos de teste que exploram condições limites têm uma maior probabilidade de encontrar defeitos. Ou seja, os valores que estão exatamente sobre ou imediatamente acima ou abaixo dos limites das classes de equivalência têm maior probabilidade de revelar defeitos.

Sendo assim, a análise do valor limite é uma critério de projeto de casos de teste que completa o particionamento em classes de equivalência (Pressman, 2005). Mas ao invés de os dados de teste serem escolhidos aleatoriamente em uma classe de equivalência, eles devem ser selecionados para que os limites de cada classe de equivalência sejam explorados.

Além disso, segundo Myers et al. (2004), além da escolha seletiva dos dados de teste, o outro ponto que distingue este critério do critério de particionamento em classes de equivalência é a observação do domínio de saída.

De acordo com Pressman (2005), as diretrizes para a aplicação da análise do valor limite são semelhantes em muitos aspectos às fornecidas para o particionamento em classes de equivalência:

1. Se uma condição de entrada especifica um intervalo limitado pelos valores a e b, casos de teste devem ser projetados com os valores a e b, e imediatamente acima e imediatamente abaixo de a e b.

2. Se uma condição de entrada especifica vários valores, casos de teste devem ser desenvolvidos para exercitar os números mínimo e máximo. Valores imediatamente acima e imediatamente abaixo do mínimo e do máximo também são testados.

3. Aplique as diretrizes 1 e 2 às condições de saída.

4. Se as estruturas de dados internas do programa têm limites prescritos, certifique-se de projetar um caso de teste para exercitar a estrutura de dados no seu limite.

Este critério, como dito anteriormente, é muito similar ao critério de particionamento em classes de equivalência com relação a vantagens e desvantagens de uso. No entanto, segundo Myers et al. (2004), se aplicado corretamente, é um dos critérios mais úteis para o projeto de casos de teste.

Grafo causa-efeito

Uma das limitações dos critérios apresentados anteriormente é que eles não exploram combinações dos dados de entrada, pois o teste de combinações de entrada não é uma tarefa simples, já que o número de combinações geralmente é muito grande.

Para minimizar esta dificuldade, o critério Grafo causa-efeito define uma maneira sistemática de seleção de um conjunto de casos de teste que explora ambiguidades e incompletude nas especificações. Como forma de derivar os casos de teste, este critério utiliza um grafo que, segundo Myers et al. (2004), é uma linguagem formal na qual a especificação é traduzida. O processo de aplicação deste critério pode ser resumido nos seguintes passos:

1. Dividir a especificação do software em partes, pois a construção do grafo para grandes especificações torna-se bastante complexa.

2. Identificar as causas e efeitos na especificação. As causas correspondem às entradas, estímulos, ou qualquer evento que provoque uma resposta do sistema em teste e os efeitos correspondem às saídas, mudanças de estado do sistema ou qualquer resposta observável. Uma vez identificados, a cada um deve ser atribuído um único número.

3. Analisar a semântica da especificação e transformar em um grafo booleano — o Grafo causa-efeito — que liga as causas e os efeitos.

4. Adicionar anotações ao grafo, as quais descrevem combinações das causas e efeitos que são impossíveis por causa de restrições semânticas ou do ambiente.

5. Converter o grafo em uma tabela de decisão, na qual cada coluna representa um caso de teste.

6. Converter as colunas da tabela de decisão em casos de teste.

A utilização deste critério se torna complexa quando é necessário construir o grafo booleano para um número elevado de causas e efeitos. Além disso, a eficiência deste critério,

assim como dos outros critérios funcionais, depende da qualidade da especificação e da habilidade e experiência do testador. Entretanto, a grande vantagem deste critério é que ele exercita combinações de dados de teste que, possivelmente, não seriam considerados por outros critérios.

2.3.2 Teste Estrutural

A técnica estrutural (ou caixa branca) estabelece requisitos de teste com base em uma dada implementação, requerendo a execução de partes ou de componentes elementares do programa (Myers et al., 2004; Pressman, 2005). Ou seja, o teste estrutural baseia-se no conhecimento da estrutura interna do programa, e os aspectos de implementação são fundamentais para a geração/seleção dos casos de teste associados.

Em geral, a maioria dos critérios da técnica estrutural utiliza uma representação de programa conhecida como "Grafo de Fluxo de Controle" (GFC) ou "Grafo de Programa", que mostra o fluxo lógico do programa.

Um grafo de fluxo de controle $G = (N, E, s)$ é um grafo dirigido que consiste de um conjunto N de nós, um conjunto E de arestas dirigidas e um nó de entrada s. Os nós representam comandos ou uma coleção de comandos sequenciais (blocos de comandos, definido abaixo) e os arcos ou arestas representam o fluxo de controle. O grafo de fluxo de controle possui nós de entrada e nós de saída nos quais a computação começa e termina, respectivamente. Os nós de entrada não possuem nenhuma aresta de entrada, ou seja, não possuem antecessor. Por outro lado, os nós de saída não possuem arestas de saída, ou seja, não possuem sucessor (Zhu et al., 1997).

Um **bloco de comandos** (ou bloco de instruções) consiste em um conjunto de comandos de um determinado programa P, de modo que, quando o primeiro comando do bloco é executado, os outros comandos do mesmo bloco também são executados sequencialmente de acordo com a ordem estabelecida. Assim todos os comandos de um bloco têm um único predecessor e um único sucessor, com exceção do primeiro que não tem um predecessor e do último que não tem um sucessor. Além disso, o primeiro comando de um bloco é o único comando que pode ser executado depois da execução do último comando do bloco anterior. Cada bloco corresponde a um nó e a transferência de controle de um bloco para outro é representada por arestas dirigidas entre os nós (Rapps e Weyuker, 1985; Zhu et al., 1997).

Um **caminho** é uma sequência finita de nós $n_1, n_2, ..., n_k, k \geq 2$, tal que existe uma aresta de n_i para n_{i+1} para $i = 1, 2, ..., k-1$. Um **caminho completo** é um caminho no qual o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G (Rapps e Weyuker, 1985). Um caminho é um **caminho simples** se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos. Um **caminho**

independente é qualquer caminho ao longo do programa que introduz pelo menos um novo nó ou uma nova aresta. Um **caminho livre-de-laço** é um caminho em que todos os nós são distintos, ou seja, nenhum nó aparece mais que uma vez. Um caminho é um **caminho livre-de-iteração-de-laço** se ele não contém o mesmo nó mais que duas vezes (Linnenkugel e Mullerburg, 1990). No teste estrutural existem também os caminhos não executáveis. Um **caminho não executável** é um caminho do grafo de fluxo de controle impossível de ser coberto para qualquer elemento do domínio de entrada. Isso acontece quando as condições lógicas que deveriam ser satisfeitas para que a sequência de nós do caminho fosse executada são contraditórias (Howden, 1986).

Desta forma, a partir do grafo de fluxo de controle, os caminhos lógicos do software são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços bem como pares de definições e usos de variáveis. Os critérios pertencentes à técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados (Maldonado, 1991; Pressman, 2005; Weyuker, 1984).

A técnica estrutural apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes à atividade de teste que podem introduzir sérios problemas na automatização do processo de validação de software (Maldonado, 1991). Independentemente dessas desvantagens, essa técnica é vista como complementar às demais técnicas de teste existentes, uma vez que cobre classes distintas de defeitos (Myers et al., 2004; Pressman, 2005). Além disso, as informações obtidas pela aplicação de critérios estruturais têm sido consideradas relevantes para as atividades de manutenção, depuração e avaliação da confiabilidade de software (Delamaro et al., 2007).

Critérios baseados na complexidade

Os critérios baseados na complexidade utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. Um critério bastante conhecido dessa classe é o critério de McCabe (McCabe, 1976) ou teste de caminho básico, que utiliza a complexidade ciclomática do GFC para derivar os requisitos de teste.

Complexidade ciclomática é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto-base de um programa e fornece um limite superior para a quantidade de testes que deve ser conduzida para garantir que todos os comandos sejam executados pelo menos uma vez (Pressman, 2005).

A complexidade ciclomática tem fundamentação na teoria dos grafos e é calculada por uma das seguintes formas possíveis:

1. O número de regiões em um GFC. Uma região pode ser informalmente descrita como uma área incluída no plano do grafo. O número de regiões é computado contando-se todas as áreas delimitadas e a área não delimitada fora do grafo; ou

2. $V(G) = E - N + 2$, tal que E é o número de arestas e N é o número de nós do GFC G; ou

3. $V(G) = P + 1$, tal que P é o número de nós predicativos contido no GFC G.

Desta forma, a complexidade ciclomática é uma métrica útil para previsão dos módulos que provavelmente sejam propensos a erro. Ela pode ser usada tanto para o planejamento de teste quanto para o projeto de casos de teste.

Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias (Myers et al., 2004; Pressman, 2005). Os critérios mais conhecidos dessa classe são:

- **Todos-Nós:** exige que a execução do programa passe, ao menos uma vez, por cada vértice do GFC, ou seja, que cada comando do programa seja executado pelo menos uma vez;

- **Todas-Arestas:** requer que cada aresta do grafo, isto é, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez;

- **Todos-Caminhos:** requer que todos os caminhos possíveis do programa sejam exercitados.

É importante ressaltar, segundo Delamaro et al. (2007), que a cobertura do critério Todos-Nós é o mínimo esperado de uma boa atividade de teste, pois, do contrário, o programa testado é entregue sem a certeza de que todos os comandos presentes foram executados ao menos uma vez. Além disso, outro ponto a se destacar é que, apesar de desejável, a cobertura do critério Todos-Caminhos de um programa é, na maioria dos casos, uma tarefa impraticável por causa da quantidade de requisitos de teste gerados. Este problema foi uma das motivações para a introdução dos critérios baseados em fluxo de dados, discutidos a seguir.

Critérios baseados em fluxo de dados

Alguns critérios baseados em fluxo de dados utilizam a análise de fluxo de dados como fonte de informação para derivar os requisitos de teste. Estes critérios exploram as interações que envolvem definições de variáveis e referências a tais definições para estabelecer os requisitos de teste.

Para que fosse possível derivar os requisitos de teste exigidos por tais critérios é necessário estender o GFC para armazenar informações a respeito do fluxo de dados do programa. Esta extensão do GFC, proposta por Rapps e Weyuker (1982) e chamada de **Grafo Def-Uso** (*Def-Use Graph*), permite que sejam exploradas as interações que envolvem definições de variáveis e os subsequente usos dessas variáveis. A definição de uma variável ocorre toda vez que um valor é atribuído a ela (**def**). O uso de uma variável, por sua vez, pode ser de dois tipos: quando a variável é usada em uma computação, diz-se que seu uso é computacional (**c-uso**); quando a variável é usada em uma condição, diz-se que seu uso é predicativo (**p-uso**). Outro conceito importante é o **par def-uso**, que se refere a um par de definição e subsequente c-uso ou p-uso de uma variável. Um **caminho livre de definição** com relação a uma variável x do nó i ao nó j é um caminho $(i, n_1, ..., n_m, j)$ para $m \geq 0$, no qual não há definições de x nos nós $n_1, ..., n_m$.

Para que as informações de definição e uso das variáveis sejam adicionadas ao grafo Def-Uso, cada nó i do grafo é associado aos conjuntos c-uso e def, e cada aresta (i, j) ao conjunto p-uso. $def(i)$ é um conjunto de variáveis definidas no nó i; $c - uso(i)$ é um conjunto de variáveis para as quais existem c-usos em i e $p - uso(i, j)$ é um conjunto de variáveis para as quais existem p-uso na aresta (i, j). Definem-se ainda outros conjuntos necessários para a construção do critério def-uso (Rapps e Weyuker, 1982). Considere um nó i e uma variável x tal que $x \in def(i)$. Assim:

- $dcu(x, i)$ é o conjunto de todos os nós j tais que $x \in c-uso(j)$ e para os quais existe um caminho livre de definição com relação a x de i a j (Rapps e Weyuker, 1982);
- $dpu(x, i)$ é o conjunto de arestas (j, k) tais que $x \in p - uso(j, k)$ e para as quais existe um caminho livre de definição com relação a x de i a (j, k) (Rapps e Weyuker, 1982).

Seja P um conjunto de caminhos completos para um grafo Def-Uso de um programa. Diz-se que um nó i está incluído em P se P contém um caminho $(n_1, ..., n_m)$ tal que $i = n_j$ para algum j, $1 \leq j \leq m$. Similarmente, uma aresta (i_1, i_2) está incluída em P se P contém um caminho $(n_1, ..., n_m)$ tal que $i_1 = n_j$ e $i_2 = n_{j+1}$ para algum j, $1 \geq j \geq m-1$. Um caminho $(i_1, ..., i_k)$ está incluído em P se P contém um caminho $(n_1, ..., n_m)$ tal que $i_1 = n_j, i_2 = n_{j+1}, ..., i_k = n_{j+k-1}$ para algum j, $1 \geq j \geq m - k + 1$ (Rapps e Weyuker, 1982).

Considere G um grafo Def-Uso e P um conjunto de caminhos completos de G. Para o propósito deste trabalho, dentre os critérios baseados em fluxo de dados são definidos os seguintes:

- **Todas-Definições:** P satisfaz o critério Todas-Definições se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a algum elemento de $dcu(x, i)$ ou $dpu(x, i)$ (Rapps e Weyuker, 1982);

- **Todos-C-Usos:** P satisfaz o critério Todos-C-Usos se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a algum elemento de $dcu(x, i)$ (Rapps e Weyuker, 1982);

- **Todos-P-Usos:** P satisfaz o critério Todos-C-Usos se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a algum elemento de $dpu(x, i)$ (Rapps e Weyuker, 1982);

- **Todos-Usos:** P satisfaz o critério Todos-Usos se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui um caminho livre de definição com relação a x de i a cada elemento de $dcu(x, i)$ e a cada elemento de $dpu(x, i)$ (Rapps e Weyuker, 1982);

- **Todos-Caminhos-DU:** P satisfaz o critério Todos-Caminhos-DU se para cada nó i do grafo Def-Uso e para cada $x \in def(i)$, P inclui cada caminho livre-de-laço e livre de definição com relação a x de i a cada elemento de $dpu(x, i)$ e a cada elemento de $dcu(x, i)$ (Rapps e Weyuker, 1982).

2.3.3 Teste Baseado em Defeitos

A técnica de teste baseada em defeitos deriva os requisitos de teste a partir de informações sobre os enganos mais frequentes cometidos pelo programador ou projetista durante o processo de desenvolvimento de software (DeMillo et al., 1987). Os critérios típicos do teste baseados em defeitos são muito utilizados para medir a qualidade do conjunto de casos de teste de acordo com sua habilidade ou efetividade para detectar defeitos (Zhu et al., 1997). Entre eles pode-se citar:

- **Semeadura de defeitos:** uma quantidade conhecida de defeitos é semeada artificialmente no programa, ou seja, defeitos artificiais são introduzidos aleatoriamente no programa a ser testado e de modo desconhecido para o testador. Em seguida, o programa é testado e os defeitos naturais e artificiais descobertos são contabilizados separadamente. Usando estimativas de probabilidade, o número de defeitos naturais ainda existentes no programa pode ser calculado e é estatisticamente previsto

com máxima probabilidade de ser f/r, em que f é o número de defeitos naturais encontrados no teste e r é a taxa do número de defeitos artificiais encontrados do total de defeitos artificiais semeados (Budd, 1981; Zhu et al., 1997).

- **Análise de mutantes:** utiliza um programa P e um conjunto de casos de teste T gerado. O primeiro passo neste critério é a construção de uma coleção de programas alternativos (**mutantes**) que diferem ligeiramente do original, ou seja, são utilizados defeitos típicos do processo de implementação de software para alterar o programa P e derivar os requisitos de teste. Cada mutante é executado para cada caso de teste do conjunto T. Essa execução termina quando um elemento de T revela que P e o programa mutante produzem diferentes saídas ou quando T é consumido completamente. No primeiro caso diz-se que o mutante morreu e, no outro caso, diz-se que o mutante ainda está vivo (DeMillo et al., 1987; Zhu et al., 1997).

2.4 Fases da Atividade de Teste

A atividade de teste de software envolve basicamente quatro etapas: planejamento do teste, projeto dos casos de teste, execução do teste e avaliação dos resultados do teste. Os primeiros testes planejados e executados geralmente concentram-se nas unidades, individualmente. À medida que o teste progride, o próximo passo é tentar encontrar defeitos em conjuntos integrados de unidades. Por fim, testa-se todo o sistema (Pressman, 2005). Assim, pode-se dizer que a atividade de teste é realizada em três fases: teste de unidade, teste de integração e teste de sistema.

O **teste de unidade** concentra esforços na menor unidade do software com o objetivo de identificar defeitos de lógica e de implementação em cada unidade isoladamente. Uma unidade refere-se à menor parte funcional de um programa que pode ser executada, como por exemplo, uma sub-rotina, um procedimento ou um método.

O **teste de integração** é uma atividade sistemática aplicada durante a integração da estrutura do programa visando a descobrir defeitos associados às interfaces entre as unidades; o objetivo é, a partir das unidades testadas na fase de teste de unidade, construir a estrutura de programa que foi determinada pelo projeto.

O **teste de sistema** verifica a integração de todos os elementos que compõem o sistema e o seu funcionamento com a finalidade de assegurar que o sistema e os demais elementos que o compõem (por exemplo, hardware e banco de dados) se combinam adequadamente e que o resultado e o desempenho global desejado é obtido. O teste de sistema consiste de uma série de testes de diferentes tipos, tais como: testes de recuperação, teste de segurança, teste de estresse e teste de desempenho.

2.5 Considerações Finais

Neste capítulo foi apresentada uma introdução à atividade de teste de software, dando enfoque à técnica de teste estrutural e seus critérios. Inicialmente os fundamentos do teste de software foram discutidos. Em seguida foram apresentados alguns conceitos básicos para o entendimento das fases, técnicas e critérios de teste, além da definição das terminologias de teste de software utilizadas ao longo deste trabalho. Foram apresentadas as técnicas de teste funcional (com os critérios de particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito), teste estrutural (com os critérios baseados em complexidade, fluxo de controle e fluxo de dados), e a baseada em defeitos (com os critérios de semeadura de defeitos e análise de mutantes), sendo que exemplos de aplicação foram mostrados para as duas primeiras técnicas. Além disso, foram brevemente abordadas as fases da atividade de teste que são: unidade, integração e sistema. Este capítulo serve como fundamentação para a proposta de trabalho apresentada nesta monografia.

Capítulo

3

Linguagens de Programação OO e OA

3.1 Considerações Iniciais

Neste capítulo é apresentada um revisão bibliográfica do paradigma de programação orientada a objetos (POO) e de sua extensão, a programação orientada a aspectos (POA). Os principais conceitos de POO e da linguagem Java são apresentados na Seção 3.2. Na Seção 3.3 são apresentados os fundamentos da POA e os principais mecanismos da linguagem AspectJ relevantes para esta monografia.

3.2 Programação Orientada a Objetos e Java

As primeiras linguagens de programação, as linguagens montadoras, somente possibilitavam que os programas fossem escritos em um nível muito baixo de abstração que se baseava em instruções de máquina, dificultando o entendimento e a depuração dos programas. Um grande avanço ocorreu com o surgimento das primeiras linguagens de alto nível, quando os programas passaram a ser representados como um conjunto de variáveis e um conjunto de procedimentos que manipulam essas variáveis. Essas linguagens são conhecidas como linguagens de programação procedimentais.

Apesar das linguagens procedimentais permitirem a representação de tipos abstratos de dados (TADs), que são abstrações de mais alto nível de variáveis e procedimentos, elas possuem algumas deficiências, como por exemplo: se dois TADs são similares, mas não

idênticos, não existe um modo de expressar essa similaridade nessas linguagens. Essa deficiência foi parcialmente corrigida pelas linguagens de programação orientadas a objetos, que permitem que similaridades entre tipos abstratos de dados sejam expressas por meio de relações de herança (Capretz, 2003).

No final da década de 60, a linguagem Simula67 introduziu conceitos hoje encontrados nas linguagens orientadas a objetos. Em meados de 1970, o Centro de Pesquisa da Xerox (PARC) desenvolveu a linguagem Smalltalk, a primeira totalmente orientada a objetos. No início da década de 80, a AT&T lançou a Linguagem C++, uma evolução da linguagem C em direção à orientação a objetos. Atualmente, a grande maioria das linguagens incorpora características de orientação a objetos, como por exemplo, a linguagem Java.

A ideia básica da POO é simular a realidade, criando abstrações do mundo real, com o objetivo de representar as características relevantes dos objetos envolvidos no sistema que tenta simular. Esses objetos podem enviar e receber mensagens e reagir a mensagens recebidas. Assim, a POO introduz uma abordagem em que o programa é visto como uma coleção de objetos que se relacionam (Kung et al., 1995). A POO tem como alicerces cinco conceitos-chave:

- **Classe:** Entidade caracterizada por um conjunto de dados e também pelas operações que podem ser realizadas sobre esses dados. Aos elementos do conjunto de dados dá-se o nome de atributo e às operações dá-se o nome de métodos.

- **Objeto:** Instância de uma classe criada em tempo de execução. Cada objeto tem seu próprio valor para cada atributo, mas compartilha os nomes de atributos e métodos com outras instâncias da mesma classe.

- **Encapsulamento:** Separação das partes internas e externas do objeto de tal forma que os dados internos sejam acessados ou modificados somente pela interface provida pelo encapsulamento, permitindo a proteção e ocultação da estrutura interna de cada objeto.

- **Herança:** Mecanismo pelo qual uma classe pode estender outra classe. A classe criada (subclasse) herda todas as variáveis e métodos da classe já existente (superclasse), além de poder definir suas propriedades específicas. Essa característica permite criar uma hierarquia de classes, em que os níveis mais altos possuirão as características comuns a todos os objetos da classe hierarquicamente superior, enquanto nos níveis inferiores estarão especializações das classes superiores (ou superclasses).

- **Polimorfismo:** É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assi-

natura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. A decisão sobre qual método deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, por meio do mecanismo de ligação tardia (*late binding* ou *run-time binding*).

3.2.1 Java

A linguagem de programação Java foi desenvolvida pela Sun Microsystems no início da década de 90. Ela pode ser caracterizada como: orientada a objetos, distribuída, independente de plataforma e multitarefa (Microsystems, 2009). A seguir são apresentados alguns conceitos e terminologias utilizadas nessa linguagem [1].

Tipos Primitivos e de Referência

Os tipos da linguagem Java são divididos em duas categorias: tipos primitivos e tipos de referência. Os tipos primitivos incluem: um tipo booleano, um tipo caractere, quatro tipos inteiros e dois tipos de ponto flutuante. A Tabela 3.1 resume os tipos de dados primitivos.

Tabela 3.1: Os tipos primitivos da linguagem Java.

Tipo	Conteúdo	Valor *default*	Tamanho
`boolean`	`true` ou `false`	`false`	1 bit
`char`	caractere Unicode	\u0000	16 bits
`byte`	Inteiro sinalizado	0	8 bits
`short`	Inteiro sinalizado	0	16 bits
`int`	Inteiro sinalizado	0	32 bits
`long`	Inteiro sinalizado	0	64 bits
`float`	Ponto flutuante IEEE 754	0.0	32 bits
`double`	Ponto flutuante IEEE 754	0.0	64 bits

Quanto aos tipos de referência, existem três categorias, todas de 32 bits: de array, de classe e de interface. Os três tipos possuem valores que são referências para objetos criados dinamicamente, ou seja: um valor do tipo classe referencia uma instância de classe; um valor do tipo array referencia um array; por fim, um valor do tipo interface referencia uma instância de classe que implementa a interface. Um outro valor de referência é o valor `null`, que indica que a variável não referencia nenhum objeto e também é utilizado como valor padrão quando uma variável de referência não recebe nenhum valor inicial.

Além dos tipos de dados apresentados anteriormente, Java possui um tipo de dado para trabalhar com textos que é o tipo `String`. Por ser uma classe, o tipo `String` não é considerado um tipo primitivo da linguagem.

[1] A maior parte desta seção foi baseada na dissertação de Franchin (2007).

Literais, Expressões e Sentenças

Literais são valores que aparecem diretamente no código Java. Eles incluem números inteiros e pontos flutuantes, caracteres com aspas simples, strings de caracteres com aspas duplas e as palavras reservadas `true`, `false` e `null`.

Uma expressão é uma estrutura em Java que pode ser simples ou complexa. Uma expressão é dita simples quando consiste apenas de literais e variáveis. Uma expressão é considerada complexa quando utiliza operadores para combinar expressões simples.

Uma sentença (*statement*) ou instrução é um comando único executado pelo interpretador Java. O interpretador executa uma sentença por vez, seguindo a ordem em que foi escrita. Entretanto, existem algumas sentenças conhecidas como sentenças de fluxo de controle, tais como condicionais (`if` e `switch`) e laços (`for`, `while` e `do-while`) que alteram a ordem de execução normal.

Classes e Objetos

Uma classe é um modelo usado para definir vários objetos com características semelhantes. Um programa é constituído de uma classe ou de um conjunto de classes. Os elementos básicos de uma classe são chamados de membros da classe e podem ser divididos em duas categorias:

- **Atributos:** especificam o estado da classe.
- **Métodos:** especificam os mecanismos pelos quais a classe ou objetos da classe podem operar.

Um objeto é uma instância de uma classe criado em tempo de execução. Para que seja possível acessar os atributos e os métodos de um objeto, é preciso atribuir uma referência ao objeto. A declaração e a criação de um novo objeto da instância da classe <ClassName>, que será referenciado pela variável <VariableRef> segue a seguinte sintaxe:

```
<ClassName> <VariableRef> = new <ClassName>(<ParamList>);
```

Algumas palavras reservadas servem para ampliar ou restringir o acesso a um membro de uma classe. Elas são acrescentadas na declaração do membro da classe e funcionam como especificadoras de acesso. A linguagem Java possui os seguintes especificadores de acesso: `public`, `private` e `protected`, que são discutidos na Tabela 3.2. A omissão do especificador na declaração define um membro `default`[2].

[2] `default` não é uma palavra reservada. É simplesmente um nome dado para um nível de acesso quando nenhum especificador de acesso é informado.

Declaração de Classe

A sintaxe para a declaração de classes é a seguinte:

```
<Modifier> class <ClassName> [extends <SuperClassName>] [implements <InterfaceName>]{
   ...
}
```

Tabela 3.2: Os especificadores de acesso da linguagem Java.

Especificador de acesso	Descrição
`public`	o membro é acessível dentro da classe em que foi declarado e também em todas as classes de quaisquer pacotes da aplicação.
`private`	o membro é acessível somente dentro da classe em que foi declarado.
`protected`	o membro é acessível dentro da classe em que foi declarado, em todas as classes que estão no mesmo pacote em que a sua classe foi definida e em todas as subclasses que estejam ou não no mesmo pacote.
`default`	o membro é acessível tanto dentro da classe em que foi declarado como em todas as classes que estão no mesmo pacote em que a sua classe foi definida.

A declaração de uma classe inicia, opcionalmente, com a inclusão de modificadores (<Modifier>), mostrados na Tabela 3.3. Em seguida é inserida a palavra reservada `class` seguida de um identificador <ClassName>.

As classes que incluem o modificador `abstract` na sua declaração são chamadas de classes abstratas. Toda classe abstrata possui ao menos um método abstrato, ou seja, um método que não possui implementação. As classes abstratas são consideradas classes incompletas, pois dependem da definição de uma classe derivada.‘

A declaração de uma classe pode incluir a palavra reservada `extends` sucedida de outro identificador (<SuperClassName>). Isso cria uma relação de herança entre as classes, ou seja, a classe que está sendo declarada é derivada de uma outra classe que a precede na relação de herança. Além disso, pode ocorrer a inclusão da palavra reservada `implements` seguida de um identificador (<InterfaceName>) ou mais identificadores. Isso indica que a classe que está sendo declarada implementa a(s) interface(s) incluída(s) na sua declaração.

Declaração de Atributos

Uma classe pode possuir dois tipos de atributos: atributos estáticos (ou atributos de classe) e atributos de instância. Um atributo estático é associado a uma classe. Já um atributo de instância é associado a uma instância da classe. Por padrão da linguagem, um atributo é declarado como de instância. Para declarar um atributo estático, acrescenta-se a palavra reservada `static`. A sintaxe para a declaração de atributos é a seguinte:

```
<AccessSpecifier> <Modifier> <Type> <FieldName> [= <Expression>]
```

Tabela 3.3: Os modificadores da linguagem Java.

Modificador	Uso	Descrição
`abstract`	Classe	define classes abstratas.
	Método	utilizado quando se deseja declarar um método sem, contudo, especificar seu corpo. Isto funciona como uma espécie de lembrete para que alguma classe derivada complete a declaração fornecendo um corpo.
`final`	Atributo	faz com que o atributo seja considerado uma constante. Após a atribuição inicial, seu valor não poderá ser alterado.
	Classe	define uma classe que não pode ser derivada, isto é, não pode ser superclasse de nenhuma subclasse.
	Método	faz com que não seja permitida a sobrecarga ou sobrescrita do método.
`native`	Método	utilizado para designar métodos implementados em outras linguagens, como por exemplo, C++.
`static`	Atributo	define um atributo estático.
	Método	define um método estático.
`synchronized`	Método	utilizado para impedir que dois ou mais métodos que executam concorrentemente acessem os dados de uma classe ao mesmo tempo.
`transient`	Atributo	utilizado para evitar que um certo atributo seja salvo durante uma operação de serialização de objetos (*object serialization*).
`volatile`	Atributo	avisa ao compilador que múltiplas linhas de execução (*multi-threading*) podem acessar o atributo, que é conhecido como volátil.

A declaração de um atributo inicia pela indicação do tipo de dado (<Type>) que pode ser do tipo primitivo ou de referência, seguido pelo identificador do atributo (<FieldName>) e sucedida opcionalmente por uma expressão (<Expression>) que é utilizada para atribuir um valor inicial para o atributo. A declaração pode ser precedida por uma especificação de acesso (<AccessSpecifier>) e por modificadores (<Modifier>). Os especificadores de acesso são apresentados na Tabela 3.2. Quanto aos modificadores, eles podem ser: `final`, `volatile`, `static` ou `transient` e são discutidos na Tabela 3.3.

Declaração de Métodos

Uma classe pode possuir dois tipos de métodos: métodos estáticos (ou métodos de classe) e métodos de instância. Um método estático está associado com a classe e só pode acessar os atributos estáticos da classe. Já um método de instância está associado com a instância da classe e pode acessar tanto os atributos de instância quanto os estáticos declarados na classe. Por padrão, um método é declarado como de instância. Para declarar um método estático, adiciona-se a palavra reservada `static` na declaração. A sintaxe de declaração de métodos é a seguinte:

```
<AccessSpecifier> <Modifier> <ReturnType> <MethodName> (<ParamList>) [throws exceptions]
```

```
{
   ...
}
```

A declaração de um método inicia-se com a informação do tipo de dado de retorno (<ReturnType>), que pode ser um tipo primitio ou de referência, ou ainda pode ser `void` (situação em que o método não retorna valor algum). Em seguida, deve-se indicar o nome do método (<MethodName>) sucedido por uma lista de parâmetros (<ParamList>) separados por vírgulas, e colocados entre parênteses. Tal lista também pode ser nula. A declaração pode ser precedida por uma especificação de acesso (<AccessSpecifier>) ou de modificadores (<Modifier>). Os especificadores de acesso são apresentados na Tabela 3.2. Quanto aos modificadores, eles podem ser: `final`, `abstract`, `static`, `native` ou `transient` e são descritos na Tabela 3.3.

Um **construtor** é um tipo especial de método. Os construtores possuem o mesmo nome da classe em que são definidos e determinam como um objeto é iniciado quando ele é criado. Toda classe em Java tem ao menos um construtor. Se não houver uma definição explícita de um construtor na classe, Java fornece um construtor padrão que não tem parâmetros e que executa uma iniciação padrão do objeto.

Em uma mesma classe podem existir métodos que possuem o mesmo nome e o mesmo tipo de retorno, mas que diferem pelo número e/ou pelos tipos dos parâmetros. Esta técnica é chamada **sobrecarga** (*overloading*) de métodos. A decisão sobre qual deles é executado depende do número e/ou dos tipos dos argumentos passados na chamada do método e pode ser tomada em tempo de compilação (*static binding* ou *early binding*).

Quando uma classe define um método de instância usando o mesmo nome, mesmo tipo de retorno, igual número e mesmos tipos de parâmetros de um método presente em sua superclasse, esse método sobrescreve o método da superclasse. Esta técnica é chamada **sobrescrita** (*overriding*) de métodos. Na sobrescrita, pode ocorrer o caso do compilador não ter como decidir qual método será executado se o método foi sobrescrito em outras classes—afinal, um objeto de uma classe derivada pode estar sendo referenciado como sendo um objeto da superclasse. Se este for o caso, o método que deve ser selecionado é o da classe derivada e não o da superclasse. Dessa forma, a decisão sobre qual dos métodos deve ser selecionado, feita de acordo com o tipo do objeto, só pode ser tomada em tempo de execução, pelo mecanismo de **ligação tardia** (*late binding* ou *run-time binding*).

Interface

Interface é um tipo de referência, similar à classe, que contém assinaturas, mas não implementações, de um conjunto de métodos e também definições de constantes. Uma interface define um protocolo de comunicação entre dois objetos. Quando uma classe implementa

uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe.

Herança

Herança é uma maneira de derivar uma classe a partir de outra classe, ou seja, uma subclasse herda métodos e campos de uma superclasse.

Com exceção da classe `Object`, uma classe tem exatamente uma superclasse direta. Uma classe herda campos e métodos de todas as suas superclasses, direta ou indiretamente. A subclasse pode sobrepor métodos que herda, ou ele pode ocultar campos ou métodos que herda.

A classe `Object` é o topo da hierarquia de classes. Todas as classes são descendentes desta classe e métodos são herdados a partir dela. Alguns métodos úteis herdados da classe `Object` são: `toString()`, `equals()`, `clone()` e `getClass()`.

Uma classe pode ser derivada de qualquer outra classe que permita isso. Pode-se impedir que uma classe seja herdada usando a palavra-chave final na declaração da classe. Da mesma forma, é possível impedir que um método seja herdado declarando-o como um método final.

Com relação a classes abstratas, elas só podem ser herdadas, e nunca instanciadas. Uma classe abstrata pode conter somente métodos abstratos e não implementados. As subclasses, então, devem implementar esses métodos abstratos.

Exceções

Uma exceção é um evento que ocorre durante a execução de um programa que interrompe o fluxo normal das instruções. Quando um erro ocorre em um método, ele cria um objeto que é lançado (*throwing*) ao sistema em tempo de execução. O objeto chamado objeto de exceção (*Exception Object*), contém informações sobre o erro, incluindo o seu tipo, o estado do programa e quando o erro ocorreu.

Um programa pode capturar exceções usando uma combinação dos blocos `try`, `catch` e `finally`. O bloco `try` identifica o bloco de código em que uma exceção pode ocorrer. O bloco `catch` identifica um bloco de código conhecido como manipulador de exceção e trata uma exceção. Por fim, o bloco `finally` identifica um bloco de código que sempre é executado mesmo que uma exceção seja lançada, sendo muito útil, por exemplo, para fechar arquivos e recuperar recursos.

Ambiente de Programação Java

Em Java, é importante distinguir quatro tecnologias inter-relacionadas: a Linguagem de Programação Java, o formato de arquivo class do Java, a Java API (Application Programming Interface) e a máquina Virtual Java. A linguagem de programação Java é a linguagem em que as aplicações Java são escritas. Diferentemente de outras linguagens de programação que compilam ou interpretam um programa para que ele seja executado, na linguagem Java os programas são compilados e interpretados. O compilador inicialmente transforma o código-fonte do programa em arquivos class. Esses arquivos são interpretados pela Máquina Virtual Java (JVM). Na execução do programa, alguns recursos do sistema, tais como entrada e saída, que são implementados nas classes da Java API, são acessados. Na Figura 3.1 o ambiente de programação Java é ilustrado.

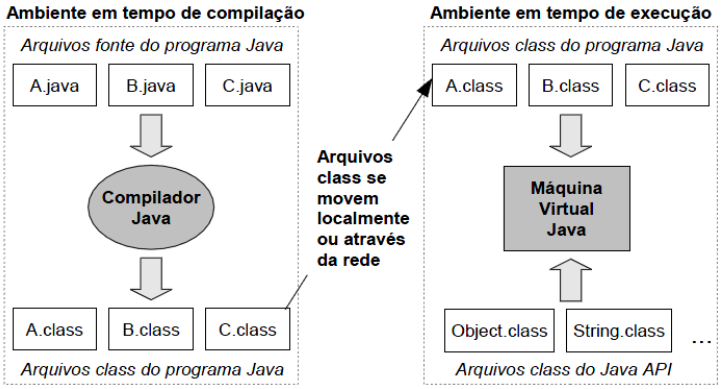


Figura 3.1: O ambiente de programação Java (adaptado de Venners (1999)).

Máquina Virtual Java (JVM)

Um programa fonte escrito na linguagem Java é traduzido pelo compilador para *bytecodes*, isto é, o código de máquina de um processador virtual, chamado Máquina Virtual Java (JVM). A JVM é um programa capaz de interpretar os *bytecodes* produzidos pelo compilador. Com isto, um programa Java pode ser executado em qualquer plataforma, desde que ela tenha uma JVM instalada. A vantagem dessa técnica é evidente: garantir uma maior portabilidade para os programas Java em código-fonte e compilados. A Figura 3.2 mostra um esquema visual do compilador e interpretador Java.

3.3 Programação Orientada a Aspectos e AspectJ

A divisão em partes é importante para se reduzir a complexidade. É muito difícil para um ser humano compreender um sistema de grande porte se este for monolítico e não tiver fronteiras claras que definem suas funções.

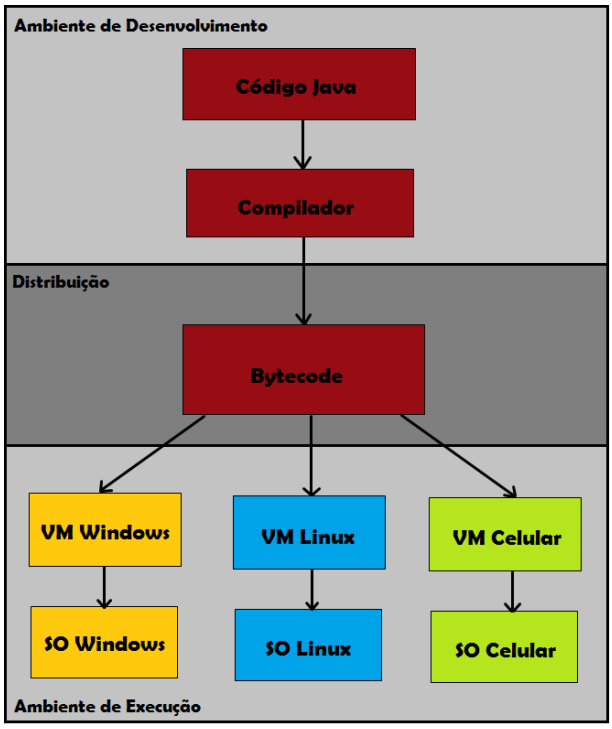


Figura 3.2: Compilador e interpretador Java.

Em um projeto de software – para fins de simplicidade, legibilidade, e consequente facilidade na manutenção e maior potencial para reutilização – é importante que os vários interesses relevantes para um sistema estejam localizados em módulos separados. A evolução das linguagens de programação a partir de construções de baixo nível — código de máquina e linguagens de montagem — até abordagens de alto nível — programação procedimental, programação lógica, programação funcional e programação orientada a objetos — tem contribuído para se obter uma melhor **separação dos interesses** de um sistema com a divisão de um domínio de conhecimento em partes menores, para poder entender objetivamente cada uma delas, porém cada linguagem à sua maneira (utilizando sub-rotinas, procedimentos, funções, classes, APIs) e em graus diferentes (Elrad et al., 2001).

Em meados dos anos 90, alguns pesquisadores constataram a existência de certos interesses que, independentemente da técnica de programação utilizada ou da maneira como o sistema venha a ser decomposto, não se encaixam em módulos individuais, mas ficam espalhados por várias unidades do software. Esses interesses são chamados de **interesses transversais**, pois afetam naturalmente classes e/ou métodos que modularizam outros interesses. Dois prolemas costumam ser provocados: o **espalhamento**, que acontece quando um interesse encontra-se em diversos módulos do sistema; e o **entrelaçamento**, que acontece quando há a implementação de diversos interesses dentro de um único módulo (Lemos, 2005).

Um exemplo de interesse transversal é apresentado na Figura 3.3. O diagrama apresentado refere-se a um editor de figuras simples. Uma `Figura` consiste de `ElementoDeFigura`, que pode ser `Ponto` ou `Linha`. Há também uma `Tela` onde as figuras são desenhadas. Quando um elemento de figura se mover é necessário que a tela seja atualizada. Uma possível solução utilizando orientação a objetos seria chamar o método de atualização da Tela (`Tela.atualiza()`) no do corpo dos métodos `setX`, `setY` e `move` da classe `Ponto` e no corpo dos métodos `setP1`, `setP2` e `move` da classe `Linha`, conforme apresentado na Figura 3.4. Observe que o interesse de atualização de tela não pertence a nenhuma das duas classes, mas entrecorta ambas e, por isso, pode ser entendido como transversal às classes.

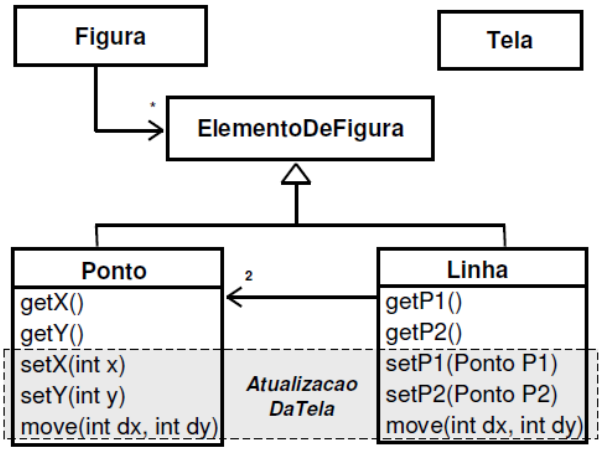


Figura 3.3: Interesse ortogonal de atualização de tela (adaptado do trabalho de Elrad et al. (2001)).

Com o intuito de resolver os problemas causados, foi proposta a programação orientada a aspectos (POA), uma tecnologia de suporte para a implementação dos interesses transversais de maneira localizada. A POA é resultado de vários estudos em metaprogramação, programação orientada a assuntos, filtros de composição e programação adaptativa (Czarnecki et al., 2000). O termo **aspecto** refere-se aos interesses transversais que podem ser implementados em módulos separados (Kiczales et al., 2001a).

A POA oferece mecanismos para que os aspectos possam ser construídos em módulos separados e oferece meios para a definição de pontos do programa onde esses aspectos possam definir comportamento. A partir daí, um programa executável pode ser gerado, combinando os módulos básicos com os aspectos. Dessa forma, a POA pretende dar suporte aos interesses transversais assim como a POO tem dado suporte aos objetos (Kiczales et al., 2001a).

```
class Ponto implements ElementoDeFigura {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Tela.atualiza();
  }

  void setY(int y) {
    this.y = y;
    Tela.atualiza();
  }

  void move(int dx, int dy) {
    x += dx; y += dy;
    Tela.atualiza();
  }
}
```

```
class Linha implements ElementoDeFigura {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Tela.atualiza();
  }

  void setP2(Point p2) {
    this.p2 = p2;
    Tela.atualiza();
  }

  void move(int dx, int dy) {
    p1.move(dx,dy); p2.move(dx,dy);
    Tela.atualiza();
  }
}
```

Figura 3.4: Solução OO para o exemplo de atualização de tela.

3.3.1 AspectJ

A linguagem AspectJ é uma extensão de Java criada para permitir a programação orientada a aspectos de maneira genérica, no contexto dessa linguagem. Foi desenvolvida no *Xerox Palo Alto Research Center* e posteriormente foi agregada ao projeto *Eclipse* da IBM. Basicamente, as novas construções de AspectJ consistem em: conjuntos de junção (*pointcut*) que identificam conjuntos de pontos de junção; adendos (*advice*) que definem o comportamento em um dado conjunto de junção; construções para afetar estaticamente a estrutura dos módulos básicos do programa (declarações inter-tipos e que alteram a hierarquia de classes); e os aspectos (*aspect*) que encapsulam as construções novas e as tradicionais de uma classe Java [3].

Na Figura 3.5 é mostrada uma implementação em AspectJ para o exemplo de interesse transversal apresentado anteriormente. Essa implementação é utilizada ao longo da seção. O aspecto `AtualizacaoDaTela` se encarrega de atualizar a visualização todas as vezes que alguma figura é alterada. Como as figuras são alteradas a partir dos métodos que alteram suas coordenadas, os adendos são definidos nas chamadas a esses métodos.

Pontos de Junção

Os pontos de junção são "pontos" bem definidos na execução do fluxo de programa. Eles podem ser uma chamada de método, a execução de um construtor, ou até mesmo o acesso a um atributo de uma classe específica (Gradecki e Lesiecki, 2003). Na Tabela 3.4 são listadas as categorias de pontos de junção definidas na linguagem AspectJ.

[3] A maior parte desta seção foi baseada no livro de Laddad (2003)

```
01. public class Ponto implements ElementoDeFigura {
02.   private int x = 0, y = 0;
03.   int getX() { return x; }
04.   int getY() { return y; }
05.   void setX(int x) { this.x = x; }
06.   void setY(int y) { this.y = y; }
07.   void move(int dx, int dy) { x += dx; y += dy; }
08. }
09.
10. public class Linha implements ElementoDeFigura {
11.   private Ponto p1, p2;
12.   Ponto getP1() { return p1; }
13.   Ponto getP2() { return p2; }
14.   void setP1(Ponto p1) { this.p1 = p1; }
15.   void setP2(Ponto p2) { this.p2 = p2; }
16.   void move(int dx, int dy) { p1.move(dx,dy); p2.move(dx,dy); }
17. }
18.
19. public aspect AtualizacaoDaTela {
20.   public pointcut mudancaEstado() :
21.   call(void ElementoDeFigura+.move(int,int)) ||
22.   call(* Ponto.set*(*)) || call(* Linha.set*(*));
23.
24.   before(): mudancaEstado() {
25.     System.out.println("O estado da figura vai mudar...");
26.   }
27.
28.   after() returning: mudancaEstado() {
29.     Tela.atualiza();
30.   }
31. }
```

Figura 3.5: Exemplo de código em AspectJ.

Tabela 3.4: Categorias de pontos de junção do AspectJ (Kiczales et al., 2001b).

Categoria	Descrição
Chamada de método / Chamada de construtor	Um método ou um construtor de uma classe é chamado. Pontos de junção deste tipo encontram-se no objeto chamador ou possuem valor nulo (se a chamada é feita a partir de um método estático).
Execução de método / Execução de construtor	Um método ou construtor é executado. Pontos de junção deste tipo ocorrem no objeto chamado, dentro do contexto do método.
Leitura de atributo	Um atributo de um objeto, classe ou interface é lido.
Escrita de atributo	Um atributo de objeto ou classe é escrito.
Execução de tratador de exceção	Um tratador de exceção é invocado.
Iniciação de classe	Iniciadores estáticos de uma classe (se existirem) são executados.
Iniciação de objeto	Iniciadores dinâmicos de uma classe (se existirem) são executados durante a criação do objeto, abrangendo desde o retorno da chamada ao construtor da classe pai até o retorno do primeiro construtor chamado.
Pré-inicialização de objeto	Pré-iniciadores de uma classe são executados, abrangendo desde a chamada ao primeiro construtor até a chamada ao construtor da classe pai.

Conjuntos de Junção

Os conjuntos de junção são utilizados para identificar pontos de junção no fluxo do programa. A partir da especificação desses pontos, regras de combinação podem ser definidas, tal como realizar uma certa ação antes ou depois do pontos de junção. Além disso, os

conjuntos de junção podem expor as informações de contexto dos pontos de junção alcançados, podendo ser utilizadas pelas ações que afetam esses pontos. No AspectJ, os conjuntos de junção podem conter nomes ou serem anônimos. Na Figura 3.6 é mostrado um exemplo de conjunto de junção nomeado. Um conjunto de junção pode ser definido como uma combinação de conjuntos de junção, utilizando operadores lógicos binários 'e' (&&) e 'ou' (|| – como no exemplo). Além disso, o operador unário de negação (!) também pode ser usado quando não se quer capturar pontos de junção definidos por um conjunto de junção específico. O primeiro conjunto de junção que compõe `mudancaEstado`, por exemplo, captura todas as chamadas ao método `move` da classe `ElementoDeFigura`, que recebe dois inteiros como parâmetros e não retorna nenhum valor.

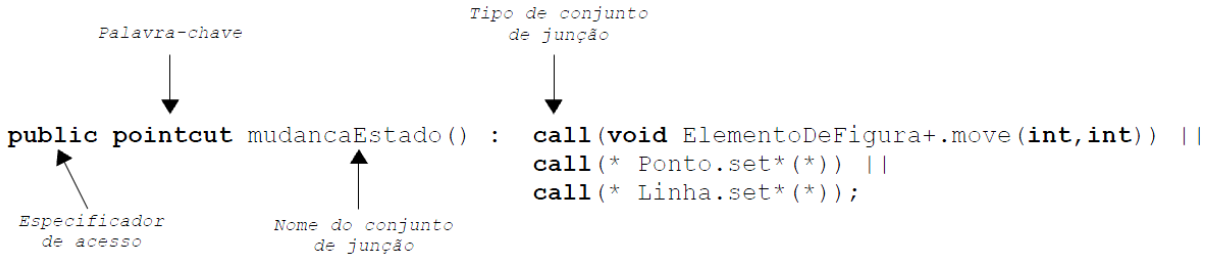


Figura 3.6: Destaque do conjunto de junção nomeado, definido no exemplo apresentado na Figura 3.5 nas linhas 20-22 (adaptado do livro de Laddad (2003)).

Existem vários tipos de conjuntos de junção em AspectJ que se diferenciam pelo modo como capturam pontos de junção em um programa. Um dos tipos de conjuntos de junção é o que captura pontos de junção baseado na categoria a que eles pertencem. Alguns mapeamentos entre as categorias de pontos de junção (Tabela 3.4) e a sintaxe dos conjuntos de junção correspondente são apresentados na Tabela 3.5.

Tabela 3.5: Mapeamento entre as categorias de pontos de junção e a sintaxe dos conjuntos de junção correspondente.

Categoria de Pontos de Junção	Sintaxe do Conjunto de Junção
Execução de método	`execution`(AssinaturadeMétodo)
Chamada a método	`call`(AssinaturadeMétodo)
Execução de construtor	`execution`(AssinaturadeConstrutor)
Chamada a construtor	`call`(AssinaturadeConstrutor)
Iniciação de classe	`staticinitialization`(AssinaturadeTipo)
Acesso de leitura de atributo	`get`(AssinaturadeAtributo)
Acesso de modificação de atributo	`set`(AssinaturadeAtributo)
Execução de tratador de exceção	`handler`(AssinaturadeTipo)
Iniciação de objeto	`initialization`(AssinaturadeConstrutor)
Pré-iniciação de objeto	`preinitialization`(AssinaturadeConstrutor)

Adendos

Adendos são construções similares a métodos e definem comportamentos que devem ocorrer em um dado ponto de junção. Existem três tipos de adendos no AspectJ:

- Anteriores (`before()`), que executam antes do ponto de junção;
- Posteriores (`after()`), que executam depois do ponto de junção;
- De contorno (`around()`), que têm a habilidade de executar em substituição ao ponto de junção, podendo também continuar com a execução normal ou causar a execução com o contexto alterado, por meio do método especial `proceed()`.

No exemplo da Figura 3.5 são mostrados dois adendos, um anterior e outro posterior, que executam antes e depois dos pontos de junção capturados pelo conjunto de junção `mudancaEstado`, respectivamente. O primeiro apenas imprime uma mensagem avisando que o estado está para mudar e o segundo atualiza a tela de fato, depois dos elementos de figura terem sido alterados.

Algumas diferenças entre métodos e adendos no AspectJ são:

- adendos não têm nome;
- adendos não podem ser chamados diretamente (é trabalho do sistema executá-los nos momentos apropriados);
- adendos não têm especificadores de acesso.

Modificações Estáticas

Modificações estáticas são construções que implementam o entrecorte estático em AspectJ. Enquanto o entrecorte dinâmico modifica o comportamento de execução do programa, o entrecorte estático altera a estrutura estática dos tipos — classe, interfaces e outros aspectos — e seus comportamentos em tempo de compilação. Existem os seguintes tipos de entrecorte estático: declarações inter-tipos, modificações da hierarquia de classes, e introdução de avisos e erros de compilação.

As declarações inter-tipos permitem que atributos, construtores e métodos sejam adicionados às classes básicas do programa. A modificação na hierarquia das classes é feita por meio da declaração inter-tipo `declare parents`. Essa declaração permite alterar a hierarquia e indicar que alguma classe herda de outra classe ou implementa certa interface. Com a introdução dos avisos e erros de compilação, pode-se implementar comportamentos similares às diretivas `#error` e `#warning` de alguns pré-processadores C/C++. A declaração `declare error` permite declarar um erro que interrompe a compilação e mostra uma

mensagem toda a vez que o compilador encontra um dado ponto de junção (capturado por um conjunto de junção). A construção `declare warning` funciona de maneira similar, porém a compilação não pára e somente a mensagem é mostrada.

Aspectos

Aspectos são unidades modulares que implementam os interesses transversais. A declaração de um aspecto é muito similar à declaração de uma classe em Java. Ela pode incluir declarações de conjuntos de junção, declarações de adendos e todos os outros tipos de declarações permitidas na declaração de uma classe, tais como declarações de atributos e métodos (Kiczales et al., 2001b). Assim como classes, aspectos contêm privilégios de acesso, podem herdar de outras classes (ou de aspectos abstratos) e implementar interfaces.

Em AspectJ existem três tipos de instanciação de aspectos: `Singleton`, `Per-object` e `Per-control-flow`. Neste trabalho é tratado somente o primeiro tipo. `Singleton` é o tipo padrão atribuído a todos os aspectos. Um aspecto deste tipo possui uma única instância que é criada automaticamente quando o programa é executado e é usada durante toda a aplicação. É no contexto dessa única instância que os adendos do aspecto são executados (Gradecki e Lesiecki, 2003).

3.4 Considerações Finais

Neste capítulo foram apresentadas a programação orientada a objetos e orientada a aspectos. Foi discutido como o paradigma de programação OO possibilita a construção de sistemas com uma arquitetura mais organizada na qual os interesses do sistema são implementados em módulos individuais (classes). Porém, certos interesses (interesses transversais), não se encaixam em módulos individuais e ficam espalhados por várias unidades do software. A programação orientada a aspectos (POA) tenta resolver o problema de espalhamento e entrelaçamento de código por meio de uma construção que implementa os interesses transversais em módulos separados (aspecto, na linguagem AspectJ). Além disso, foi apresentada uma breve introdução à linguagem Java e à linguagem AspectJ, que é uma extensão da linguagem Java criada para permitir a programação orientada a aspectos.

Capítulo

4

Teste de Programas OO e OA

4.1 Considerações Iniciais

A POO e a POA acrescentaram novas construções e conceitos aos já conhecidos das linguagens de programação tradicionais e que, portanto, devem ser explorados em abordagens de teste adequadas para esses contextos. Neste capítulo são apresentadas tanto as aplicações diretas das abordagens de teste que foram propostas para o paradigma procedimental, quanto algumas adaptações necessárias para a aplicação dessas técnicas de teste nesses novos contextos. Ênfase é dada à técnica estrutural (baseada em fluxo de controle e de dados). Na Seção 4.2 são apresentadas as fases do teste de software OO e OA. Na seção 4.3 são apresentados trabalhos sobre teste de unidade e teste de integração de programas OO. Por fim, na Seção 4.4 são apresentados alguns trabalhos que abordam o teste de programas OA.

4.2 Fases do Teste de Software OO e OA

Com base nos conceitos de teste apresentados no Capítulo 2, pode-se considerar, apesar de não haver um consenso, que em programas OO as menores unidades a serem testadas são os métodos; e em programas OA, considerando-os como extensões de programas OO (como acontece em AspectJ), os métodos (inclusive os introduzidos por aspectos) e os adendos.

Por definição, uma classe engloba um conjunto de atributos e métodos que manipulam esses atributos e um aspecto engloba basicamente conjuntos de atributos, métodos, adendos e conjuntos de junção. Assim, considerando uma única classe ou um único aspecto já é possível pensar em teste de integração. Métodos da mesma classe ou mesmo aspecto, bem como adendos e métodos de um mesmo aspecto, podem interagir para desempenhar funções específicas, caracterizando uma integração que deve ser testada.

Levando em conta tais considerações, e baseando-se no livro de Sommerville (2007) e na abordagem de Harrold e Rothermel (1994), a atividade de teste de programas OO e OA pode ser particionada nas seguintes fases (Lemos et al., 2004):

1. **Teste de Unidade:** O teste de cada método e adendo isoladamente, também chamado de teste intra-método ou intra-adendo.

2. **Teste de Módulo:** O teste de uma coleção de unidades dependentes – unidades que interagem por meio de chamadas ou interações com adendos. Esta fase pode ser dividida nos seguintes tipos de teste (considerando classes e aspectos como entidades diferentes):

   - Intermétodo: Consiste em testar cada método público juntamente com outros métodos da mesma classe chamados direta ou indiretamente (chamadas indiretas são aquelas que ocorrem fora do escopo do próprio método, dentro de um método chamado em qualquer profundidade) (Harrold e Rothermel, 1994).
   - Adendo-método: Consiste em testar cada adendo juntamente com outros métodos chamados por ele direta ou indiretamente.
   - Método-adendo: Consiste em testar cada método público juntamente com os adendos que o afetam direta ou indiretamente (considerando que um adendo pode afetar outro adendo). Nesse tipo de teste não é considerada a integração dos métodos afetados com os outros métodos chamados por eles, nem com método chamados pelos adendos.
   - Adendo-adendo: Consiste em testar cada adendo juntamente com outros adendos que o afetam direta ou indiretamente.
   - Intermétodo-adendo: Consiste em testar cada método público juntamente com os adendos que o afetam direta e indiretamente, e com métodos chamados direta ou indiretamente. Este tipo de teste inclui os quatro primeiros tipos de teste descritos acima.
   - Intraclasse: Consiste em testar as interações entre os métodos públicos de uma classe quando chamados em diferentes sequências, considerando ou não a interação com os aspectos (Harrold e Rothermel, 1994).

- Interclasse: Consiste em testar as interações entre classes diferentes, considerando ou não a interação dos aspectos.

3. **Teste de Sistema:** A integração de todos os módulos forma um subsistema ou um sistema completo. Para esta fase geralmente é utilizado o teste funcional.

4.3 Teste de Programas Orientados a Objetos

Muitas das abordagens de teste estrutural de programas orientados a objetos foram adaptadas de abordagens anteriormente propostas para o paradigma procedimental. Entretanto, o paradigma OO apresenta novos desafios para o teste de software. Um programa OO é composto de classes e de objetos, características não encontradas no paradigma procedimental. Além disso, são necessárias abordagens especiais para lidar com herança, polimorfismo e acoplamento dinâmico (*dynamic binding*). A partir daí, algumas abordagens de teste estrutural foram propostas.

Uma das primeiras abordagens de teste estrutural de programas OO foi apresentada por Harrold e Rothermel (1994). Essa abordagem apoia o teste estrutural baseado em fluxo de dados para todos os tipos de interações de fluxo em uma classe. Ela considera quatro níveis de teste: intramétodo, intermétodo, intraclasse e interclasse. Muitas das abordagens para teste estrutural OO baseiam-se nesses níveis de teste. Em seu trabalho, Harrold e Rothermel consideram a classe como a menor unidade a ser testada no teste de unidade de um programa OO. Entretanto, outros autores como Domingues (2001) e Vincenzi (2004) consideram os métodos como sendo a menor unidade a ser testada. Na Tabela 4.1 são sintetizados os tipos de teste de software OO que podem ser aplicados em cada uma das fases de teste considerando o método ou a classe como sendo a menor unidade.

Tabela 4.1: Relação entre as fases de teste e o teste de programas OO (adaptada de Domingues (2001)).

Menor Unidade	**Fase**	**Nível**
Método	Unidade	Intramétodo
	Integração	Intermétodo, Intraclasse e Interclasse
	Sistema	Toda a aplicação
Classe	Unidade	Intramétodo, Intermétodo e Intraclasse
	Integração	Interclasse
	Sistema	Toda a aplicação

Harrold e Rothermel definiram o grafo de chamadas de classe (*class call graph*) para representar a estrutura de chamadas de métodos em uma classe e também as chamadas que podem ser feitas de fora da classe. Nesse grafo os nós representam os métodos e as arestas representam chamadas entre os métodos. A implementação parcial em C++ da classe

`SymbolTable` mostrada na Figura 4.1 é utilizada durante toda esta seção para explicar a proposta desses pesquisadores. O grafo de chamadas para esta classe é apresentado na Figura 4.2. As linhas pontilhadas representam chamadas externas (provenientes de métodos de outras classes) a métodos públicos da classe `SymbolTable`.

```
01 // symboltable.h: definition
02 #include "symbol.h"
03
04 class SymbolTable {
05 private:
06   TableEntry *table;
07   int numentries, tablemax;
08   int *Lookup(char *);
09 public:
10   SymbolTable(int n) {
11      tablemax = n;
12      numentries = 0;
13      table = new TableEntry[tablemax]; };
14   SymbolTable() { delete table; };
15   int AddtoTable(char *symbol, char *syminfo);
16   int GetfromTable(char *symbol, char *syminfo);
17 };
18
19 // symboltable.c: implementation
20 #include "symboltable.h"
21
22 int SymbolTable::Lookup(char *key, int index) {
23   int saveindex;
24   int Hash(char *);
25   saveindex = index = Hash(key);
26   while (strcmp(GetSymbol(index),key) != 0) {
27     index++;
28     if (index == tablemax) /* wrap around */
29       index = 0;
30     if (GetSymbol(index)==0 || index==saveindex)
31       return NOTFOUND;
32   }
33   return FOUND;
34 }
35
```

```
36 int SymbolTable::AddtoTable(char *symbol,
                               char *syminfo) {
37   int index;
38   if (numentries < tablemax) {
39     if (Lookup(symbol,index) == FOUND)
40       return NOTOK;
41     AddSymbol(symbol,index);
42     AddInfo(syminfo,index);
43     numentries++;
44     return OX;
45   }
46   return NOTOK;
47 }
48
49 int SymbolTable::GetfromTable(char *symbol,
                                 char **syminfo) {
50   int index;
51   if (Lookup(symbol,index) == NOTFOUND)
52     return NOTOK;
53   *syminfo = GetInfo(index);
54   return OK;
55 }
56
69 void SymbolTable::AddInfo(syminfo,index)
70   ...
73   strcpy(table[index].syminfo,syminfo);
75 }
76
77 char *SymbolTable::GetInfo(index)
78   ...
82   return table[index].syminfo;
83 }
```

Figura 4.1: Implementação parcial da classe `SymbolTable` (Harrold e Rothermel, 1994).

A seguir são discutidas algumas abordagens de teste estrutural de unidade e de integração para programas OO. Toma-se como base a abordagem de Harrold e Rothermel (1994), porém considerando o método como sendo a menor unidade de teste. Assim, o teste intramétodo (discutido na Seção 4.3.1) é considerado um teste de unidade OO, e os testes intermétodo, intraclasse e interclasse (discutidos na Seção 4.3.2) são considerados testes de integração OO.

4.3.1 Teste Estrutural de Unidade de Programas OO

A primeira abordagem de teste estrutural de unidade para programas OO proposta é o teste intramétodo (Harrold e Rothermel, 1994) cuja finalidade é testar cada método individualmente. Para ilustrar o teste intramétodo, considere o código-fonte da Figura 4.1. Neste exemplo, o teste intramétodo é realizado em cada um dos dez métodos da classe `SymbolTable` separadamente. Para avaliar relações de fluxo de dados intramétodo em programas OO, Harrold e Rothermel utilizam o par definição-uso (d, u). Considerando

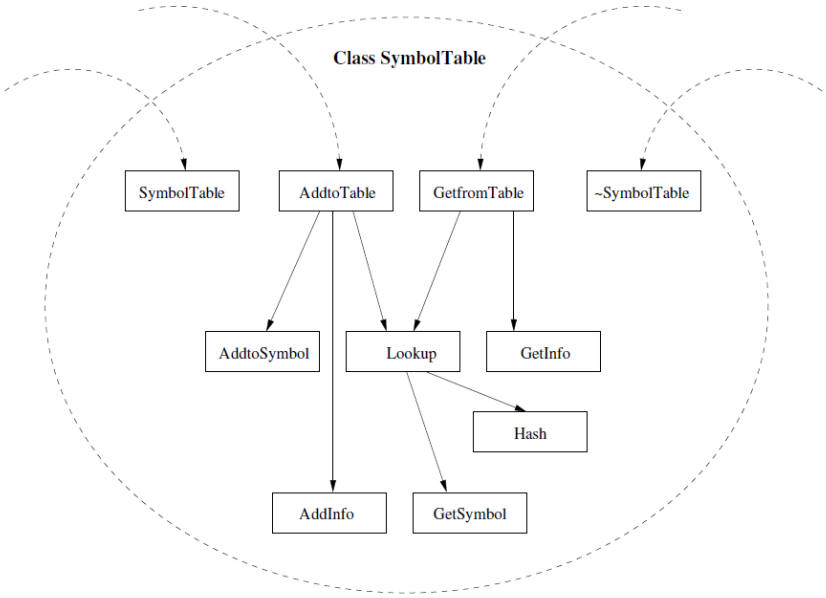


Figura 4.2: Grafo de chamadas de classe para a classe `SymbolTable` (Harrold e Rothermel, 1994).

uma classe C, uma sentença d que define uma variável e uma sentença u que contém um uso da variável, define-se:

- **Par Def-Uso Intramétodo:** Seja M um método de C. Se d e u estão em M e existe um programa P que chama M, tal que (d, u) é um par def-uso exercitado durante uma simples invocação de M, então (d, u) é um par def-uso intramétodo.

Analisando o código-fonte da Figura 4.1, um exemplo de par def-uso intramétodo ocorre no método `Lookup` com relação à variável `index`, que é definida na linha 27 e usada na linha 28.

Uma outra abordagem de teste estrutural de programas OO é proposta por Vincenzi (2004). O pesquisador apresenta o teste de fluxo de controle e fluxo de dados intramétodo considerando o método como a menor unidade do programa. Em seu trabalho, Vincenzi utiliza o grafo de fluxo de dados ($\mathcal{DUG}$) para representar o fluxo de controle e o fluxo de dados intramétodo tanto durante a execução normal do programa quanto na presença de exceções. Além disso, com base no grafo $\mathcal{DUG}$, o pesquisador revisou os critérios baseados em fluxo de controle e em fluxo de dados e definiu alguns critérios para serem aplicados em programas OO.

De acordo com a abordagem adotada pelo pesquisador, antes de se construir o grafo $\mathcal{DUG}$, constrói-se o que é chamado de Grafo de Fluxo de Dados de Instrução (*Data-Flow Instruction Graph* — $\mathcal{IG}$) de cada método. Formalmente, o grafo $\mathcal{IG}$ de um método m é definido como um grafo dirigido $\mathcal{IG}\ (m) = (NI, EI, si, TI)$ no qual:

- NI representa o conjunto não vazio de nós de um grafo $\mathcal{IG}$: $NI = \{\ n_i | n_i$ corresponde a uma instrução de *bytecode* i, para toda instrução de *bytecode* alcançavel i de $m\ \}$.
- $si \in NI$ é nó de entrada, ou seja, corresponde ao nó que contém a primeira instrução do método m.
- $TI \subseteq NI$ é o conjunto de nós de saída.
- $EI = EI_r \cup EI_e$ é o conjunto completo de arestas de $\mathcal{IG}$, no qual:
  - EI_r e EI_e correspondem a dois subconjuntos disjuntos de arestas regulares e de exceção, respectivamente;
    - EI_e é o conjunto de arestas de exceção definido como $EI_e = \{(n_i, n_j) |$ a instrução de n_i está no escopo de um tratador de exceção que inicia na instrução de $n_j\}$.
    - EI_r é o conjunto de arestas regulares definido como $EI_r = \{(n_i, n_j) |$ a instrução de n_j pode ser executada imediatamente após a instrução em n_i e $(n_i, n_j) \notin EI_e\}$.

Uma vez que $\mathcal{IG}$, representando o fluxo de controle de determinado método, tenha sido computado é necessário inserir nele informações de fluxo de dados. Para isso Vincenzi definiu um modelo de fluxo de dados que indica quais instruções de *bytecode* são responsáveis pela definição e uso de variáveis e como variáveis de referência e vetores (*arrays*) devem ser considerados. Além disso, foram feitas algumas suposições para identificação de definições e usos de variáveis agregadas, atributos de instância, atributos estáticos e variáveis usadas em uma chamada de método (parâmetros ou argumentos).

Baseado nas suposições para identificação de definições e usos de variáveis e nas diferentes classes de instruções de *bytecode*, o grafo $\mathcal{IG}$ de um dado método é percorrido e a cada nó do grafo é associado um conjunto $def(i)$ que representa o conjunto de variáveis definidas naquele nó e um conjunto $uso(i)$ que representa o conjunto de variáveis usadas naquele nó.

O $\mathcal{IG}$ oferece uma maneira prática de percorrer o conjunto de instruções de uma dada unidade, identificando definições e usos de variáveis. Entretanto, o número de nós e arestas envolvidos nesse tipo de grafo pode ser muito grande. Dessa maneira, constrói-se o grafo Def-Uso ($\mathcal{DUG}$) baseando-se no conceito de bloco de instruções (discutido na Seção 2.3.2).

O grafo $\mathcal{DUG}$ representa o modelo base que é utilizado para se derivar requisitos de teste de fluxo de controle e de dados intramétodo. Um grafo $\mathcal{DUG}$ de um dado método m é definido como um grafo dirigido $\mathcal{DUG}\ (m) = (N, E, s, T)$ no qual cada nó $n \in N$ representa um bloco de instruções:

- N representa o conjunto de nós de um grafo $\mathcal{DUG}$: $N = \{n | n$ corresponde a um bloco de instruções de *bytecode* de $m\}$; I_n é a n-upla ordenada de instruções agrupadas no nó n;
- $s \in N$ é o nó de entrada de m;
- $T \subseteq N$ é o conjunto (possivelmente vazio) de nós de saída.
- $E = E_e \cup E_r$ é o conjunto completo de arestas de $\mathcal{DUG}$. Seja $\mathcal{IG}\ (m) = (NI, EI, si, TI)$, tem-se:
  - E_e é o conjunto de arestas de exceção definido como $E_e = \{(n_i, n_j) |$ existe em $\mathcal{IG}\ (m)$ uma aresta de exceção do último elemento de I_{n_i} para o primeiro elemento de $I_{n_j}\}$.
  - E_r é o conjunto de arestas regulares definido como $E_e = \{(n_i, n_j) |$ existe em $\mathcal{IG}\ (m)$ uma aresta regular do último elemento de I_{n_i} para o primeiro elemento de $I_{n_j}\}$.

A Figura 4.3 ilustra como os conjuntos de variáveis definidas e usadas de um grafo $\mathcal{IG}$ são mapeados para os conjuntos $def(i)$ e $uso(i)$ de um grafo $\mathcal{DUG}$. O código-fonte de um método simples `dummy` e o seu respectivo *bytecode* são mostrados nas Figuras 4.3(a) e (b), respectivamente. A Figura 4.3(c) traz a correspondência entre as linhas de código-fonte e as linhas do *bytecode*. O grafo $\mathcal{IG}$ correspondente ao conjunto de instruções de *bytecode* da Figura 4.3(b) é apresentado na Figura 4.3(d), com seus respectivos conjuntos de variáveis definidas ($def(i)$) e usadas ($uso(i)$) em cada nó, e, por fim, a Figura 4.3(e) mostra o grafo $\mathcal{DUG}$ do método `dummy()`.

Vincenzi também definiu alguns critérios de teste baseados em fluxo de controle e em fluxo de dados para serem aplicados no contexto de programas OO. Considerando o teste de fluxo de controle, o pesquisador definiu os critérios de teste: **todos-nós-independentes-de-exceção**, **todos-nós-dependentes-de-exceção**, **todas-arestas-independentes-de-exceção** e **todas-arestas-dependentes-de-exceção**. Em relação aos critérios de fluxo de dados, Vincenzi definiu os critérios de teste: **todos-usos-independentes-de-exceção** e **todos-usos-dependentes-de-exceção**.

4.3.2 Teste Estrutural de Integração de Programas OO

Harrold e Rothermel (1994) propuseram a primeira abordagem de teste estrutural de integração para programas OO. Essa abordagem considera os testes: intermétodo, intraclasse e interclasse. No teste **intermétodo**, os métodos são testados em conjunto com outros métodos da mesma classe. O teste **intraclasse** tem como finalidade testar a interação

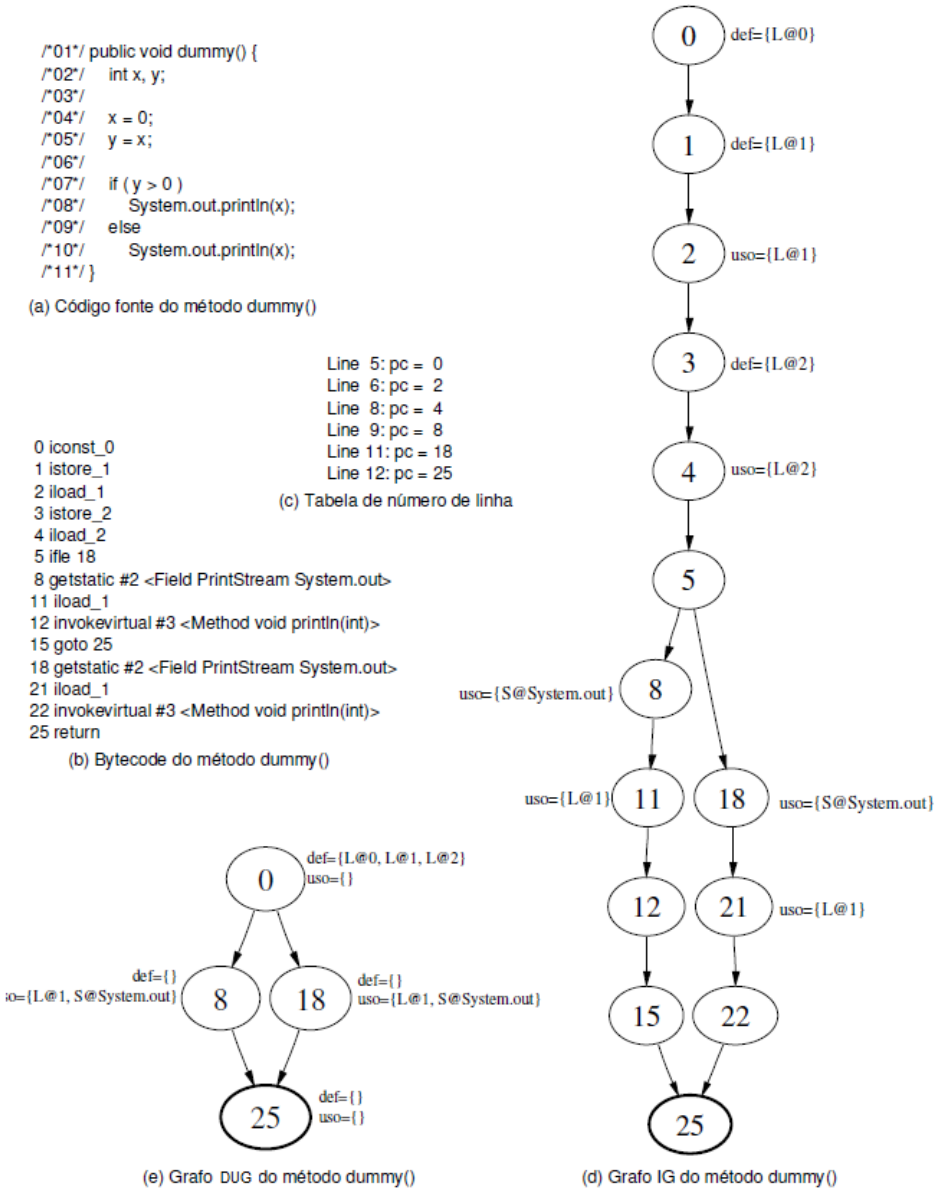


Figura 4.3: Exemplo dos grafos $\mathcal{IG}$ e $\mathcal{DUG}$ (adaptado de Vincenzi (2004)).

entre métodos públicos declarados em uma mesma classe quando chamados em diferentes sequências. Por fim, o teste **interclasse** é responsável por testar a interação entre métodos públicos declarados em classes distintas quando chamados em diferentes sequências.

Para os níveis de teste intermétodo e intraclasse, Harrold e Rothermel utilizam o par definição-uso (d, u) que permite avaliar relações de fluxo de dados em programas OO nesses níveis. Considerando uma classe C, uma sentença d que define uma variável e uma sentença u que contém um uso da variável, define-se:

- **Par Def-Uso Intermétodo:** Seja M_0 um método público de C e seja $\{M_1, M_2, \cdots, M_n\}$ o conjunto de métodos chamados direta ou indiretamente quando M_0 é invocado. Suponha que d está em M_i e que u está em M_j, sendo que tanto M_i quanto

M_j estão em $\{M_1, M_2, \cdots, M_n\}$. Se existe um programa P que chama M_0 tal que, em P, (d, u) é um par def-uso exercitado durante uma simples invocação de M_0 por P, e $M_i \neq M_j$ ou M_i e M_j são invocações separadas do mesmo método, então (d, u) é um par def-uso intermétodo.

- **Par Def-Uso Intraclasse:** Seja M_0 um método público de C e seja $\{M_1, M_2, \cdots, M_n\}$ o conjunto de métodos chamados direta ou indiretamente quando M_0 é invocado. Seja N_0 um método público de C e seja $\{N_1, N_2, \cdots, N_n\}$ o conjunto de métodos chamados direta ou indiretamente quando N_0 é invocado. Suponha que d está em algum dos métodos em $\{M_1, M_2, \cdots, M_n\}$ e u em algum dos métodos em $\{N_1, N_2, \cdots, N_n\}$. Se existe um programa P que chama M_0 e N_0, tal que em P, (d, u) é um par def-uso e a chamada a M_0 encerra após a execução de d e antes da execução de u, então (d, u) é um par def-uso intraclasse.

Analisando o código-fonte da Figura 4.1, o par (29,41) é um exemplo de par def-uso intermétodo porque o método `AddTable` chama o método `Lookup` passando `index` como parâmetro. O valor de `index` é definido na linha 29 (método `Lookup`). No retorno, a variável `index` é usada na chamada ao método `AddSymbol`. Como exemplo de par def-uso intraclasse considere a sequência <`AddtoTable`, `GetfromTable`>. O método `AddtoTable` pode adicionar uma informação de símbolo na tabela chamando o método `AddInfo`. O método `GetfromTable` pode acessar a informação da tabela por meio do método `GetInfo`. Assim, a definição de uma informação na tabela (linha 73) e uso da tabela (linha 82) gera um par def-uso intraclasse.

Harrold e Rothermel argumentam que para os testes intermétodo e intramétodo (mostrado na Seção 4.3.1) pode ser utilizado o algoritmo de Pande, Landi e Ryder (PLR) (Pande et al., 1994), concebido para computar os pares def-uso intra e interprocedimentais para a linguagem C. Porém, para a computação dos pares def-uso intraclasse, o algoritmo não pode ser utilizado diretamente e, para isso, é gerado o **grafo de fluxo de controle de classe** ($\mathcal{CCFG}$- *Class Control Flow Graph*), que conecta todos os métodos de uma classe.

Para construir o $\mathcal{CCFG}$, primeiramente é necessário gerar o grafo de chamada, que representa as chamadas entre métodos e também as chamadas que podem ser feitas de fora da classe – os nós representam cada método e as arestas representam as chamadas. A partir daí, para poder ser utilizado o algoritmo PLR, é necessário englobar o grafo de chamadas com um *frame*[1] para permitir que sejam feitas as chamadas aos métodos em quaisquer sequências para a computação dos pares def-uso intraclasse. O *frame* funciona

[1]Harrold e Rothermel (1994) definem o *frame* como uma abstração de um método principal (*main*) P, no qual chamadas aos métodos públicos são selecionadas arbitrariamente por um comando de escolha (*switch*) S, em que S é englobado por um laço L.

como uma espécie de unidade pseudo-controladora da classe e possui cinco nós: *frame entry*, *frame exit*, que representam a entrada e saída do *frame*; *frame loop*, que facilita o sequenciamento dos métodos; *frame call* e *frame return*, que representam a chamada e o retorno a qualquer método público que possa ser invocado. O frame ainda possui quatro arestas: (*frame entry*, *frame loop*), (*frame loop*, *frame call*), (*frame loop*, *frame exit*) e (*frame return*, *frame loop*). A partir daí, cada nó que representa um método no grafo de chamadas é substituído pelo seu grafo de fluxo de controle.

Na Figura 4.4 é mostrado o $\mathcal{CCFG}$ parcial da classe `SymbolTable`. Nessa figura apenas são expandidos os nós correspondentes aos métodos `AddtoTable`, `GetFromTable` e `Lookup` com os grafos de fluxo de controle de cada um deles, adicionando também arestas referentes às chamadas de um método a outros da mesma classe. Os Cs e Rs contidos na figura correspondem às chamadas e retornos de métodos, respectivamente.

Para efetuar teste de integração entre classes, Harrold e Rothermel definem um quarto nível de teste de fluxo de dados chamado teste interclasse. O teste interclasse considera pares def-uso (d, u), tais que a definição ocorre em um método M_1 de uma classe C_1 e o uso ocorre em um outro método M_2 de uma classe C_2. Para derivar os pares def-uso deve-se construir os $\mathcal{CCFG}$ das classes envolvidas[2].

4.4 Teste de Programas Orientados a Aspectos

As técnicas modernas de desenvolvimento de software podem auxiliar na produção de sistemas com maior qualidade, a partir de projetos constituídos de módulos mais coesos e menos acoplados, com código mais legível e de alto nível. Entretanto, por mais avançadas que sejam, essas técnicas não podem evitar que defeitos sejam introduzidos ao longo do desenvolvimento, já que essa atividade está sujeita a falhas humanas. De fato, novas técnicas de desenvolvimento podem inclusive introduzir novos tipos de defeitos, já que mecanismos que apoiam um maior poder de expressividade podem ser mal utilizados, fazendo com que essas técnicas se tornem obstáculo ao invés de auxílio (Lemos, 2009).

Existem evidências de que o cenário não é diferente com a POA. Segundo Alexander et al. (2004), os motivos que tornam o processo de teste em programas OA mais difíceis de serem efetuados são:

- Aspectos não têm identidade ou existência independente, sendo dependentes do contexto de outras classes;

[2]Harrold e Rothermel (1994) não apresentaram uma definição formal para pares Def-Uso interclasse conforme apresentada para outros níveis de teste.

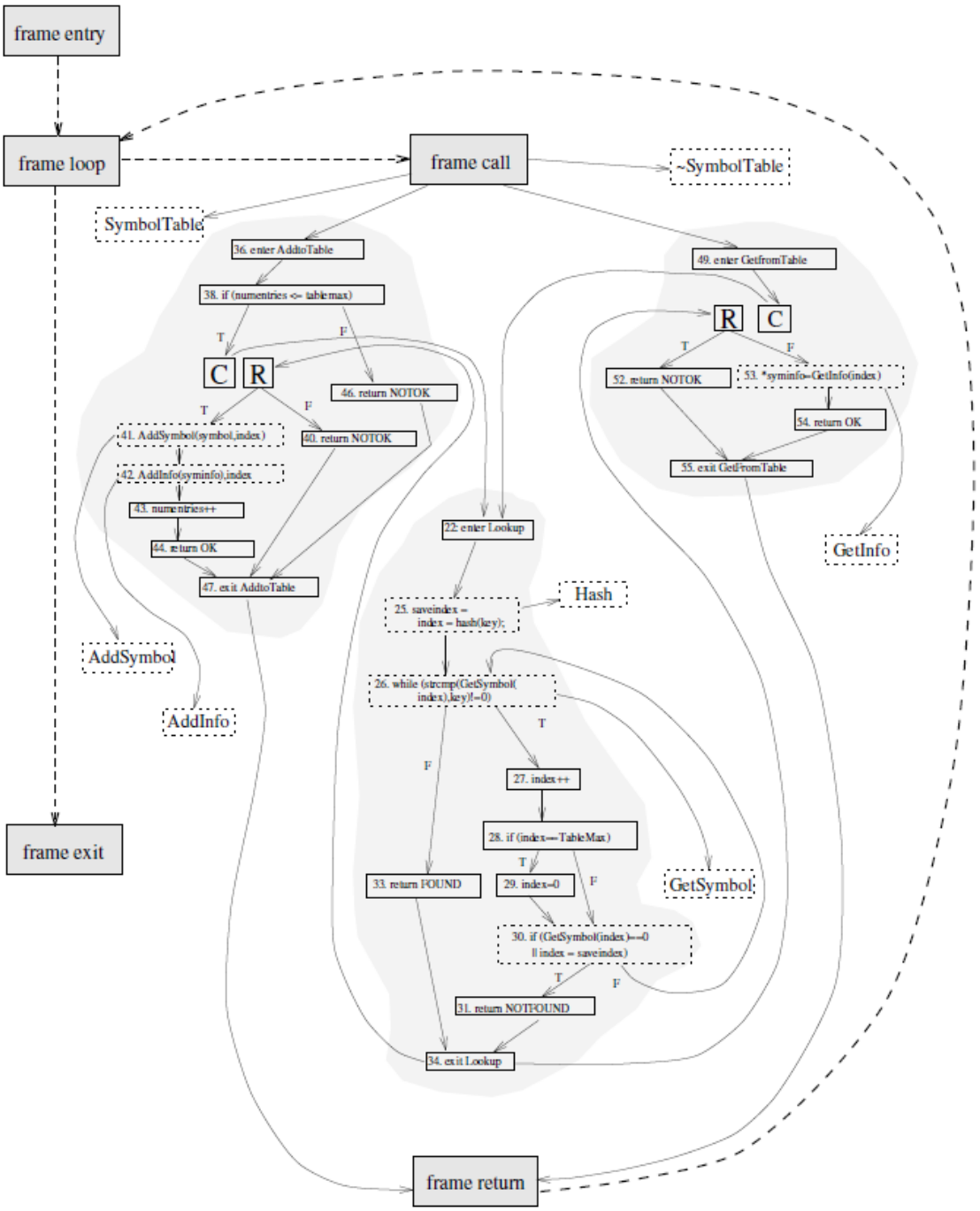


Figura 4.4: Grafo de fluxo de controle de classe ($\mathcal{CCFG}$) para a classe `SymbolTable` (Harrold e Rothermel, 1994).

- A implementação aspectual pode ser fortemente acoplada ao seu contexto de combinação,ou seja, os aspectos dependem da representação e implementação interna das classes em que estão combinados;

- Dependências de controle e dados não são facilmente compreensíveis a partir do código-fonte dos aspectos ou classes;

- O comportamento emergente do programa, em que o defeito pode estar na implementação da classe ou do aspecto, ou pode ser um efeito colateral de uma particular ordem de combinação de múltiplos aspectos.

Desta forma, o teste de sistemas OA deve ser baseado em modelos de defeitos que reflitam a estrutura e os comportamentos característicos dessa técnica. Assim, os critérios e as estratégias para testar programas OA devem ser desenvolvidos em termos desse modelo. Alexander et al. (2004), por exemplo, propõem uma taxonomia de defeitos baseado no mau uso dessa técnica, dividida nas seguintes classes: (1) restrição incorreta em padrões de conjuntos de junção; (2) precedência incorreta de aspectos; (3) defeito na preservação de pós-condições impostas; (4) defeito na preservação de invariantes de estado; (5) foco incorreto no fluxo de controle; e (6) mudanças incorretas em dependências de controle.

Embora muitas abordagens tenham sido propostas para programas OO e procedimentais, elas não podem ser aplicadas a programas OA (Zhao, 2002). Assim, tornam-se necessárias novas estratégias e ferramentas de teste que deem suporte a essa técnica.

Nesta seção são apresentadas as principais abordagens de teste de programas OA encontradas na literatura. A seleção dos trabalhos foi feita com base na literatura mais recente e na revisão sistemática sobre teste de programas OA conduzida por Ferrari e Maldonado (2007). Para dar uma ideia geral, na Tabela 4.2 são mostrados os trabalhos em que são descritos nesta seção, indicando: a técnica na qual a abordagem é baseada (Técnica); a fase de teste que a abordagem é aplicada (Fase); se a abordagem define formalmente algum critério de teste específico para programas OA (Critério); e se o trabalho apresenta alguma implementação de ferramenta de apoio (Ferramenta).

Tabela 4.2: Resumo dos trabalhos de teste de programas OA.

Técnica	**#**	**Autores**	**Fase**	**Critério**	**Ferramenta**
Estrutural	1	(Zhao, 2002)	unidade	N	N
	2	(Zhao, 2003)	unidade	N	N
	3	(Lemos, 2005)	unidade	S	S
	4	(Xie e Zhao, 2006)	–	S	S
	5	(Bernardi e Lucca, 2007)	integração	S	S
	6	(Franchin, 2007)	integração	S	S
	7	(Xu e Rountev, 2007)	unidade	N	S
	8	(Neves, 2009)	integração	S	S
	9	(Lemos, 2009)	integração	S	S
Estrutural e Mutação	10	(Mortensen e Alexander, 2004)	–	N	N
	11	(Mortensen e Alexander, 2005)	–	N	N
	12	(Lemos et al., 2006)	integração	N	N
Estrutural e Funcional	13	(van Deursen et al., 2005)	integração	S	N

S/N = Sim/Não
– = Não enfatizam

4.4.1 Teste Estrutural de Unidade de Programas OA

Em programas orientados a aspectos, um aspecto (ou uma classe) é projetado para ser o mais independente possível do seu ambiente. Assim, é possível exercitá-los individualmente por meio de pequenos programas de teste. Entretanto, um aspecto pode afetar o comportamento de uma ou mais classes por meio de adendos, tornando o relacionamento entre o aspecto e as classes que ele afeta mais complexo. Deste modo, em programas OA é impraticável testar um aspecto ou uma classe isoladamente (Zhao, 2003).

A primeira abordagem de teste estrutural para programas OA foi proposta por Zhao. Em seu trabalho, foi apresentada uma abordagem baseada em fluxo de dados que tem o objetivo de testar dois tipos de unidades: testar os aspectos juntamente com os métodos cujos comportamentos podem ser afetados pelos adendos (perspectiva dos aspectos); e testar as classes juntamente com os adendos que podem afetar o seu comportamento (perspectiva das classes). A partir daí foram definidos alguns conceitos:

- **aspecto combinado** (**c-aspecto**) é um aspecto combinado com alguns métodos pertencentes a uma ou mais classes de modo que o comportamento dos métodos pode ser afetado pelo adendo do aspecto;

- **classe combinada** (**c-classe**) é uma classe combinada com alguns adendos de um ou mais aspectos, tal que os adendos podem afetar o comportamento dos métodos da classe;

- **método combinado** (**c-método**) é um método combinado com um ou mais adendos que pode afetar seu comportamento;

- **adendo combinado** (**c-adendo**) é um adendo combinado com um ou mais métodos que ele afeta.

Na Figura 4.5 é apresentado o exemplo utilizado por Zhao para ilustrar a abordagem proposta. A classe `Point` representa pontos e possui atributos x e y referentes às coordenadas do ponto. A classe `Shadow` representa sombras de pontos, e também possui atributos x e y para as suas coordenadas. O aspecto `PointShadowProtocol` implementa a relação entre pontos e sombras e introduz um atributo `shadow` na classe `Point` para representar a sombra de um dado ponto. Para relacionar os objetos, o aspecto associa uma sombra a cada ponto que é criado, por meio de adendo posterior que executa quando o construtor de `Point` é chamado. Tal adendo faz uso do método `associate` para associar a sombra criada ao ponto, atribuindo-a à `shadow` (o atributo introduzido). Para manter a relação consistente, sempre que um objeto ponto se move como consequência da execução

de um dos métodos que alteram suas coordenadas (`setX` e `setY`), a sua sombra é atualizada pelos adendos posteriores que executam sempre que esses métodos são chamados. O método `getShadow` é utilizado pelos adendos posteriores para obter a sombra do objeto ponto interceptado.

```
ce0 public class Point {
 s1   protected int x, y;
me2   public Point(int _x, int _y) {
 s3     x = _x;
 s4     y = _y;
      }
me5   public int getX() {
 s6     return x;
      }
me7   public int getY() {
 s8     return y;
      }
me9   public void setX(int _x) {
s10     x = _x;
      }
me13  public void printPosition() {
s14     System.out.println("Point at("+x+", "+y+")");
      }
me15  public static void main(String[] args) {
s16     Point p = new Point(1,1);
s17     p.setX(2);
s18     p.setY(2);
      }
    }
ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;
me22   Shadow(int _x, int _y) {
 s23     x = _x;
 s24     y = _y;
      }
me25  public void printPosition() {
 s26     System.out.println("Shadow at ("+x+","+y+")");
      }
    }
```

```
ase27 aspect PointShadowProtocol {
  s28   private int shadowCount = 0;
 me29   public static int getShadowCount() {
  s30     return PointShadowProtocol.aspectOf().shadowCount;
        }
  s31   private Shadow Point.shadow;
 me32   public static void associate(Point p, Shadow s) {
  s33     p.shadow = s;
 me34   public static Shadow getShadow(Point p) {
  s35     return p.shadow;
        }
 pe36   pointcut setting(int x, int y, Point p):
          args(x, y) && call(Point.new(int, int);
 pe37   pointcut settingX(Point p):
          target(p) && call(Point.setX(int));
 pe38   pointcut settingY(Point p):
          target(p) && call(Point.setY(int));

 ae39   after(int x, int y, Point p) returning :
          setting(x, y, p) {
  s40     Shadow s = new Shadow(x + Shadow.offset,
                                y + Shadow.offset);
  s41     associate(p, s);
  s42     shadowCount++;
        }
 ae43   after(Point p): settingX(p) {
  s44     Shadow s = getShadow(p);
  s45     s.x = p.getX() + Shadow.offset;
  s46     p.printPosition();
  s47     s.printPosition();
        }
 ae48   after(Point p): settingY(p) {
  s49     Shadow s = getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
        }
      }
```

Figura 4.5: Exemplo para o teste de programas OA (Zhao, 2003).

De acordo com Zhao, as c-classe e os c-aspectos são considerados as menores unidades do programa. Além disso, os aspectos e classes são compostos de módulos (parte da unidade de teste) que podem ser c-adendo, c-método, método simples, construtor ou método introduzido. A partir daí foi definido que os casos de teste de fluxo de dados podem ser derivados de três perspectivas: intramódulo, intermódulo e intra-aspecto ou intraclasse.

Na perspectiva **intramódulo** são selecionados casos de teste para exercitar pares def-uso internos do módulo. Na perspectiva **intermódulo**, por sua vez, são selecionados casos de teste para exercitar pares def-uso que extrapolam o módulo. Na perspectiva **intra-aspecto** ou **intraclasse** são selecionados os casos de teste para exercitar os pares def-uso obtidos a partir de uma sequência de chamadas aleatórias de métodos públicos de um c-aspecto ou c-classe.

Para facilitar a geração de casos de teste e assegurar a qualidade dos c-aspectos e c-classes, Zhao propõe um modelo estrutural para teste de unidade de programas OA. O modelo estrutural consiste de três diferentes tipos de grafos de fluxo de controle: Grafo de

Fluxo de Controle para módulos individuais ($\mathcal{CFG}$), Grafo de Fluxo de Controle Interprocedimental para módulos interativos ($\mathcal{ICFG}$) e Grafo de Fluxo de Controle com Moldura para aspectos e classes ($\mathcal{FCFG}$). Na Figura 4.6 é mostrado o $\mathcal{FCFG}$ para duas unidades.

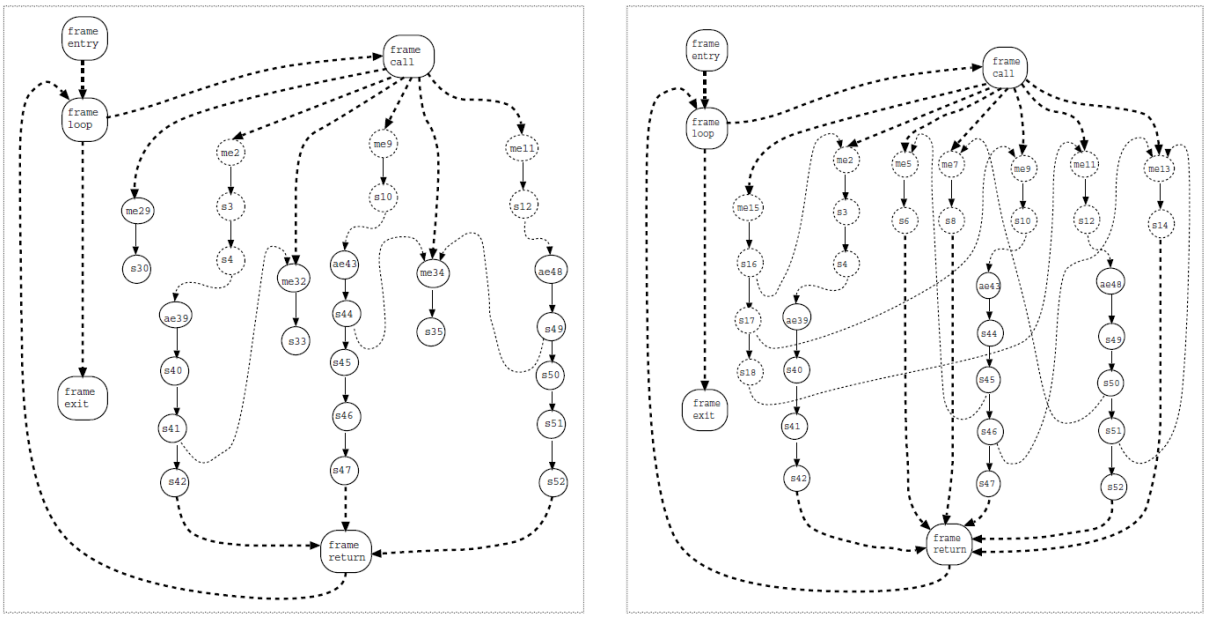


Figura 4.6: $\mathcal{FCFG}$ para o c-aspecto `PointShadowProtocol` e para a c-classe `Point` (Zhao, 2003).

Uma outra abordagem é proposta por Lemos et al. (2007) que explora o teste estrutural de unidades de programas orientados a objetos e a aspectos – métodos e adendos – e de unidades aspectuais de programas AspectJ (Kiczales et al., 2001a) isoladamente (i.e. teste de unidade). Nesse trabalho, cada método e cada adendo declarado são considerados como sendo as menores unidades do programa. Para representar os nós e as arestas transversais dos programas OA, o pesquisador estendeu o grafo de fluxo de controle ($\mathcal{CFG}$) e o grafo Def-Uso ($\mathcal{DUG}$). A partir dessa extensão constrói-se o Grafo de Fluxo de Controle Orientado a Aspectos ($\mathcal{AOCFG}$) e o Grafo Def-Uso Orientado a Aspectos ($\mathcal{AODU}$), ambos baseados em instruções de *bytecode*.

O grafo $\mathcal{AOCFG}$ é construído para representar qualquer unidade de um programa OA, ou seja, tanto métodos quanto adendos, e o grafo $\mathcal{AODU}$ é o grafo $\mathcal{AOCFG}$ com informações de definições e usos de variáveis utilizado para a aplicação de critérios baseados em fluxo de dados. Segundo Lemos, como o grafo $\mathcal{AODU}$ é o $\mathcal{AOCFG}$ estendido, é necessária apenas a construção do grafo $\mathcal{AODU}$ para derivar requisitos de teste tanto para o fluxo de controle quanto para o fluxo de dados do programa. Portanto, a definição do grafo $\mathcal{AOCFG}$ será omitida.

Desta forma, segundo Lemos, o grafo $\mathcal{AODU}$ de uma dada unidade u é definido como $\mathcal{AODU}\ (u) = (N, E, s, T, C)$ tal que cada nó $n \in N$ representa um bloco de instruções:

- N representa o conjunto de nós de um grafo $\mathcal{AODU}$: $N = \{n | n$ corresponde a um bloco de instruções de *bytecode* de $u\}$, ou seja N é um conjunto não vazio de nós, representando todos os blocos de instruções de *bytecode* de u; I_n é a n-tupla ordenada de instruções agrupadas no nó u;

- $E = E_r \cup E_e$ é o conjunto completo de arestas do grafo $\mathcal{AODU}$. Considere $\mathcal{IG}$ $(u) = (NI, EI, si, TI, CI)$:

  - E_r é o conjunto de arestas regulares definido como $E_r = \{(n_i, n_j) |$ existe uma aresta regular que parte do último elemento de I_{n_i} para o primeiro elemento de I_{n_j} no $\mathcal{IG}$ $(u)\}$;
  - E_e é o conjunto de arestas de exceção definido como $E_e = \{(n_i, n_j) |$ existe uma aresta de exceção com origem no último elemento de I_{n_i} e chegada no primeiro elemento de I_{n_j} no IG $(u)\}$
  - $E_c \subseteq E_r$ é o conjunto de arestas transversais (*crosscutting edges*) definido como $E_c = \{(x, y) \in E \mid (x \in C) \vee (y \in C)\}$ (componente C definido abaixo).

- $s \in N$ é o nó de entrada de u;

- $T \subseteq N$ é o conjunto (possivelmente vazio) de nós de saída.

- $C \subseteq N$ é o conjunto (possivelmente vazio) de nós de transversais. Neste caso, um nó transversal corresponde a um bloco de instruções na qual uma das instruções representa a execução de um adendo.

A representação gráfica do $\mathcal{AODU}$ é definida da seguinte forma:

- Um nó regular é representado por um círculo desenhado com uma linha simples e seu rótulo contém a primeira instrução de *bytecode* do bloco representado pelo nó;

- Um nó de chamada é representado por um círculo desenhado com linhas duplas e seu rótulo contém o deslocamento da primeira instrução de *bytecode* representada pelo nó;

- Um nó transversal é representado por uma elipse desenhada com linha tracejada. Seu rótulo informa, além do deslocamento da primeira instrução de *bytecode* representada pelo nó, que tipo de adendo afeta aquele ponto (`before`, `around` ou `after`) e a qual aspecto o adendo pertence;

- Um nó de saída é representado por um círculo desenhado com uma linha simples negritada e seu rótulo contém o deslocamento da primeira instrução de *bytecode* representada pelo nó;

Na Figura 4.7 é mostrada uma aplicação OA simples escrita em AspectJ e na Figura 4.8 é mostrado o grafo $\mathcal{AODU}$ do método `affectedMethod`.

```
public class Point {
  public int x;
  public int y;
  public AClass a;

  public Point(int _x, int _y) {
      x = _x;
      y = _y;
  }

  public void affectedMethod(Point p, int _x, int _y) {
    try {
      if (p.x <= 10 && p.y <= 10) {
        p.x = _x; p.y = _y;
        p.a = new AClass(10, 20);
      }
      p.printPoint(p);
      System.out.println(p.x);
    } catch (AnException ae) {
      System.out.println("Exception catched!");
    }
  }

  public void printPoint(Point p) throws AnException {
    if (p.x == 0) {
      AnException ae = new AnException();
      throw ae;
    }
    ...
  }
  ...
  public static class AnException extends Exception {
    ...
  }
}
```

```
public aspect AnAspect {
  pointcut exec(Point p, int i, int j):
    execution(void Point.affectedMethod(Point, int, int))
    && args(p, i, j);

  pointcut settingA(AClass a):
    set(AClass Point.a) && args(a) && !within(aspects.*);

  pointcut handlerPC(Point p):
    handler(Point.AnException) && this(p);

  before(Point p, int i, int j): exec(p, i, j) {
    if (p.x >= 0) p.x = i + 3;
    if (p.y >= 0) p.y = j + 4;
  }

  after (Point p, int i, int j) returning(): exec(p, i, j) {
    System.out.println("after returning exec");
    if (i > 10)
      System.out.println("i > 10");
    else
      System.out.println("i <= 10");
    if (p.x > 10)
      System.out.println("p.x > 10");
    else
      System.out.println("p.x <= 10");
  }

  void around(AClass a) : settingA(a) {
    System.out.println("around settingA");
    a.a = 20; a.b = 30;
    proceed(a);
  }

  before(Point p) : handlerPC(p) {
    p.x = 40;
  }
}
```

Figura 4.7: Exemplo de um programa OA escrito em AspectJ (Lemos, 2005).

Além disso, Lemos definiu critérios de teste baseados em fluxo de controle e de dados específicos para o teste de unidade de programas OA. Considerando o teste de fluxo de controle, os critérios de teste **todos-nós**, **todas-arestas**, **todos-nós-independentes-de-exceção**, **todos-nós-dependentes-de-exceção**, **todas-arestas-independentes-de-exceção** e **todas-arestas-dependentes-de-exceção** (propostos por Vincenzi) foram revisados para o contexto do teste de programas OA. Além disso, o pesquisador definiu mais dois novos critérios de fluxo de controle específicos de programas OA: o critério **todos-nós-transversais** e o critério **todas-arestas-transversais**. Com relação ao critério de fluxo de dados, Lemos revisou no contexto do teste de programas OA os critérios **todos-usos**, **todos-usos-independentes-de-exceção** e **todos-usos-dependentes-de-exceção** e definiu um novo critério específico para programas OA denominado **todos-usos-transversais**. Lemos também estendeu a ferramenta *JaBUTi* (Vincenzi, 2004) para apoiar o teste unitário de aspectos, implementando o grafo $\mathcal{AODU}$ e os critérios propostos.

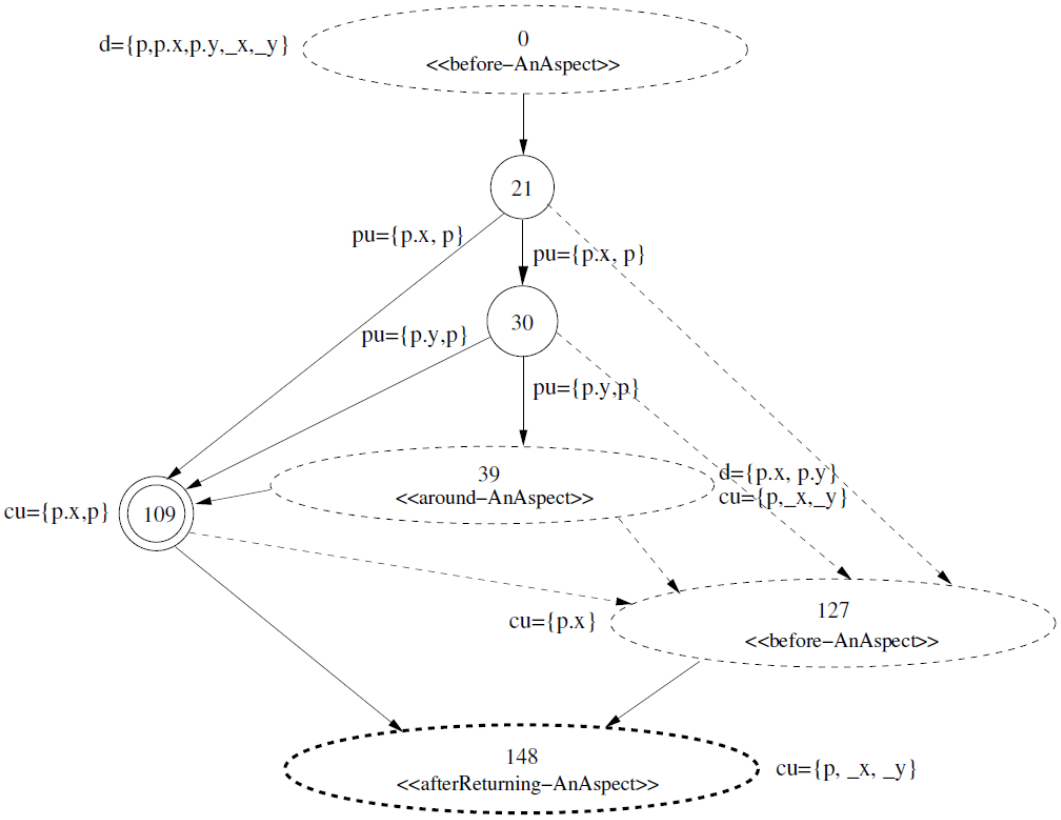


Figura 4.8: Grafo $\mathcal{AODU}$ do método `affectMethod` (Lemos, 2005).

4.4.2 Teste Estrutural de Integração de Programas OA

O trabalho de Lemos propõe uma nova abordagem de teste estrutural, já que é possível testar unidades de software orientadas a objetos e a aspectos. Porém, um problema não levado em conta no teste de unidade é a interação entre unidades, com respeito à correção de suas interfaces. Nesse contexto, o teste de unidade não é suficiente para dar a confiança suficiente de que o programa está correto, sendo necessário prosseguir com o teste de integração e o teste de sistema.

Franchin (2007), em continuidade ao trabalho de Lemos (2005), propôs o teste de integração de pares de unidades orientadas a objetos e a aspectos. Esse trabalho foi inspirado na proposta apresentada por Vilela et al. (1999), que propunha o teste par-a-par para programas procedimentais.

O teste estrutural de integração par-a-par (*pairwise*) considera as unidades em pares para derivar os requisitos de teste. Em programas OO, considerando o método como a menor unidade de teste, o teste abrange os pares de métodos (m-m). Por fim, em programas OA, considerando um método ou um adendo como sendo a menor unidade a ser testada, essa abordagem de teste compreende os seguintes pares de unidades: método-método (m-m), método-adendo (m-a), adendo-método (a-m) e adendo-adendo (a-a).

Segundo Franchin, para efetuar o teste estrutural de integração par-a-par entre os pares de unidades de programas OO (m-m) ou OA (m-m, m-a, a-m ou a-a) deve-se considerar todo o fluxo de execução (fluxo de controle e de dados) que ocorre entre a unidade chamadora e a unidade chamada. E para representar todo o fluxo de execução que ocorre entre um par de unidades, foi proposto um grafo que representasse a integração dos grafos das unidades que se relacionam, ou seja, um grafo de integração par-a-par. A partir daí, pode-se definir o Grafo Def-Uso Par-a-Par (*PairWise Def-Use* - $\mathcal{PWDU}$) como um grafo dirigido cuja utilização tem como objetivo apoiar o teste estrutural de integração par-a-par das unidades de um programa OO ou OA.

O grafo $\mathcal{PWDU}$ é uma abstração formada pela integração do grafo $\mathcal{AODU}$ da unidade chamadora (ou que chama) e do grafo $\mathcal{AODU}$ da unidade chamada. Essa integração é feita de modo que o grafo $\mathcal{AODU}$ da unidade chamada seja integrado ao grafo $\mathcal{AODU}$ da unidade chamadora. Para diferenciar os nós e as arestas das unidades integradas, são definidos um novo tipo de nó – nó integrado (*integrated node*) que representa os nós da unidade chamada – e dois novos tipos de arestas – aresta integrada (*integrated edge*), que é a aresta entre dois nós integrados e a aresta de integração (*integration edge*), que representa a aresta entre um nó da unidade chamadora e um nó da unidade chamada, ou vice-versa.

Além disso, Franchin definiu critérios de teste baseados em fluxo de controle e de dados específicos para teste estrutural de integração par-a-par de programas OO e OA. Considerando o teste de fluxo de controle, foram propostos os critérios **todos-nós-integrados** e **todas-arestas-integradas** que foram revisados a partir dos critérios todos-nós e todas-arestas. Com relação ao critério de fluxo de dados, o critério todos-usos foi revisado e o critério **todos-usos-integrados** foi proposto.

Como exemplo, na Figura 4.9, é mostrado o código de uma aplicação OA. A aplicação possui duas classes e um aspecto. A classe `Calculo` possui o método `calcular` que efetua os cálculos de soma e subtração dos números e armazena os resultados em atributos de instância. A classe `Principal` possui o método `efetuarCalculos` que cria um objeto da classe `Calculo` e, a partir deste objeto, chama o método `calcular`. Por fim, o aspecto `AspectLog` tem o conjunto de junção `pcLog` que captura toda execução do método calcular da classe `Calculo`. O adendo anterior, responsável por imprimir registros de log que contêm os números que serão calculados, é disparado nos pontos de junção selecionados pelo conjunto de junção `pcLog`.

Na Figura 4.11, é mostrado o grafo $\mathcal{PWDU}$ do par de unidades `efetuarCalculos` da classe `Principal` com o método `calcular` da classe `Calculo` a partir do grafo $\mathcal{AODU}$ da unidade chamadora (Figura 4.10(a)) e do grafo $\mathcal{AODU}$ da unidade chamada (Figura 4.10(b)). O nó 11 do grafo $\mathcal{AODU}$ do método `efetuarCalculos` é o nó onde ocorre a

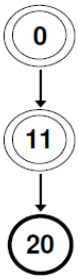
```
01. public aspect AspectLog {
02.   pointcut pcLog(int a1, int a2):
03.     execution(* src.Calculo.calcular(..))
          && args(a1,a2);
04.
05.   before(int a1, int a2) : pcLog(a1,a2) {
06.     System.out.println("Numeros: " +
          a1 + " e " + a2);
07.   }
08. }

09. public class Calculo {
10.   public int resSum;
11.   public int resSub;
12.
13.   public void calcular(int p1, int p2) {
14.     this.resSum = p1 + p2;
15.     if ( p1 > p2 )
16.       this.resSub = p1 - p2;
17.     else
18.       this.resSub = p2 - p1;
19.   }
20. }
```

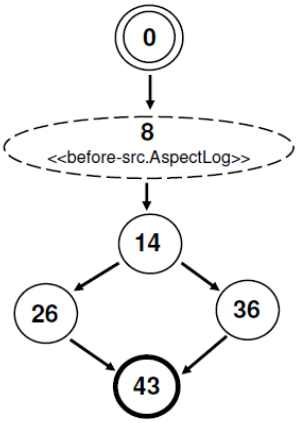
```
21. public class Principal {
22.   public static void efetuarCalculos(
        int d1, int d2) {
23.     int num1 = d1;
24.     int num2 = d2;
25.     Calculo calc = new Calculo();
26.     calc.calcular(num1, num2);
27.     System.out.println("Soma = " +
          calc.resSum);
28.     System.out.println("Subtracao = " +
          calc.resSub);
29.   }
30. }
```

Figura 4.9: Código de uma aplicação que realiza soma e subtração (Franchin, 2007).

chamada ao método `calcular` e onde será feita a integração do grafo $\mathcal{AODU}$ do método `calcular`.



(a) Grafo $\mathcal{AODU}$ do método `efetuarCalculos`



(b) Grafo $\mathcal{AODU}$ do método `calcular`

Figura 4.10: Grafos $\mathcal{AODU}$ das unidades chamadora e chamada (Franchin, 2007).

Franchin também estendeu a ferramenta *JaBUTi/AJ* para dar apoio ao teste estrutural de integração par-a-par e à análise de cobertura de testes realizados segundo os três critérios de teste par-a-par propostos.

Neves (2009), em continuidade ao trabalho de Franchin, definiu um grafo de programa que integra um método ou adendo com todos os métodos que ele chama diretamente e todos os comportamentos introduzidos pelos aspectos que o interceptam diretamente.

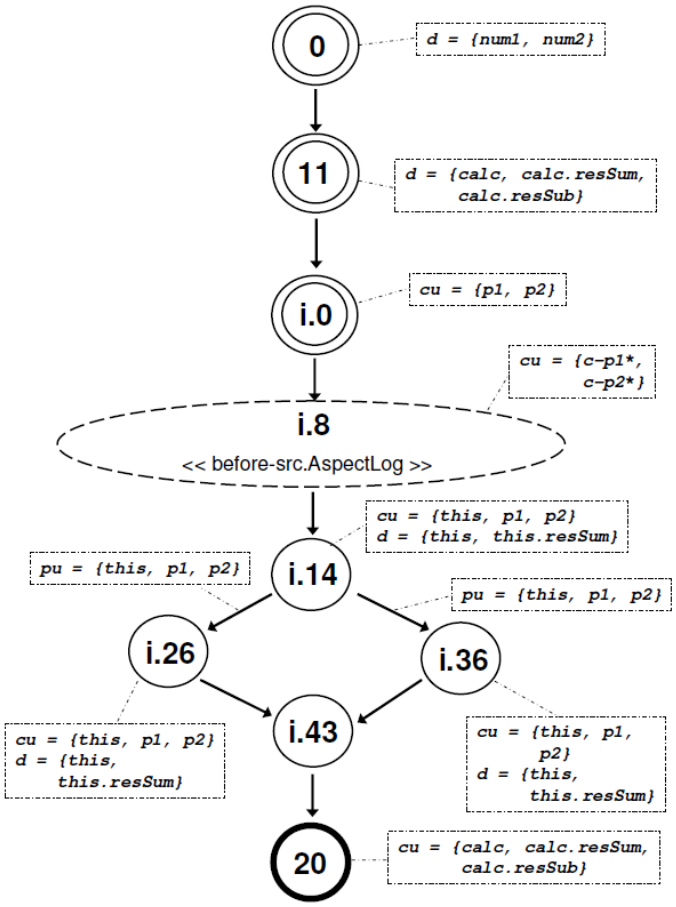


Figura 4.11: Grafo $\mathcal{PWDU}$ do par de unidades (Franchin, 2007).

Isto é, chamadas e interceptações com profundidade 1. Com esse objetivo, Neves definiu formalmente o grafo $\mathcal{IN}1\mathcal{P}$ e critérios de fluxo de dados e de controle. Com isso, é possível derivar requisitos de teste tanto para o fluxo de controle quanto para o fluxo de dados do programa.

Desta forma, segundo Neves, o grafo $\mathcal{IN}1\mathcal{P}$ das unidades $u_0, u_1, \cdots, u_n$ com u_0 como unidade chamadora e $u_1, \cdots, u_n$ como unidades chamadas, tem como nó de entrada o mesmo nó de entrada do grafo $\mathcal{AODU}$ da unidade u_0 (chamadora). O mesmo vale para os nós de saída, ou seja, os nós de saída do grafo $\mathcal{IN}1\mathcal{P}$ de $u_0, u_1, \cdots, u_n$ serão os mesmos nós de saída do grafo $\mathcal{AODU}$ da unidade u_0. A definição formal do grafo $\mathcal{IN}1\mathcal{P}$ de $u_0, u_1, \cdots, u_n$ é apresentada a seguir. Considere o grafo $\mathcal{AODU}$ das unidades $u_0, u_1, \cdots, u_n$ da forma:

$\mathcal{AODU}\ (u_0) = (N_0, E_0, s_0, T_0, I_0)$

$\mathcal{AODU}\ (u_1) = (N_1, E_1, s_1, T_1, I_1)$

$\cdots$

$\mathcal{AODU}\ (u_j) = (N_j, E_j, s_j, T_j, I_j)$

$\cdots$

$\mathcal{AODU}\ (u_n) = (N_n, E_n, s_n, T_n, I_n)$

Como cada unidade pode ser chamada mais de uma vez por u_0 (inclusive ele próprio se for recursivo), o conjunto de chamadas de u_0 com repetições pode ser representado da seguinte forma: $u_{0_1} \cdots u_{0_f} u_{1_1} \cdots u_{1_g} \cdots u_{n_1} \cdots u_{n_h}$ para $f, g, h \in \mathbb{N}^+$ e indicando o número de chamadas repetidas de $u_0, u_1, \cdots, u_n$, onde u_j não é um método de biblioteca do sistema. Considere ainda que $1 \leq j \leq n$ e $0 \leq k \leq n$, com $n \in \mathbb{N}$, s, x, y sejam nós do grafo $\mathcal{IN}1\mathcal{P}$ e que $d \in \mathbb{N}$. O grafo $\mathcal{IN}1\mathcal{P}$ das unidades $u_0, u_1, \cdots, u_n$ é definido como um grafo dirigido $\mathcal{IN}1\mathcal{P}\ (u_0, u_1, \cdots, u_n) = (N, E, s, T, I, I_s, R)$, tal que:

- Seja $\overline{N_k} = N_{k_1} \cup N_{k_2} \cup \cdots \cup N_{k_i}$, onde $i \in \mathbb{N}^+$ indica o número de vezes que u_k é chamado repetidamente em u_0.
  - $N = N_0 \cup \overline{N_0} \cup \overline{N_1} \cup \cdots \cup \overline{N_n}$ representa o conjunto completo de nós do grafo $\mathcal{IN}1\mathcal{P}$:
    - $N_I = \overline{N_0} \cup \overline{N_1} \cup \cdots \cup \overline{N_n}$ é o conjunto de nós integrados;

  Observações: Se u_0 não é recursiva $\overline{N_0} = \emptyset$. Se u_j só é chamado (ou afetado) uma vez $\overline{N_j} = N_j$.

- Seja $\overline{E_k} = E_{k_1} \cup E_{k_2} \cup \cdots \cup E_{k_i}$, onde $i \in \mathbb{N}^+$ indica o número de vezes que u_k é chamado repetidamente em u_0
  - $E = E'_0 \cup \overline{E_0} \cup \overline{E_1} \cup \cdots \cup \overline{E_n} \cup \overline{E_{I_0}} \cup \overline{E_{I_1}} \cup \cdots \cup \overline{E_{I_n}}$ é o conjunto completo de arestas do grafo $\mathcal{IN}1\mathcal{P}$, tal que:
    - $E'_0 \subseteq N_0 X N_0$ é o conjunto de arestas de u_0 definido como $E'_0 = E_0 - E_d$. Sendo que $E_0 = \{(x, y) \in E \mid (x \in N_0) \wedge (y \in N_0)\}$ e E_d é o conjunto de arestas que ligam os nós de interação aos nós subsequentes e que foram removidas. E_d é definido como $E_d = \{(x, y) \in E_0 | (x \in I_s)\}$. E_{r_0} e E_{e_0} são subconjuntos disjuntos de E_0 de arestas regulares e de exceção, respectivamente.
    - $E_{k_i} \subseteq N_{k_i} X N_{k_i}$ é o conjunto de arestas de u_{k_i} definido como $E_{k_i} = \{(x, y) \in E | (x \in N_{k_i}) \wedge (y \in N_{k_i})\}$. $E_{r_{k_i}}$ e $E_{e_{k_i}}$ são subconjuntos disjuntos de E_{k_i} de arestas regulares e de exceção, respectivamente.
    - $\overline{E_{I_k}} = E_{I_{k_1}} \cup E_{I_{k_2}} \cdots \cup E_{I_{k_i}}$, onde $i \in \mathbb{N}^+$ indica o número de vezes que u_k é chamado repetidamente em u_0.
    - $E_{I_{k_i}}$ é o conjunto de arestas de integração, criadas para integrar os n grafos $AODU$, definido como $E_{I_{k_i}} = \{((x, y) \in E | x \in I_s \wedge y = s_{k_i}) \vee (x \in N_{k_i} \wedge y \in R)\}$ sendo que:

· Se $x \in N_{k_i}$ e $y \in R$, então $x \in T_{k_i}$;

∗ $E_I = \overline{E_{I_0}} \cup \overline{E_{I_1}} \cup \cdots \cup \overline{E_{I_n}}$ é o conjunto de arestas integradas;

- $s \in N$ e $s = s_0$ é o nó de entrada do grafo $\mathcal{IN}1\mathcal{P}$, tal que:
  - $s_0 \in N_0$ é o nó de entrada de u_0;
  - $s_{k_i} \in N_{k_i}$ é o nó de entrada de u_{k_i};
- $T \subseteq N$ e $T = T_0$ é o conjunto de nós de saída do grafo $\mathcal{IN}1\mathcal{P}$, tal que:
  - T_0 é o conjunto de nós de saída de u_0;
  - T_{k_i} é o conjunto de nós de saída de u_{k_i}
- $I = I_0 \cup \overline{I_0} \cup \overline{I_1} \cup \cdots \cup \overline{I_n}$ é o conjunto completo de nós de interação (ou seja, nós transversais e nós de chamada) do grafo $\mathcal{IN}1\mathcal{P}$, tal que:
  - $\overline{I_k} = I_{k_1} \cup I_{k_2} \cup \cdots I_{k_i}$
  - $I_{k_i} \subseteq N_{k_i}$ é o conjunto de nós de interação de u_{k_i};
- $I_s = I_0$ é o conjunto dos nós de interação que serão expandidos.
- $R \subseteq N_0$ é o conjunto dos nós de retorno das chamadas a $u_{0_1} \cdots u_{0_f} u_{1_1} \cdots u_{1_g} \cdots u_{n_1} \cdots u_{n_h}$ para $f, g, h \in \mathbb{N}^+$, definido como $R = \{y \in N_0 | \exists (x, y) \in E_d \wedge x \in I_s\}$.

Além disso, Neves definiu critérios de teste baseados em fluxo de controle e dados. Considerando o teste de fluxo de controle, foram propostos os critérios **todos-nós-integrados-N1** e **todas-arestas-integradas-N1** que foram revisados a partir dos critérios todos-nós (*all-nodes*) e todas-arestas (*all-edges*). Com relação ao critério de fluxo de dados, Neves (2009) revisou no contexto da abordagem proposta o critério todos-usos (*all-uses*) e propôs o critério **todos-usos-integrados-N1**.

Como exemplo de construção do grafo $\mathcal{IN}1\mathcal{P}$, na Figura 4.12, é mostrado o código de uma aplicação que calcula recursivamente o número de Fibonacci para um determinado valor passado como parâmetro. Na Figura 4.13 é mostrado o grafo de unidade do método `fib` com as chamadas recursivas ocorrendo nos nós 7 e 14 na Figura 4.14 é mostrado o $\mathcal{IN}1\mathcal{P}$ do método `fib`.

Neves também estendeu a ferramenta JaBUTi/AJ para criar o grafo de todas as unidades envolvidas no nível 1 possibilitando realizar e instrumentar o teste de integração e calcular a cobertura obtida.

Lemos (2009) propõe uma outra abordagem de teste estrutural de integração baseando-se nos mecanismos de conjuntos de junção para AspectJ. Essa abordagem visa a garantir a cobertura da estrutura do adendo em cada ponto de junção que ele pode atuar e, dessa

```
public int fib(int k){
    if (k<=2){
        return 1;
    }
    else{
        return fib(k-1) + fib(k-2);
    }
}
```

Figura 4.12: Código em Java que calcula o número de Fibonacci.

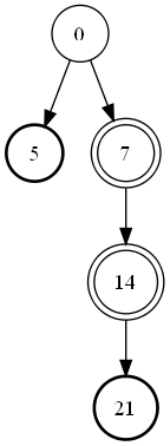


Figura 4.13: Grafo $\mathcal{AODU}$ do método `fib`

forma, aumentar a confiança de que as interações adicionadas estão corretas. Para isso, ele define um grafo de fluxo de controle e de dados para programas OA, chamado Def-Uso Baseado em Conjuntos de Junção (ou *Pointcut-based Def-Use Graph* - $\mathcal{PCDU}$). O $\mathcal{PCDU}$ deve ser construído para cada par adendo-conjunto de junção. Ele compreende os $\mathcal{AODU}$ das unidades afetadas pelo adendo, e o $\mathcal{AODU}$ do adendo repetido em cada possível ponto de junção. O $\mathcal{AODU}$ do adendo deve ser repetido, pois a intenção é rastrear sua execução em cada ponto de junção.

Além disso, Lemos definiu critérios de teste baseados em fluxo de controle e de dados específicos para a abordagem proposta. Considerando o teste de fluxo de controle, foram propostos os critérios **todos-nós-baseados-em-conjunto-de-junção** e **todas-arestas-baseadas-em-conjunto-de-junção**. Com relação ao critério de fluxo de dados, foi proposto o critério **todos-usos-baseados-em-conjunto-de-junção**.

Uma outra abordagem de teste estrutural de integração de programas orientados a aspectos é proposta por Bernardi e Lucca (2007). Essa abordagem é baseada na cobertura das interações entre adendos e métodos e considera apenas as unidades afetadas (direta ou indiretamente) por aspectos.

Para representar todo o fluxo de execução que ocorre entre as unidades, foi proposto o grafo $\mathcal{IACFG}$ (*Interprocedural Aspect Control Flow Graph*) que representa a integração

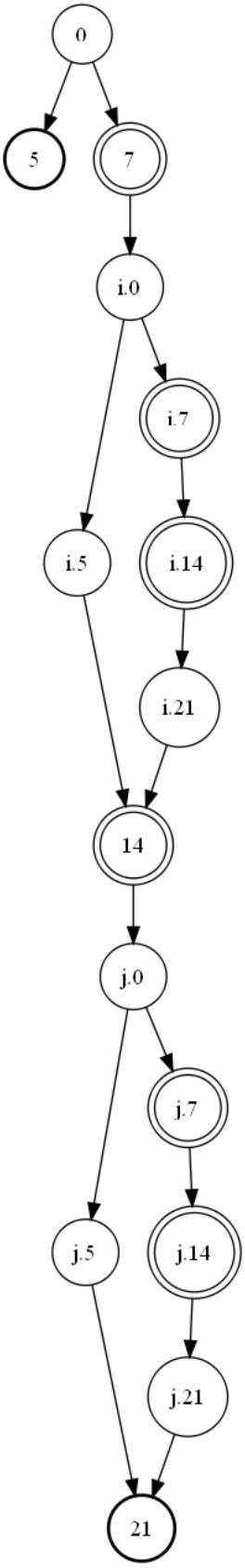


Figura 4.14: Grafo $\mathcal{IN}1\mathcal{P}$ do método `fib`.

dos grafos $\mathcal{OCFG}$ (*Operation Control Flow Graph*) das unidades afetadas por aspectos a partir de uma unidade inicial por meio da análise estática do código-fonte. O modelo de defeitos proposto por Alexander et al. (2004) foi estendido para considerar, de uma maneira mais específica, as características únicas e construções da linguagem AspectJ e guiar a análise de cobertura de critérios baseada no grafo $\mathcal{IACFG}$.

Como exemplo, na Figura 4.15 é mostrado o código-fonte de um programa, e na Figura 4.16 é mostrado o grafo $\mathcal{IACFG}$ correspondente.

```
1 package it.unisannio.se.sequenceGenerator;
2 aspect A {
3             static bool ClassA.inEnabled() {..}
4     static long l;
5     pointcut p() :
6     call(* Method2(...)) &&
7                 if (ClassA.isEnabled());
8     before() : p() {
9                         setEnabled();
10                      }
11     void around() : p() {
12                 l++;
13                    proceed(); // This around advice always
proceed.
14                 l--;
15            }
16 }
17 class ClassA {
18          ClassB b;
19          String field;
20           void Method1(){
21                  b.Method2(); // matched by p() associated
to
22                                  // both advices in asepct A.
23                field = ''field write'';
24            }
25 }
26 class ClassB {
27           void Method2() {
28                    setDisabled();
29            }
30 }
```

Figura 4.15: Um exemplo simples de um programa orientado a aspectos (Bernardi e Lucca, 2007).

Bernardi e Lucca também definiram os seguintes critérios de teste baseados no modelo de defeitos proposto:

- *All Join-point shadow*: Este critério exige que cada nó do tipo *join-point shadow*, ou seja, cada nó que representa um ponto de junção seja exercitado pelo menos uma vez por um caso de teste;

- *All Join-point shadow branches*: Este critério exige que cada arco que sai de um nó do tipo *join-point shadow* seja exercitado pelo menos uma vez por um caso de teste. Este critério inclui o critério *All Join-point shadow*;

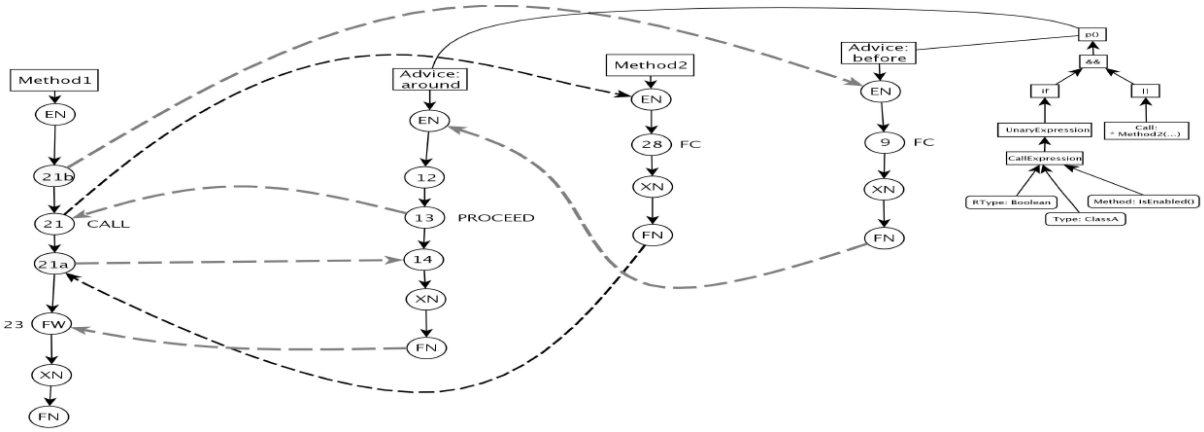


Figura 4.16: O $\mathcal{IACFG}$ do programa mostrado na Figura 4.15 (Bernardi e Lucca, 2007).

- *All pointcut node expressions*: Este critério requer que cada expressão booleana definida em cada ponto de junção seja exercitada com pelo menos um resultado verdadeiro e um resultado falso;

- *All pointcut node conditions*: Este critério requer que cada expressão booleana composta (ou seja, uma expressão com mais de uma expressão booleana e que seja conectada por operadores OR, AND, etc.) tenha um resultado verdadeiro e um resultado falso para cada nó de conjunto de junção com expressões compostas. Este critério inclui o critério *All pointcut node expressions*;

- *All pointcut node multiple conditions*: Este critério requer que todas as combinações de expressões booleanas definidas em um conjunto de junção sejam exercitadas. Ou seja, todas as combinações de verdadeiro e falso devem ser exercitadas considerando todas as expressões booleanas de um conjunto de junção. Este critério inclui o critério *All pointcut node conditions*;

- *All advices*: Este critério requer que todo adendo seja exercitado por pelo menos um caso de teste. O caso de teste deve exercitar o arco de retorno do adendo para o método, bem como todos os arcos relacionados a um comando de `proceed` no código do adendo;

Vale ressaltar que os critérios *All Join-point shadow* e *All Join-point shadow branches* são similares aos critérios de fluxo de controle propostos por Lemos e Neves.

Bernardi e Lucca também criaram um protótipo de uma ferramenta que faz a análise do código-fonte do programa em teste, implementa o algoritmo de construção dos grafos $\mathcal{OCFG}$ e $\mathcal{IACFG}$, e mostra a representação visual do grafo $\mathcal{IACFG}$.

O trabalho de Bernardi e Lucca, assim como o de Zhao, se baseia no código-fonte para a montagem do grafo, enquanto os trabalhos de Lemos, Franchin e Neves se baseiam no *bytecode* para a montagem dos respectivos grafos. Ou seja, as abordagens de Bernardi e

Lucca e Zhao são dependentes da disponibilidade do código-fonte para testes. Além disso, os trabalhos de Lemos, Franchin e Neves automatizam completamente as abordagens propostas, enquanto Bernardi e Lucca implementa uma ferramenta que realiza apenas a montagem e visualização do grafo.

Outro ponto que vale ressaltar é que a abordagem proposta por Bernardi e Lucca não apoia o teste da parte OO do programa, já que apenas as unidades afetadas direta e indiretamente por aspectos são representadas no grafo $\mathcal{IACFG}$. Além disso, diferentemente dos trabalhos de Lemos, Franchin e Neves, o trabalho de Bernardi e Lucca não propõe critérios baseados em fluxo de dados, que são extremamente relevantes no teste de integração.

4.4.3 Outras Abordagens de Teste de Programas OA

Mortensen e Alexander (2004, 2005) apresentam uma abordagem mista de teste estrutural e baseado em defeitos para programas OA escritos em AspectJ. Os autores tentam adequar os tipos de defeitos da taxonomia apresentada por Alexander et al. (2004), definindo um conjunto de critérios estruturais para atuarem em diferentes níveis de granularidade. No contexto de teste de mutação os autores definem três operadores de mutação para atuar nas definições de escopo dos descritores de conjuntos de junção e nas definições de precedência de aspectos. Os critérios e operadores não são formalmente definidos, eles são apenas discutidos em alto nível, e nenhuma implementação da abordagem é apresentada.

van Deursen et al. (2005) apresentam uma estratégia para refatoração de software OO com o apoio da POA. Porém, nessa estratégia também está incluída uma estratégia de teste para garantir o correto comportamento do software independentemente da refatoração. A estratégia, intitulada BETTAR (*Better Evolvability Through Tested Aspect Refactorings*), consiste de cinco passos, podendo-se destacar o projeto de casos de testes capaz de exercitar as porções de código refatoradas. Uma taxonomia de defeitos também é proposta no trabalho, incluindo defeitos relacionados a POA.

Xie e Zhao (2006) apresentam uma abordagem de teste estrutural e baseado em estados com o apoio de um *framework* intitulado Aspectra. O Aspectra é utilizado para gerar classes empacotadoras (*wrappers*) que são submetidas para ferramentas que geram casos de teste de comportamento aspectual considerando a cobertura estrutural e a de estados.

Lemos et al. (2006) apresentam uma abordagem mista de teste estrutural e baseado em defeitos para o teste de descritores de conjuntos de junção. A estratégia proposta é dividida em dois passos: (1) detectar pontos de junção inadequados selecionados pelo descritores de conjuntos de junção com apoio de teste estrutural baseado em fluxo de controle; e (2) identificar pontos de junção negligenciados com apoio de teste baseado em análise de mutantes.

Xu e Rountev (2007) propuseram uma abordagem de teste de regressão para programas AspectJ que utiliza um grafo de fluxo de controle para analisar o comportamento adicional introduzido por aspectos como forma de geração de requisitos de teste de regressão.

4.5 Considerações Finais

Neste capítulo algumas abordagens de teste estrutural OO e OA, tanto de unidade quanto de integração, foram discutidas. De acordo com essa pesquisa bibliográfica, é importante ressaltar a necessidade de abordagens de teste estrutural de programas OA que enfatizam a integração de todas as unidades na cadeia de chamadas. Além disso, nota-se a importância de uma ferramenta de apoio à aplicação das abordagens de teste propostas. Portanto, no próximo capítulo é apresentada a proposta de uma abordagem de teste estrutural de integração de programas OO e OA e no Capítulo 6 é mostrada a automatização da abordagem proposta.

Capítulo

5

Teste Estrutural de Integração Contextual de Programas OO e OA

5.1 Considerações Iniciais

Neste capítulo é apresentada uma abordagem para o teste estrutural de integração contextual para programas OO e OA escritos em Java e AspectJ. A finalidade dessa abordagem é descobrir defeitos que possam existir nas relações entre as unidades pertencentes a uma cadeia de chamadas/entrecortes no contexto da unidade sob teste. Ou seja, a ideia da abordagem é permitir que seja possível testar as unidades que se relacionam direta e indiretamente com a unidade sob teste.

Na Seção 5.2 são apresentados os fundamentos básicos da abordagem proposta, juntamente com um exemplo que será utilizado ao longo de toda a dissertação. Na Seção 5.3 é apresentado o modelo utilizado para possibilitar a representação do fluxo de execução entre as unidades, bem como o tratamento dado a algumas construções específicas das linguagens Java e AspectJ. Na Seção 5.4 é apresentado o modelo de fluxo de dados utilizado na abordagem de teste proposta. Na Seção 5.5 é apresentada uma família de critérios de teste estrutural de integração para programas OO e OA. Na Seção 5.6 é descrita uma estratégia básica de teste para ser utilizada na aplicação dos critérios propostos. Por fim, na Seção 5.7 são apresentadas as considerações finais deste capítulo.

5.2 Fundamentos do Teste Estrutural de Integração Contextual

Programas OO e OA são constituídos por unidades que se relacionam com o objetivo de produzir um determinado comportamento no programa. Nesta dissertação considera-se cada método, adendo e método inter-tipo declarado como as menores unidades do programa. Dessa forma, como um defeito pode estar relacionado com a interação entre as unidades, o teste de integração ganha importância, pois mesmo se a lógica e a funcionalidade das unidades apresentarem indícios de conformidade com as especificações e ausência de defeitos, isso não implica no seu correto funcionamento quando integradas.

A proposta do teste estrutural de integração contextual é considerar, na atividade de teste, todas as unidades que se relacionam com a unidade sob teste até o nível de profundidade máximo ou escolhido pelo testador com o propósito de descobrir defeitos que possam existir nas relações entre essas unidades. Para a aplicação desse tipo de análise é necessário um modelo para representar o fluxo de controle e de dados a partir de uma unidade a ser testada.

Para a aplicação desse tipo de análise de cobertura é necessário um modelo para representar o fluxo de controle e de dados a partir de uma unidade a ser testada. Seguindo as abordagens de teste propostas anteriormente pelo grupo de teste do autor desta dissertação, e levando em consideração os trabalhos relacionados apresentados no Capítulo 4, é definido neste capítulo um grafo de fluxo de controle e de dados para programas OO e OA, bem como tratamentos especiais dados à construções específicas de Java e AspectJ. Baseado nesse grafo e em um modelo de fluxo de dados definido por Vincenzi e adaptado por Franchin, é definida uma família de critérios de fluxo de controle e de dados (Franchin, 2007; Vincenzi, 2004).

O exemplo denominado `Shape` apresentado aqui, e que será utilizando durante o restante desta dissertação para exemplificar a abordagem de teste proposta e a implementação da extensão da *JaBUTi/AJ*, foi obtido do Programming Tools Group and Sable Research Group (2011). O programa `Shape` simula uma aplicação que modela diversas formas geométricas. Nesse exemplo existe uma superclasse chamada `TwoDShape` que têm como subclasses os tipos geométricos `Circle`, `Square` e `Triangle`. Cada uma das subclasses possui atributos e operações específicas, além das operações e atributos herdados da superclasse. Também existe a classe `Coordinate` que define as coordenadas de um forma geométrica no espaço. Além disso, o programa `Shape` possui os aspectos `Trace` e `TraceMyClasses` responsáveis por imprimir mensagens de rastreabilidade no programa. A Figura 5.1 mostra o código-fonte parcial do programa `Shape`.

```
1   // Copyright (c) Xerox Corporation 1998-2011.
2
3   public abstract class TwoDShape {
4
5     protected Coordinate coord;
6
7     protected TwoDShape(Coordinate c) {
8       this.coord = c;
9     }
10
11    public double getX() { return coord.getX(); }
12
13    public double getY() { return coord.getY(); }
14
15    public double distance(TwoDShape s) {
16      double dx = Math.abs(s.getX() - coord.getX());
17      double dy = Math.abs(s.getY() - coord.getY());
18      return Math.sqrt(dx*dx + dy*dy);
19    }
20
21    public abstract double perimeter();
22
23    public abstract double area();
24
25    public String toString() {
26      return (" @ " + coord);
27    }
28  }
29
30  public class Circle extends TwoDShape {
31    protected double radius;
32    protected Coordinate origin;
33
34    public Circle() {
35      super(new Coordinate(0, 0));
36      origin = new Coordinate(0, 0);
37      radius = 1.0;
38    }
39
40    public Circle(Coordinate newOrigin,
41                  double newRadius) {
42      super(newOrigin);
43      origin = newOrigin;
44      radius = newRadius;
45    }
46
47    public Circle(Coordinate c) {
48      this(c, 1.0);
49    }
50
51    public Circle(double r) {
52      this(new Coordinate(0.0, 0.0), r);
53    }
54
55    public double area() {
56      return Math.PI * radius*radius;
57    }
58
59    public double perimeter() {
60      return 2 * Math.PI * radius;
61    }
62
63    public Coordinate getOrigin() {
64      return origin;
65    }
66
67    public double getRadius() {
68      return radius;
69    }
70
71    public String toString() {
72      return ("Circle radius = "
73                + String.valueOf(radius)
74                  + super.toString());
75    }
76  }
77  public aspect TraceMyClasses extends Trace {
78    pointcut myClass(): within(TwoDShape)
79      || within(Circle) || within(Square)
80      || within(Triangle);
81  }
82
83  abstract aspect Trace {
84
85    public static int TRACELEVEL = 2;
86    protected static PrintStream
87                            stream = System.err;
88    protected static int callDepth = 0;
89
90    public static void initStream(PrintStream s) {
91      stream = s;
92    }
93
94    protected static void traceEntry(String str) {
95      if (TRACELEVEL == 0) return;
96      if (TRACELEVEL == 2) callDepth++;
97      printEntering(str);
98    }
99
100   protected static void traceExit(String str) {
101     if (TRACELEVEL == 0) return;
102     printExiting(str);
103     if (TRACELEVEL == 2) callDepth--;
104   }
105
106   private static void printEntering(String str) {
107     printIndent();
108     stream.println("--> " + str);
109   }
110
111   private static void printExiting(String str) {
112     printIndent();
113     stream.println("<-- " + str);
114   }
115
116   private static void printIndent() {
117     for (int i = 0; i < callDepth; i++)
118       stream.print("  ");
119   }
120
121   abstract pointcut myClass();
122   pointcut myConstructor(): myClass()
123                               && execution(new(..));
124   pointcut myMethod(): myClass()
125                               && execution(* *(..));
126
127   before(): myConstructor() {
128     traceEntry(""
129           + thisJoinPointStaticPart.getSignature());
130   }
131
132   after() returning() : myConstructor() {
133     traceExit(""
134           + thisJoinPointStaticPart.getSignature());
135   }
136
137   before(): myMethod() {
138     traceEntry(""
139           + thisJoinPointStaticPart.getSignature());
140   }
141
142   after() returning() : myMethod() {
143     traceExit(""
144           + thisJoinPointStaticPart.getSignature());
145   }
146 }
```

Figura 5.1: Código-fonte parcial do programa `Shape`.

5.3 Modelo de Representação (Grafo $\mathcal{C}o\mathcal{DU}$)

Para a realização do teste estrutural de integração contextual é necessário definir um modelo que represente todo o fluxo de execução a partir de um método sob teste. Ou seja, o modelo deve representar, de forma integrada, as unidades envolvidas no fluxo de

chamadas/entrecortes a partir da execução do método sob teste. Neste trabalho é definido um grafo, que deve ser construído para cada unidade a ser testada, chamado grafo Def-Uso Contextual (ou *Contextual Def-Use graph* – $\mathcal{C}o\mathcal{DU}$). o $\mathcal{C}o\mathcal{DU}$ é composto pela integração dos $\mathcal{AODU}$s (ver Seção 4.4.1) das unidades pertencentes à cadeia de chamadas/entrecortes do método sob teste com o $\mathcal{AODU}$ do método sob teste.

O grafo $\mathcal{C}o\mathcal{DU}$ da unidade u_0 com profundidade máxima de chamadas/entrecortes p e profundidade de chamadas/entrecortes escolhida d, $1 \leq d \leq p$, é definido como um grafo dirigido $\mathcal{C}o\mathcal{DU}\ (u_0, d) = (N, E, S, T, I, I_s, R)$, tal que:

- u_0 é a unidade chamadora (ou que entrecorta) e $u_{1.1}, \cdots, u_{1.n}$ são as unidades diretamente chamadas ou executadas por meio da ação de aspectos ($u_{1.1}, \cdots, u_{1.n}$ são, portanto, métodos ou adendos). As unidades $u_{1.1}, \cdots, u_{1.n}$ por sua vez, podem chamar (serem entrecortadas) diretamente outras (por outras) unidades $u_{2.1}, \cdots, u_{2.m}$. As unidades $u_{2.1}, \cdots, u_{2.m}$ podem chamar (serem entrecortadas) diretamente outras (por outras) unidades $u_{3.1}, \cdots, u_{3.l}$, e assim sucessivamente, com grau máximo de profundidade p. Portanto, o grafo $\mathcal{C}o\mathcal{DU}$ deve integrar as unidades ($u_0, u_{1.1}, \cdots, u_{1.n}$, $\cdots, u_{2.1}, \cdots, u_{2.m}, \cdots, u_{d.1}, \cdots, u_{d.z}$), e assim sucessivamente, para o nível de profundidade de chamada/entrecorte d escolhido (o caminho mais longo de interação tem profundidade p, ou seja, $1 \leq d \leq p$).

  Além disso, não existe a restrição de que $u_{a.i} \neq u_{b.j}$ ou $u_{a.i} \neq u_0$ para quaisquer $i, j \in \mathbb{N}^+$ e $1 \leq a, b \leq d$. Isto é, no conjunto de unidades chamadas pode existir algum $u_{a.i}$ tal que $u_{a.i} = u_0$ ou $u_{a.i} = u_{b.j}$ (chamada recursiva). Também é possível existir algum subconjunto $u_{a.i} = \cdots = u_{b.j}$ (uma unidade é chamada ou executada por meio da ação de aspectos duas ou mais vezes em um mesmo nível de profundidade). Vale ressaltar que as componentes das unidades nesses casos são tratadas separadamente.

  Portanto, as componentes do grafo são definidas como segue:

- $N = N_0 \cup N_{1.1} \cup \cdots \cup N_{1.n} \cup N_{2.1} \cup \cdots \cup N_{2.m} \cup \cdots \cup N_{d.1} \cup \cdots \cup N_{d.z}$ representa o conjunto completo de nós do grafo $\mathcal{C}o\mathcal{DU}$, tal que:

  - N_0 representa os nós do grafo da unidade sob teste u_0;
  - $N_{1.1}, \cdots, N_{1.n}, N_{2.1}, \cdots, N_{2.m}, \cdots, N_{d.1}, \cdots, N_{d.z}, \cdots$ representam todos os nós dos grafos das unidades que ocorrem na cadeia de chamadas/entrecortes de u_0;
  - N' é o conjunto de nós integrados do grafo $\mathcal{C}o\mathcal{DU}$ definido como $N' = N - N_0$;

- $E = E'_0 \cup E'_{1.1} \cup \cdots \cup E'_{1.n} \cup E'_{2.1} \cup \cdots \cup E'_{2.m} \cup \cdots \cup E'_{d.1} \cup E'_{d.z} \cup \cdots \cup E_{I_0} \cup E_{I_{1.1}} \cup \cdots \cup E_{I_{d-1.z}}$ é o conjunto completo de arestas do grafo $\mathcal{C}o\mathcal{DU}$, tal que:

  - $E'_0 \subseteq N_0 \times N_0$ é o conjunto de arestas de u_0 definido como $E'_0 = E_0 - E_{x_0}$ e E_x é o conjunto de arestas que ligam os nós de interação aos nós subsequentes e que foram removidas. E_x é definido como $E_x = \{(n_1, n_2) \in E_0 | (n_1 \in I_s)\}$;
  - $E'_{a.i} \subseteq N_{a.i} \times N_{a.i}$ é o conjunto de arestas de $u_{a.i}$ definido como $E'_{a.i} = E_{a.i} - E_{x_{a.i}}$ e $E_{x_{a.i}}$ é o conjunto de arestas que ligam os nós de interação aos nós subsequentes e que foram removidas. $E_{x_{a.i}}$ é definido como $E_{x_{a.i}} = \{(n_1, n_2) \in E_{a.i} | (n_1 \in I_s)\}$;
  - E_{I_0} é o conjunto de arestas de integração, criadas para integrar os n grafos $\mathcal{AODU}$, definido como $E_{I_0} = \{(n_1, n_2) \in E | ((n_1 \in I_s) \wedge (n_2 = s_{1.i})) \vee ((n_1 \in N_0) \wedge (n_2 \in R))\}$;
  - $E_{I_{a.i}}$ é o conjunto de arestas de integração, criadas para integrar os n grafos integrados $\mathcal{AODU}$, definido como $E_{I_{a.i}} = \{(n_1, n_2) \in E | ((n_1 \in I_s) \wedge (n_2 = s_{a+1.j})) \vee ((n_1 \in N_{a.i}) \wedge (n_2 \in R))\}$ sendo que:
    - se $n_1 \in N_{a.i}$ e $n_2 \in R$, então $x \in T_{a.i}$;
  - $E' = E - E'_0 - E_{I_0}$ é o conjunto de arestas integradas;

- $S \subseteq N$ e $S = \{s_0, s_{1.1}, \cdots, s_{1.n}, s_{2.1}, \cdots, s_{2.m}, \cdots, s_{d.1}, \cdots, s_{d.z}\}$ é o conjunto de nós de entrada do grafo $\mathcal{CoDU}$, tal que:
  - $s_0 \in N_0$ é o nó de entrada de u_0;
  - $s_{a.i} \in N_{a.i}$ é o nó de entrada de $u_{a.i}$;

- $T \subseteq N$ e $T = T_0 \cup T_{1.1} \cup \cdots \cup T_{1.n} \cup T_{2.1} \cup \cdots \cup T_{2.m} \cup \cdots \cup T_{d.1} \cup \cdots \cup T_{d.z}$ é o conjunto de nós de saída do grafo $\mathcal{CoDU}$, tal que:
  - T_0 é o conjunto de nós de saída de u_0;
  - $T_{a.i}$ é o conjunto de nós de saída de $u_{a.i}$;

- $I = I_0 \cup I_{1.1} \cup \cdots \cup I_{1.n} \cup \cdots \cup I_{2.1} \cup \cdots \cup I_{d.z}$ é o conjunto completo de nós de interação (ou seja, nós transversais e nós de chamada) do grafo $\mathcal{CoDU}$, tal que:
  - $I_0 \subseteq N_0$ é o conjunto de nós de interação de u_0;
  - $I_{a.i} \subseteq N_{a.i}$ é o conjunto de nós de interação de $u_{a.i}$;

- $I_s \subseteq I$ é o conjunto de nós de interação que serão expandidos, tal que:
  - $I_s = I - I_r$, onde $I_r = \{n_1 | n_1 \in (I_0 \vee I_{a.i})$ e $u_{a.i}$ é um método de biblioteca ou uma chamada recursiva em um nível de profundidade maior que 1$\}$;

- $R \subseteq N$ é o conjunto de nós de retorno das chamadas ou entrecortes de $u_0, u_{1.1}, \cdots, u_{1.n}$, $\cdots, u_{2.1}, \cdots, u_{2.m}, \cdots, u_{d.1}, \cdots, u_{d.z}$.

A representação gráfica do $\mathcal{C}o\mathcal{DU}$ é definida da seguinte forma:

- Um nó regular é representado por um círculo desenhado com uma linha simples e seu rótulo contém a primeira instrução de *bytecode* do bloco;
- Um nó de chamada é representado por um círculo desenhado com linhas duplas e seu rótulo contém o deslocamento da primeira instrução de *bytecode* representada pelo nó;
- Um nó transversal é representado por uma elipse desenhada com linha tracejada. Seu rótulo informa, além do deslocamento da primeira instrução de *bytecode* representada pelo nó, que tipo de adendo afeta aquele ponto (`before`, `around` ou `after`) e a qual aspecto o adendo pertence;
- Um nó de saída é representado por um círculo desenhado com uma linha simples negritada e seu rótulo contém o deslocamento da primeira instrução de *bytecode* representada pelo nó;
- Um nó integrado é representado como um nó regular, um nó de chamada, um nó transversal ou um nó de saída[1] e é rotulado com três cadeias de caracteres separadas por um ".". A primeira parte do rótulo representa o nível de profundidade de chamadas/entrecortes em que a instrução (ou conjunto de instruções) representado(s) pelo nó está em relação à unidade sob teste. A segunda parte do rótulo identifica o grafo $\mathcal{AODU}$ em uma determinada profundidade. Por fim, a terceira parte do rótulo corresponde ao deslocamento da primeira instrução de *bytecode* representada pelo nó;
- Uma aresta regular é representada por uma linha contínua, representando o fluxo de controle normal;
- Uma aresta de exceção é representada por uma linha tracejada, representando o fluxo de controle do nó no qual uma exceção é gerada até o primeiro nó correspondente ao tratador daquela exceção;
- Uma aresta integrada é representada como uma aresta regular ou uma aresta de exceção;
- Uma aresta de integração é representada como uma aresta regular.

[1] Excetuando-se os nós de saída de exceções, um nó de saída é transformado em nó regular na integração

Os conjuntos de definições e usos dos nós do grafo $\mathcal{C}o\mathcal{DU}$ são derivados a partir das instruções presentes no bloco de instruções de cada nó. Como o grafo $\mathcal{C}o\mathcal{DU}$ criado, é possível derivar os requisitos de teste para os critérios de fluxo de controle e fluxo de dados, conforme apresentado na Seção 5.5.

Para exemplificar o grafo proposto, a seguir serão mostrados os grafos $\mathcal{C}o\mathcal{DU}$s do método `area` da classe `Circle` (linhas 55–57, Figura 5.1) e a definição formal do grafo $\mathcal{C}o\mathcal{DU}$ do método `area` no nível máximo de profundidade (nível 4).

Como dito anteriormente, o $\mathcal{C}o\mathcal{DU}$ é composto pela integração dos $\mathcal{AODU}$s das unidades pertencentes à cadeia de chamadas/entrecortes do método sob teste com o $\mathcal{AODU}$ do método sob teste. Na Figura 5.2 são mostradas as interações disparadas pela execução do método `area` e na Figura 5.3 são mostrados todos os grafos $\mathcal{AODU}$s das unidades que fazem parte da cadeia de chamadas/entrecortes, além do $\mathcal{AODU}$ do método `area`. Entretanto, vale ressaltar que na construção do grafo $\mathcal{C}o\mathcal{DU}$, nós dos grafos $\mathcal{AODU}$s podem ser divididos para representarem interações que acontecem no programa. Ou seja, os grafos $\mathcal{AODU}$s integrados no $\mathcal{C}o\mathcal{DU}$ não são necessariamente os mesmos grafos $\mathcal{AODU}$s gerados no teste de unidade.

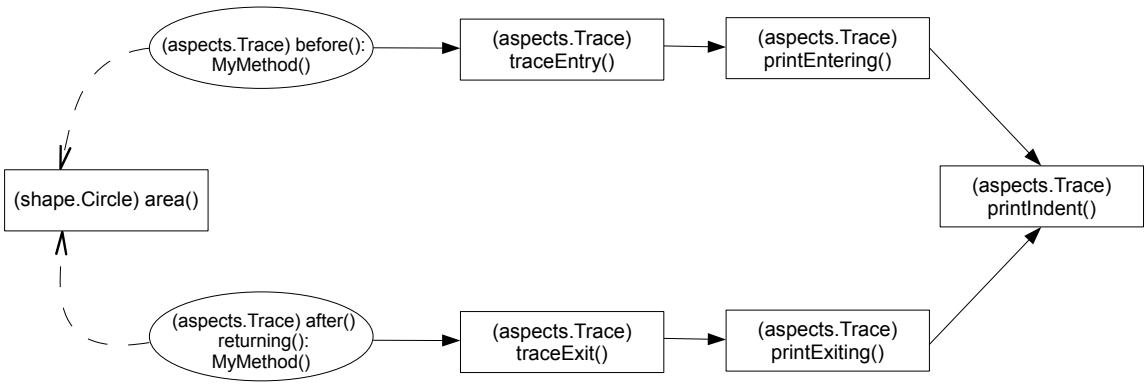
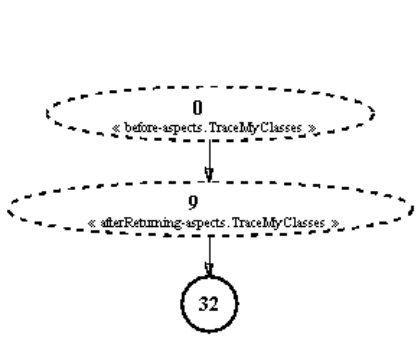
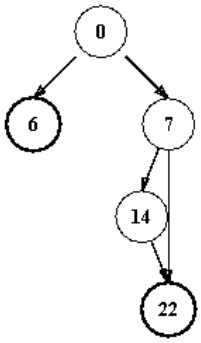


Figura 5.2: Interações disparadas a partir do método `area`.

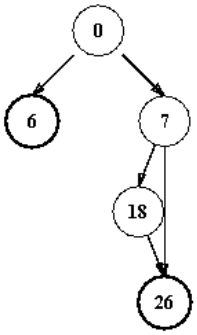
Portanto, o $\mathcal{C}o\mathcal{DU}$ do método `area`, no nível 1 de profundidade, é formado pelo $\mathcal{AODU}$ do método `area` e pelos $\mathcal{AODU}$s dos adendos `(aspects.Trace) before():MyMethod()` e `(aspects.Trace) after() returning():MyMethod()`. Na Figura 5.4 é mostrado o $\mathcal{C}o\mathcal{DU}$ do método `area` representando apenas interações no nível 1 de profundidade.



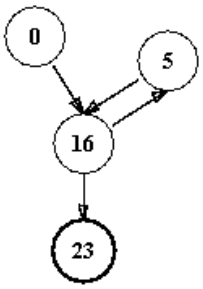
(a) $\mathcal{AODU}$ do método `area`.



(b) $\mathcal{AODU}$ do método `traceEntry`.



(c) $\mathcal{AODU}$ do método `traceExit`.



(d) $\mathcal{AODU}$ do método `printIndent`.

(e) $\mathcal{AODU}$ do método `printEntering`.

(f) $\mathcal{AODU}$ do método `printExiting`.

(g) $\mathcal{AODU}$ do adendo `(aspects.Trace) before():MyMethod()`

(h) $\mathcal{AODU}$ do adendo `(aspects.Trace) after() returning():MyMethod()`

Figura 5.3: Grafos $\mathcal{AODU}$ das unidades que formarão o $\mathcal{CoDU}$.

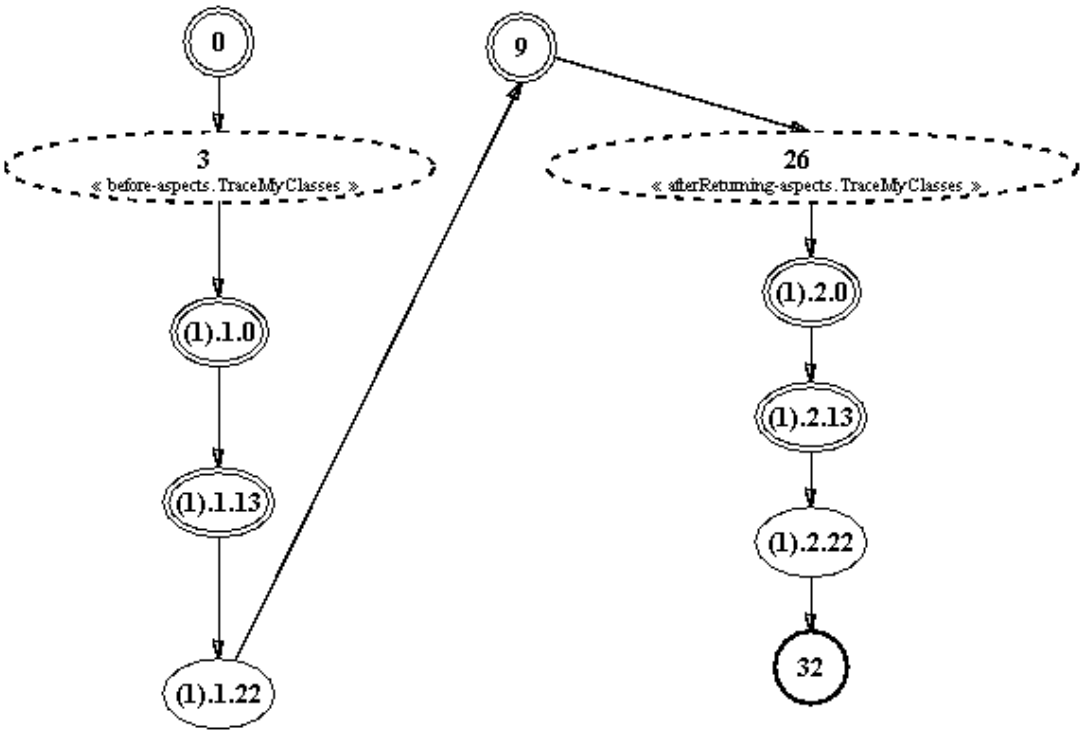


Figura 5.4: Grafo $\mathcal{CoDU}$ do método `area` no nível 1 de profundidade.

O $\mathcal{CODU}$ do método `area`, no nível 2 de profundidade, é formado pelos $\mathcal{AODU}$s representados no nível 1 de profundidade e pelos métodos `traceEntry` e `traceExit`. Na Figura 5.5 é mostrado o $\mathcal{CODU}$ do método `area` representando interações nos níveis 1 e 2 de profundidade.

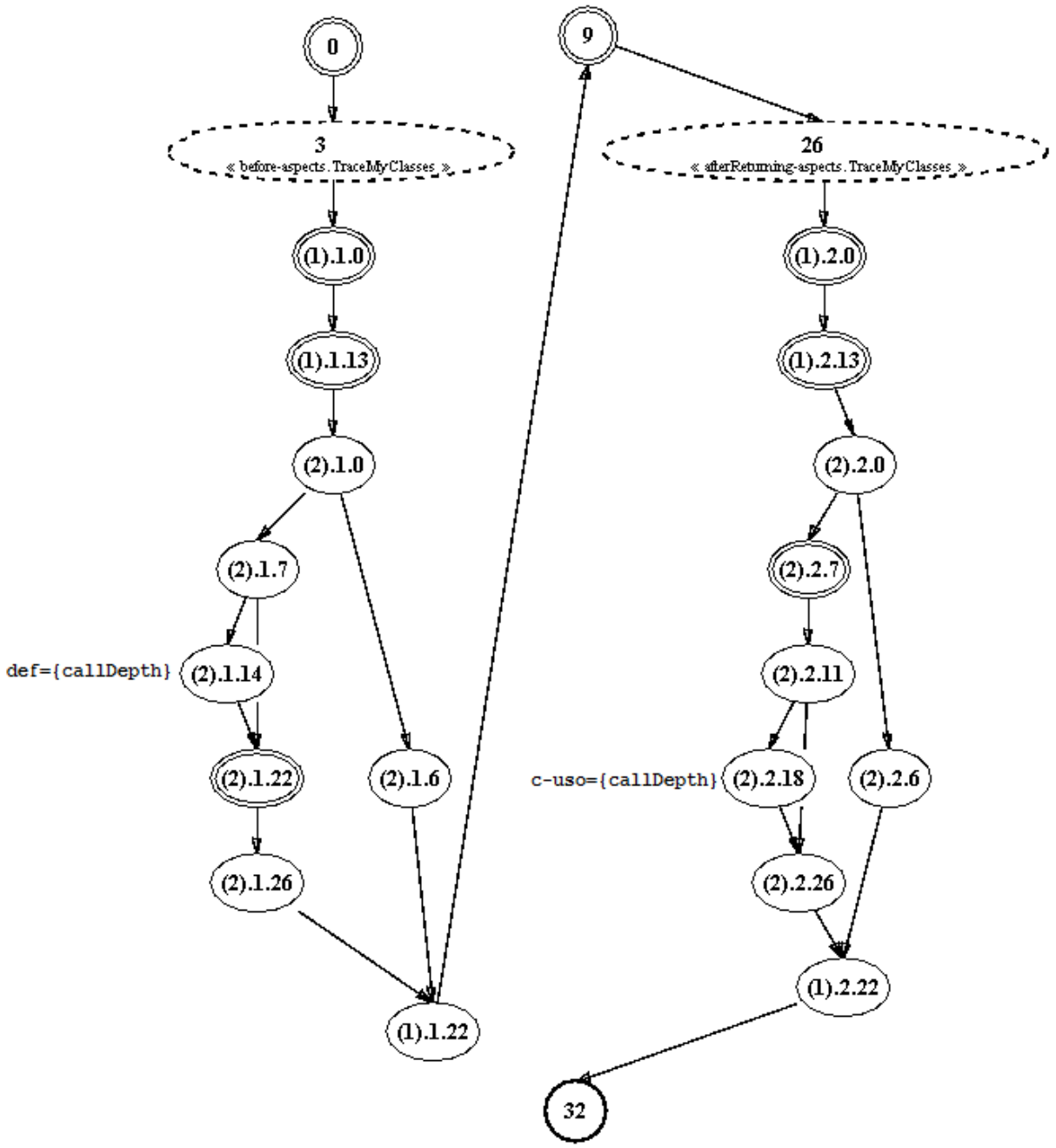


Figura 5.5: Grafo $\mathcal{CODU}$ do método `area` no nível 2 de profundidade.

O $\mathcal{CODU}$ do método `area`, no nível 3 de profundidade, é formado pelos $\mathcal{AODU}$s representados no nível 2 de profundidade, além dos métodos `printEntering` e `printExiting`.

Na Figura 5.6 é mostrado o $\mathcal{CODU}$ do método `area` representando interações nos níveis 1, 2 e 3 de profundidade.

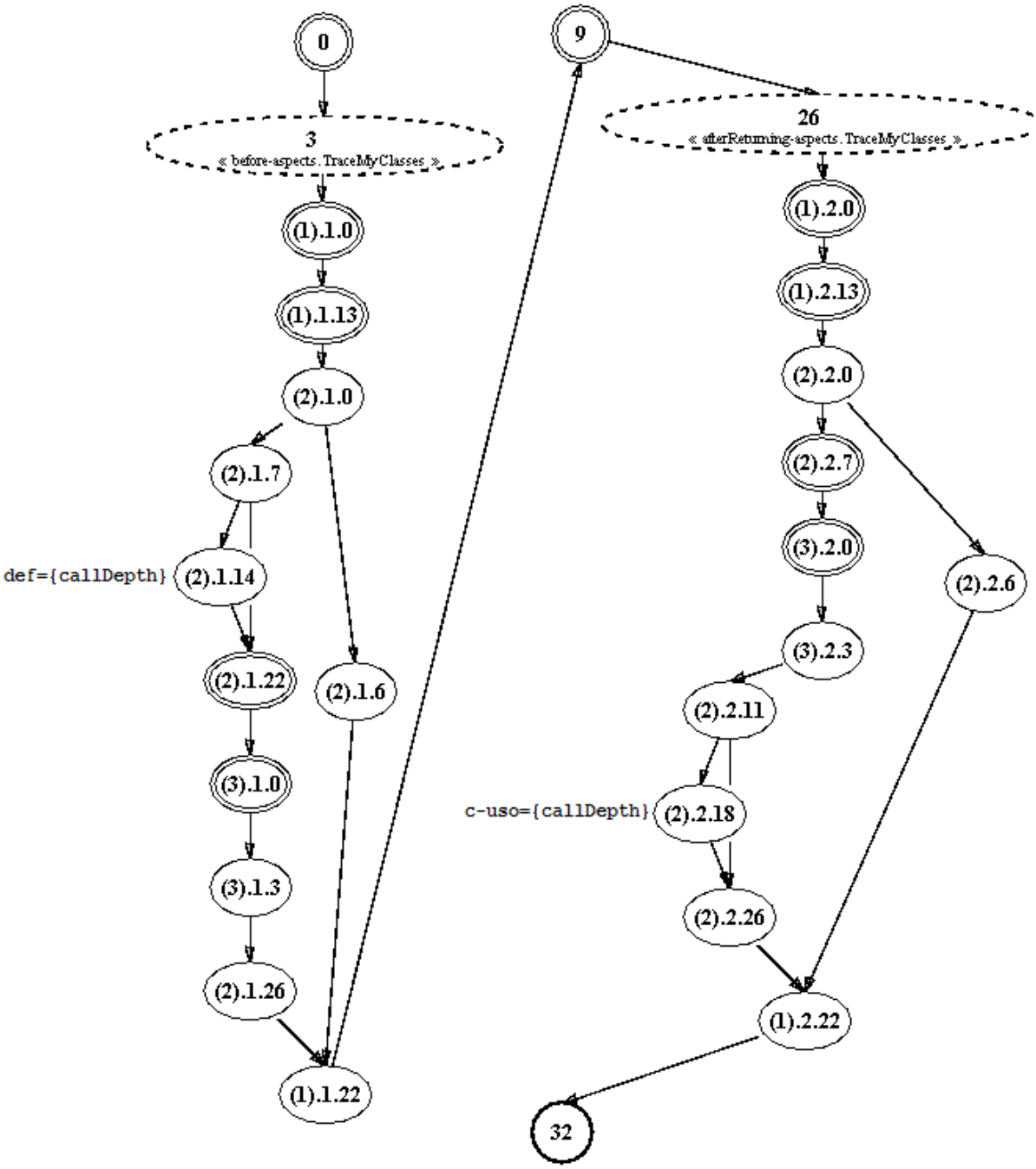


Figura 5.6: Grafo $\mathcal{CODU}$ do método `area` no nível 3 de profundidade.

O $\mathcal{CODU}$ do método `area`, no nível 4 de profundidade, é formado pelo $\mathcal{AODU}$ do próprio método `area` e pelos $\mathcal{AODU}$s dos adendos `(aspects.Trace) before():MyMethod()` e `(aspects.Trace) after() returning():MyMethod()`, e dos métodos `traceEntry`, `traceExit`, `printEntering`, `printExiting` e `printIndent`. Na Figura 5.7 é mostrado o

$\mathcal{C}o\mathcal{DU}$ considerando todas as unidades que pertencem à cadeia de chamadas/entrecortes do método `area`.

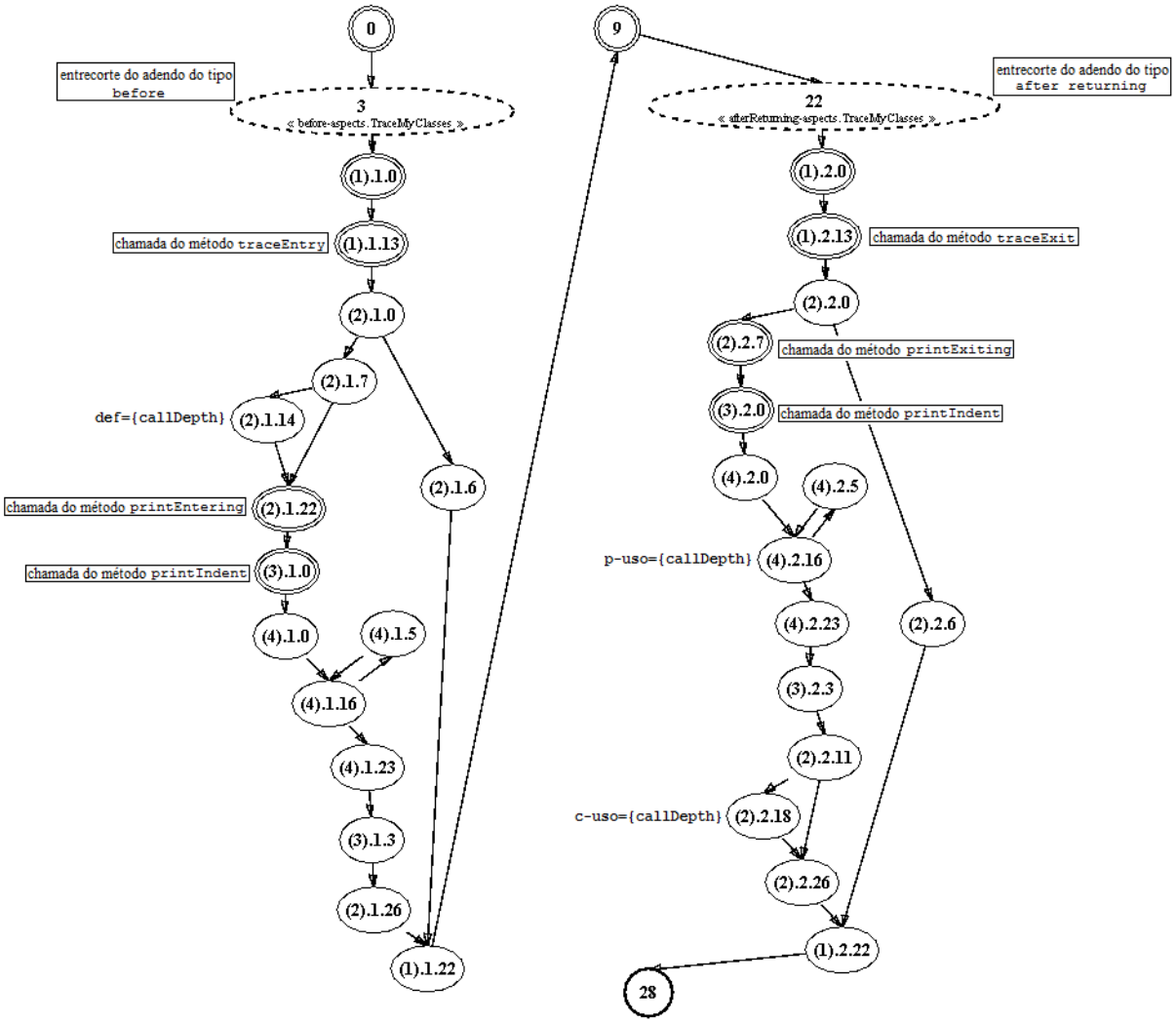


Figura 5.7: Grafo $\mathcal{C}o\mathcal{DU}$ do método `area` no nível 4 de profundidade.

Portanto, podemos definir o grafo $\mathcal{C}o\mathcal{DU}$ do método `area` no nível 4 de profundidade, mostrado na Figura 5.7, como um $\mathcal{C}o\mathcal{DU}$ $(\texttt{shape.Circle.area}, 4) = (N, E, S, T, I, I_s, R)$, tal que:

- u_0 =`shape.Circle.area`, $u_{1.1}$ =`aspects.Trace.before():MyMethod()`, $u_{1.2}$ =`aspects.Trace.after() returning():MyMethod()`, $u_{2.1}$ =`shape.Circle.traceEntry`, $u_{2.2}$ =`shape.Circle.traceExit`, $u_{3.1}$ =`shape.Circle.printEntering`, $u_{3.2}$ =`shape.Circle.printExiting`, $u_{4.1}$ =`shape.Circle.printIndent` e $u_{4.2}$ =`shape.Circle.printIndent`.

- $N = N_0 \cup N_{1.1} \cup N_{1.2} \cup N_{2.1} \cup N_{2.2} \cup N_{3.1} \cup N_{3.2} \cup N_{4.1} \cup N_{4.2}$, tal que:

- $N_0 = \{0; 3; 9; 22; 28\}$
- $N_{1.1} = \{(1).1.0; (1).1.13; (1).1.22\}$
- $N_{1.2} = \{(1).2.0; (1).2.13; (1).2.22\}$
- $N_{2.1} = \{(2).1.0; (2).1.6; (2).1.7; (2).1.14; (2).1.22\}$
- $N_{2.2} = \{(2).2.0; (2).2.6; (2).2.7; (2).1.11; (2).1.18; (2).2.26\}$
- $N_{3.1} = \{(3).1.0; (3).1.3\}$
- $N_{3.2} = \{(3).2.0; (3).2.3\}$
- $N_{4.1} = \{(4).1.0; (4).1.5; (4).1.16; (4).1.23\}$
- $N_{4.2} = \{(4).2.0; (4).2.5; (4).2.16; (4).2.23\}$
- $N' = N - N_0 = \{(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.6; (2).1.7; (2).1.14; (2).1.22; (2).2.0; (2).2.6; (2).2.7; (2).1.11; (2).1.18; (2).2.26; (3).1.0; (3).1.3; (3).2.0; (3).2.3; (4).1.0; (4).1.5; (4).1.16; (4).1.23; (4).2.0; (4).2.5; (4).2.16; (4).2.23\}$

- $E = E'_0 \cup E'_{1.1} \cup E'_{1.2} \cup E'_{2.1} \cup E'_{2.2} \cup E'_{3.1} \cup E'_{3.2} \cup E'_{4.1} \cup E'_{4.2} \cup E_{I_0} \cup E_{I_{1.1}} \cup E_{I_{1.2}} \cup E_{I_{2.1}} \cup E_{I_{2.2}} \cup E_{I_{3.1}} \cup E_{I_{3.2}}$, tal que:
  - $E'_0 = E_0 - E_{x_0} = \{(0,3); (3,9); (9,28)\} - \{(3,9); (9,28)\} = \{(0,3)\}$
  - $E'_{1.1} = E_{1.1} - E_{x_{1.1}} = \{((1).1.0,(1).1.13); ((1).1.13,(1).1.22)\} - -\{((1).1.13,(1).1.22)\} = \{((1).1.0,(1).1.13)\}$
  - $E'_{1.2} = E_{1.2} - E_{x_{1.2}} = \{((1).2.0,(1).2.13); ((1).2.13,(1).2.22)\} - -\{((1).2.13,(1).2.22)\} = \{((1).2.0,(1).2.13)\}$
  - $E'_{2.1} = E_{2.1} - E_{x_{2.1}} = \{((2).1.0,(2).1.6); ((2).1.0,(2).1.7); ((2).1.7,(2).1.14); ((2).1.7,(2).1.22); ((2).1.14,(2).1.22); ((2).1.22,(2).1.26)\} - -\{((2).1.22,(2).1.26)\} = \{((2).1.0,(2).1.6); ((2).1.0,(2).1.7); ((2).1.7,(2).1.14); ((2).1.7,(2).1.22); ((2).1.14,(2).1.22)\}$
  - $E'_{2.2} = E_{2.2} - E_{x_{2.2}} = \{((2).2.0,(2).2.6); ((2).2.0,(2).2.7); ((2).2.7,(2).2.11); ((2).2.11,(2).2.18); ((2).2.11,(2).2.26); ((2).2.18,(2).2.26)\} - -\{((2).2.7,(2).2.11)\} = \{((2).2.0,(2).2.6); ((2).2.0,(2).2.7); ((2).2.11,(2).2.18); ((2).2.11,(2).2.26); ((2).2.18,(2).2.26)\}$
  - $E'_{3.1} = E_{3.1} - E_{x_{3.1}} = \{((3).1.0,(3).1.3)\} - \{((3).1.0,(3).1.3)\} = \{\}$
  - $E'_{3.2} = E_{3.2} - E_{x_{3.2}} = \{((3).2.0,(3).2.3)\} - \{((3).2.0,(3).2.3)\} = \{\}$
  - $E'_{4.1} = E_{4.1} - E_{x_{4.1}} = \{((4).1.0,(4).1.16); ((4).1.16,(4).1.5); ((4).1.5,(4).1.16); ((4).1.16,(4).1.23)\} - \{\} = \{((4).1.0,(4).1.16); ((4).1.16,(4).1.5); ((4).1.5,(4).1.16); ((4).1.16,(4).1.23)\}$

- $E'_{4.2} = E_{4.2} - E_{x_{4.2}} = \{((4).2.0,(4).2.16);\ ((4).2.16,(4).2.5);\ ((4).2.5,(4).2.16);\ ((4).2.16,(4).2.23)\} - \{\} = \{((4).2.0,(4).2.16);\ ((4).2.16,(4).2.5);\ ((4).2.5,(4).2.16);\ ((4).2.16,(4).2.23)\}$
- $E_{I_0} = \{(3,(1).1.0);\ ((1).1.22,9);\ (22,(1).2.0);\ ((1).2.22,28)\}$
- $E_{I_{1.1}} = \{((1).1.13,(2).1.0);\ ((2).1.26,(1).1.22)\}$
- $E_{I_{1.2}} = \{((1).2.13,(2).2.0);\ ((2).2.26,(1).2.22)\}$
- $E_{I_{2.1}} = \{((2).1.22,(3).1.0);\ ((3).1.3,(2).1.26)\}$
- $E_{I_{2.2}} = \{((2).2.7,(3).2.0);\ ((3).2.3,(2).2.11)\}$
- $E_{I_{3.1}} = \{((3).1.0,(4).1.0);\ ((4).1.23,(3).1.3)\}$
- $E_{I_{3.2}} = \{((3).2.0,(4).2.0);\ ((4).2.23,(3).2.3)\}$
- $E' = \{((1).1.0,(1).1.13);\ ((1).2.0,(1).2.13);\ ((2).1.0,(2).1.6);\ ((2).1.0,(2).1.7);\ ((2).1.7,(2).1.14);\ ((2).1.7,(2).1.22);\ ((2).1.14,(2).1.22);\ ((2).2.0,(2).2.6);\ ((2).2.0,(2).2.7);\ ((2).2.11,(2).2.18);\ ((2).2.11,(2).2.26);\ ((2).2.18,(2).2.26);\ ((2).2.0,(2).2.6);\ ((2).2.0,(2).2.7);\ ((2).2.11,(2).2.18);\ ((2).2.11,(2).2.26);\ ((2).2.18,(2).2.26);\ ((4).1.0,(4).1.16);\ ((4).1.16,(4).1.5);\ ((4).1.5,(4).1.16);\ ((4).1.16,(4).1.23);\ ((4).2.0,(4).2.16);\ ((4).2.16,(4).2.5);\ ((4).2.5,(4).2.16);\ ((4).2.16,(4).2.23);\ ((1).1.13,(2).1.0);\ ((2).1.26,(1).1.22);\ ((1).2.13,(2).2.0);\ ((2).2.26,(1).2.22);\ ((2).1.22,(3).1.0);\ ((3).1.3,(2).1.26);\ ((2).2.7,(3).2.0);\ ((3).2.3,(2).2.11);\ ((3).1.0,(4).1.0);\ ((4).1.23,(3).1.3);\ ((3).2.0,(4).2.0);\ ((4).2.23,(3).2.3)\}$

- $S = \{s_0, s_{1.1}, s_{1.2}, s_{2.1}, s_{2.2}, s_{3.1}, s_{3.2}, s_{4.1}, s_{4.2}\}$, tal que:
  - $s_0 = 0$
  - $s_{1.1} = (1).1.0$
  - $s_{1.2} = (1).2.0$
  - $s_{2.1} = (2).1.0$
  - $s_{2.2} = (2).2.0$
  - $s_{3.1} = (3).1.0$
  - $s_{3.2} = (3).2.0$
  - $s_{4.1} = (4).1.0$
  - $s4.2 = (4).2.0$

- $T = T_0 \cup T_{1.1} \cup T_{1.2} \cup T_{2.1} \cup T_{2.2} \cup T_{3.1} \cup T_{3.2} \cup T_{4.1} \cup T_{4.2}$, tal que:

  - $T_0 = \{28\}$
  - $T_{1.1} = \{(1).1.22\}$
  - $T_{1.2} = \{(1).2.22\}$
  - $T_{2.1} = \{(2).1.6, (2).1.26\}$
  - $T_{2.2} = \{(2).2.6, (2).2.26\}$
  - $T_{3.1} = \{(3).1.3\}$
  - $T_{3.2} = \{(3).2.3\}$
  - $T_{4.1} = \{(4).1.23\}$
  - $T_{4.2} = \{(4).2.23\}$

- $I = I_0 \cup I_{1.1} \cup I_{1.2} \cup I_{2.1} \cup I_{2.2} \cup I_{3.1} \cup I_{3.2} \cup I_{4.1} \cup I_{4.2}$, tal que:
  - $I_0 = \{0, 3, 9, 22\}$
  - $I_{1.1} = \{(1).1.0, (1).1.13\}$
  - $I_{1.2} = \{(1).2.0, (1).2.13\}$
  - $I_{2.1} = \{(2).1.22\}$
  - $I_{2.2} = \{(2).2.7\}$
  - $I_{3.1} = \{(3).1.0\}$
  - $I_{3.2} = \{(3).2.0\}$
  - $I_{4.1} = \{\}$
  - $I_{4.2} = \{\}$

- $I_s = I - I_r = \{0, 3, 9, 22, (1).1.0, (1).1.13, (1).2.0, (1).2.13, (2).1.22, 2).2.7, (3).1.0, (3).2.0\} - \{(1).1.0, (1).2.0\} = \{0, 3, 9, 22, (1).1.13, (1).2.13, (2).1.22, 2).2.7, (3).1.0, (3).2.0\}$

- $R = \{9, 28, (1).1.22, (1).2.22, (2).1.26, (2).2.26, (3).1.3, (3).2.3\}$

5.3.1 Casos Especiais de Construção do Grafo $\mathcal{C}o\mathcal{DU}$

Existem algumas situações especiais das linguagens Java e AspectJ que merecem um tratamento específico e que resultam em restrições que devem ser impostas na montagem do grafo. Essas restrições são impostas para possibilitar o teste de situações específicas, melhorar o entendimento do fluxo de execução da unidade sob teste e tentar representar da melhor forma possível o comportamento real do programa. A seguir são discutidas alguns desses casos especiais que são considerados na montagem do grafo $\mathcal{C}o\mathcal{DU}$.

- **Chamadas a métodos de biblioteca:** No caso de uma chamada a um método de alguma biblioteca em uma unidade chamadora, este método chamado será tratado como uma linha de código normal dentro de um bloco sequencial de código, ou seja, este método não deve ser expandido e integrado. Com este tratamento, chamada a métodos de biblioteca não geram um nó individual no grafo, apenas compõem, com outras linhas de código, um nó referente a um bloco sequencial. Na Figura 5.8 é mostrado o código-fonte e o respectivo grafo $\mathcal{CODU}$ em uma situação em que a chamada ao método de biblioteca (linha 5) não gera um nó de chamada, mas é tratado como uma linha de código normal representada pelo nó 4.

```
1  public class C1 {
2    public void m1(){
3      m2();
4
5      System.out.println();
6
7      m3();
8    }
9
10   public void m2(){}
11
12   public void m3(){}
13 }
```

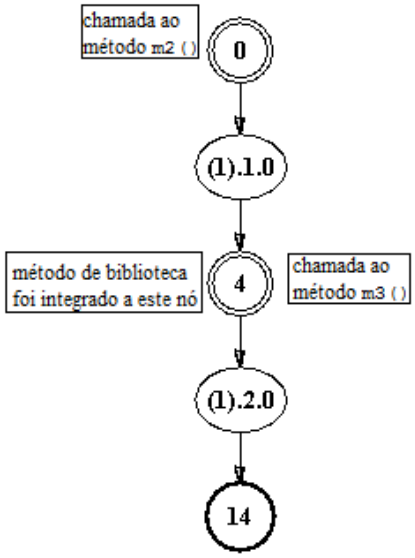


Figura 5.8: Tratamento especial dado no caso de chamadas a métodos de biblioteca.

- **Recursões:** Uma chamada recursiva será integrada no grafo apenas uma vez, ou seja, o método recursivo será representado duas vezes no $\mathcal{CODU}$. Com isso é possível testar um método/adendo em seu próprio contexto. Na Figura 5.9 é mostrado o $\mathcal{CODU}$ do método `m1()`, no qual existe uma chamada ao método recursivo `m2()`. É possível notar que os nós do grafo $\mathcal{CODU}$ que contém o rótulo iniciando com o prefixo (1).1 representam a primeira execução do método `m2()` e os nós com rótulos iniciando com o prefixo (2).1 representam a execução da recursão. Ou seja, o método `m2()` pode ser testado pelo menos uma vez em seu próprio contexto. Vale ressaltar que o nó (2).1.19 que representa outra chamada recursiva a `m2()` não foi expandido.

```
1  public class C1 {
2    public void m1(){
3      m2();
4    }
5
6    public void m2(){
7      Random generator = new Random();
8      int x = generator.nextInt();
9
10     if (x>50)
11       m2();
12     else
13       m3();
14   }
15
16   public void m3(){}
17 }
```

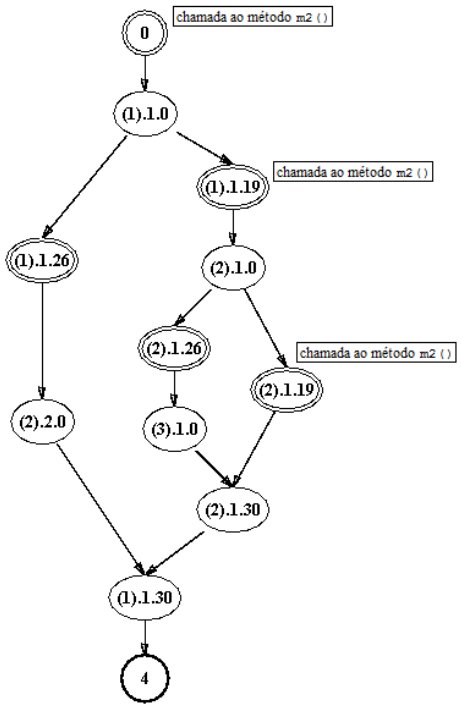


Figura 5.9: Tratamento especial dado ao $\mathcal{C}o\mathcal{DU}$ no caso de recursão.

- **Polimorfismo (sobrescrita):** A sobrescrita (*overriding*) acontece quando o compilador não tem como decidir, em tempo de compilação, qual método polimórfico será executado. Dessa forma, a decisão sobre qual dos métodos deve ser selecionado só pode ser tomada em tempo de execução pelo mecanismo de ligação tardia (*late binding*). Portanto, a análise do *bytecode* não é suficiente para dizer que método deve ser integrado. A solução adotada para esse problema é permitir que o testador escolha quais métodos polimórficos serão integrados no caso de ligação tardia, de forma semelhante ao trabalho de Neves (2009), já que a criação de todas as combinações possíveis (semelhante ao trabalho de Franchin (2007)), considerando todas as profundidades de interação, poderia gerar uma explosão na quantidade de grafos gerados. Na Figura 5.10 é mostrado um caso em que o método `main()` chama o método polimórfico `m1()` configurando um caso de sobrescrita. Nesse caso, nota-se que a chamada ao método polimórfico no grafo (nó 44) possui o indicador de chamada polimórfica <P> e todas as possíveis situações foram representadas, ou seja, o método `m1()` da classe `SuperC`, o método `m1()` da classe `C2` ou o método `m1()` da classe `C3`. Vale ressaltar que nesse caso foram escolhidas todas as possíveis representações de `m1()`, mas seria possível selecionar outras combinações para serem representadas no $\mathcal{C}o\mathcal{DU}$ resultante.

```
1  public class SuperC {
2    public SuperC(){}
3
4    public void m1() {}
5  }
6
7  public class C2 extends SuperC{
8    public void m1() {
9      Random generator = new Random();
10     int x = generator.nextInt();
11
12     if (x>10)
13       System.out.println("X é maior que 10");
14     else
15       System.out.println("X é menor ou
16                                  igual a 10");
17   }
18 }
19
20 public class C3 extends SuperC{
21   public void m1() {}
22 }
23
24 public class C1 {
25   public void main(){
26     SuperC[] obj = new SuperC[2];
27     obj[0] = new C2();
28     obj[1] = new C3();
29
30     for(int i=0;i<2;i++){
31       obj[i].m1();
32     }
33   }
34 }
```

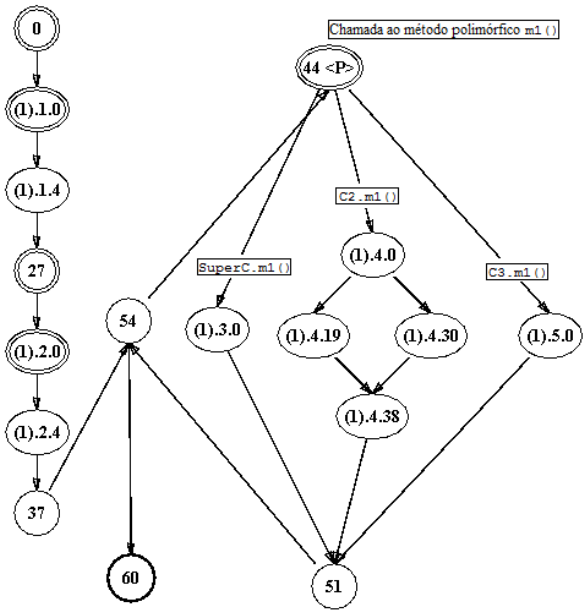


Figura 5.10: Tratamento especial dado para o caso de polimorfismo (sobrescrita).

- **Adendo do tipo `around` e comando `proceed()`:** O `around` é um tipo de adendo que é executado em substituição ao ponto de junção, ou seja, o método (adendo) entrecortado é substituído pelo adendo do tipo `around`. Opcionalmente, este tipo de adendo pode executar o comando `proceed()`, que desvia o fluxo de execução do adendo para o ponto de junção entrecortado. Para representar ambas as situações no grafo, será utilizado o mesmo conceito do funcionamento do `around` e do `proceed()`, ou seja, na representação do grafo de uma unidade entrecortada por um conjunto de junção que possua o adendo `around`, a unidade será substituída pelo adendo `around`. No caso da execução do comando `proceed()` pelo adendo, o grafo da unidade referente ao ponto de junção entrecortado será integrado ao grafo do adendo como se tivesse sido feita uma chamada a essa unidade. Na Figura 5.11 é possível notar que em vez de integrar o grafo $\mathcal{AODU}$ do método `m2()` no nó de interação (nó 0) o grafo $\mathcal{AODU}$ do adendo `around` foi integrado. Além disso, é possível notar que a chamada ao comando `proceed()` (nó (2).1.0) faz com que o grafo $\mathcal{AODU}$ do método `m2()` seja integrado (nó (3).1.0).

```
1  public class C1 {
2    public void m1() {
3      m2();
4    }
5
6    public void m2(){
7      System.out.println("Método m2");
8    }
9  }
10
11 public aspect A1 {
12   pointcut chamaM2(): call(void m2());
13
14   void around(): chamaM2(){
15     Random generator = new Random();
16     int x = generator.nextInt();
17
18     if (x>50)
19       System.out.
20          println("Substituindo
21                          m2 pelo around");
22     else
23       proceed();
24   }
25 }
```

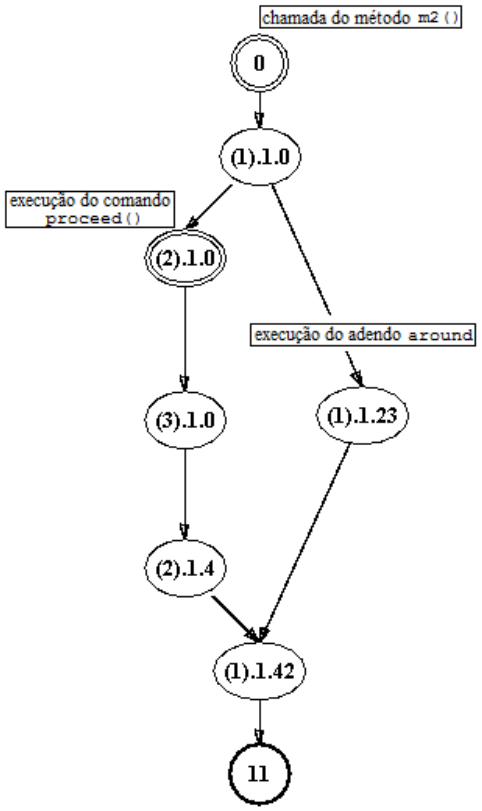


Figura 5.11: Representação de adendos do tipo `around` e do comando `proceed()`.

- **Chamadas (Entrecortes) múltiplas (os) a (de) um mesmo método (adendo) em uma unidade integrada:** Caso haja mais de uma (um) chamada (entrecorte) a (de) um mesmo método (adendo) na unidade integrada, estes métodos (adendos) devem ser integrados separadamente no grafo. Isto é, para cada método (adendo) pertencente à cadeia de chamada, o respectivo grafo de unidade desse método (adendo) deve ser integrado ao grafo, independentemente do número de vezes que é invocado pela unidade chamadora (entrecortada). Na Figura 5.12 são mostrados o um código-fonte exemplo e o grafo $\mathcal{CODU}$ do método `m1()`. O método `m1()` realiza duas chamadas ao método `m2()` (linhas 4 e 10) e ao método `m3()` (linhas 7 e 13). É possível notar que o grafo $\mathcal{CODU}$ integra duas vezes o método `m2()` nos nós 0 e 24 e duas vezes o método `m3()` nos nós 12 e 36.

```
1  public class C1 {
2    public void m1(){
3      System.out.println("Chamando m2..");
4      m2();
5
6      System.out.println("Chamando m3..");
7      m3();
8
9      System.out.println("Chamando m2 novamente..");
10     m2();
11
12     System.out.println("Chamando m3 novamente..");
13     m3();
14   }
15
16   public void m2(){}
17
18   public void m3(){}
19 }
```

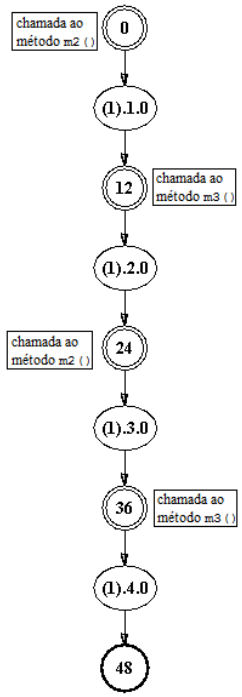


Figura 5.12: Tratamento especial dado para chamadas múltiplas a um mesmo método.

5.4 Modelo de Fluxo de Dados

Um modelo de fluxo de dados é utilizado para determinar o que caracteriza uma definição ou um uso de uma variável em uma sentença ou instrução. Esta dissertação utiliza o mesmo modelo de fluxo de dados para *bytecode* Java utilizado por Franchin (2007). Porém, antes de descrever o modelo proposto, é importante ressaltar que, para facilitar a sua elaboração, são utilizados apenas classes e métodos, desconsiderando aspectos e adendos, já que depois da compilação e combinação, no *bytecode* resultante, os programas AspectJ são idênticos aos programas Java. Dessa forma, nas definições que seguem, quando houver uma referência a métodos, entenda-se métodos e adendos. O mesmo para classes e aspectos.

O modelo proposto foi adaptado de Vincenzi que, primeiramente, classificou as instruções *bytecode* Java, relacionando cada uma com o fluxo de dados envolvido, ou seja, se uma instrução caracteriza uma definição, um uso, uma definição e uso ou uma instrução que não tem implicações (Vincenzi, 2004). Em seguida, Vincenzi fez algumas suposições para identificação de definições e usos dos diferentes tipos de variáveis em Java. Uma variável é um local de armazenamento que possui um tipo associado que pode ser primitivo ou de referência. Uma variável sempre contém um valor compatível com seu tipo, ou seja, uma variável primitiva sempre armazena valores do exato tipo primitivo e uma variável de referência pode armazenar ou uma referência nula ou uma referência para qualquer objeto

cuja classe (ou array) seja compatível com o tipo da variável. Na linguagem Java, pode-se encontrar cinco tipos de variáveis, a saber: variáveis locais, variáveis agregadas, atributos estáticos, atributos de instância e parâmetros. Os tipos de variáveis são discutidos a seguir.

1. Variáveis locais são unicamente visíveis dentro do escopo dos métodos nos quais elas são declaradas. Elas podem ser do tipo primitivo (variáveis primitivas) ou do tipo de referência (variáveis de referência). Uma variável local não é iniciada até que sua sentença de declaração seja executada.

2. Variáveis Agregadas (ou elementos de array) são variáveis não nomeadas que são criadas e iniciadas sempre que um novo array é criado. Pode existir também array de arrays, que são denominados multi-arrays (matrizes ou vetores multidimensionais). Para tratar variáveis agregadas foi utilizada a abordagem proposta por Horgan e London, que consideram uma variável agregada unidimensional como sendo uma única localização de memória de tal modo que, quando ocorre uma definição (uso) de qualquer um de seus elementos, considera-se que a variável agregada em si é que está sendo definida (usada) e não apenas o elemento particular (Horgan e London, 1991).

3. Atributos estáticos são atributos declarados em uma classe utilizando a palavra reservada `static`. Eles são criados e recebem um valor inicial quando a classe é carregada. O valor de um atributo estático é associado a classe onde ele é definido e é compartilhado com todas as instâncias da classe.

4. Atributos de instância são atributos declarados em uma classe sem utilizar a palavra reservada `static`. Se uma classe *Cl* tem um atributo de instância a, então esse atributo é criado e iniciado assim que um novo objeto da classe *Cl* é criado. Diferentemente dos atributos estáticos, os atributos de instância armazenam valores que são específicos para cada instância da classe.

5. Parâmetros formais representam valores de argumentos passados para um método. Para todo parâmetro presente na declaração do método, uma nova variável local é criada no corpo do método toda vez que ele for invocado. Um parâmetro formal é iniciado com o correspondente valor de argumento passado na chamada.

A partir daí, são definidas as seguintes suposições para identificação da definição e uso de variáveis:

1. Não é considerado o uso de um literal *c*.

2. A definição ou uso de uma variável primitiva p é considerada como sendo unicamente a definição ou uso de p.

3. A definição de uma variável de referência r pode envolver uma referência nula ou uma referência a um objeto (que esta sendo construído ou que já existe na memória), que pode ser um array ou uma instância de uma determinada classe. Assim, a definição de uma variável de referência r envolvendo uma referência nula é considerada como sendo somente a definição de r. A definição de uma variável de referência r envolvendo uma referência a um objeto é considerada como sendo a definição de r e, no caso de um array, a definição das variáveis agregadas *r[]* ou, no caso de uma instância da classe *Cl* que possua f como atributo de instância, a definição do atributo de instância *r.f*. O uso de uma variável de referência r é considerada como sendo unicamente o uso de r.

4. Variáveis agregadas são consideradas como sendo uma única posição de memória. Assim, a definição de uma variável agregada *a[]*, que é um elemento de um array referenciado pela variável de referência a, é considerada como sendo a definição de *a[]* e a definição do array referenciado por a (representado como definição de a). O uso de uma variável agregada *a[]* é considerado como sendo o uso da variável de referência a, que permite o acesso ao elemento, e o uso da variável agregada *a[]*.

5. A definição de um atributo de instância f de uma variável de referência r do tipo da classe *Cl* é considerada como sendo o uso da variável de referência r, que permite o acesso ao atributo, a definição do atributo de instância (representado por $r.f$) e a definição do objeto referenciado pela variável de referência r (representado como definição de r). O uso de um atributo de instância f é considerado como sendo o uso da variável de referência r (para acessar o atributo) e uso do atributo de instância *r.f*.

6. O acesso aos atributos estáticos (ou de classe) é feito sem a necessidade de se ter uma variável de referência. Assim, a definição ou uso de qualquer atributo estático s de uma classe *Cl* é considerada como sendo somente a definição ou uso do atributo estático representado por *Cl.s*. Mesmo que a definição ou uso do atributo estático seja feita por meio de uma variável de referência r do tipo da classe *Cl*, no nível de *bytecode*, tal variável de referência é automaticamente convertida no nome da classe, não sendo caracterizado um uso da variável de referência nesse caso.

7. Na invocação de um método de instância *mi*, tal como $r.mi(t_1; t_2; \cdots; t_n)$, onde t_i é um parâmetro que pode ser um literal ou um dos tipos de variáveis, considera-se que

ocorre um uso da variável de instância r e uso dos parâmetros $t_1, t_2, \cdots, t_n$ segundo as regras descritas nos itens de 1 a 6.

8. Em uma atribuição de uma expressão a uma variável v da forma $v = e_1\ op\ e_2\ op\ \cdots\ e_n$, onde e_i é um item da expressão que pode ser um literal ou um dos tipos de variáveis e op é um operador, considera-se que ocorre uso de $e_1, e_2, \cdots, e_n$ conforme as regras descritas nos itens 1 a 6, e definição de v.

Na Tabela 5.1 são apresentados alguns exemplos de sentenças genéricas de java e as correspondentes implicações no fluxo de dados. Porém, para o entendimento dos exemplos, considere os seguintes termos:

- c: um literal;
- n: um valor primitivo do tipo inteiro;
- p: uma variável primitiva;
- a: uma variável de referência a um array;
- *a[]*: uma variável agregada (ou elemento do array) que pode ser do tipo primitivo ou referência;
- *Cl*: uma classe que possui um atributo de instância *f*, um atributo estático *s*, um método de instância *mi* e um método estático *ms*.
- *Cl.s*: um atributo estático da classe *Cl* que pode ser do tipo primitivo ou referência;
- r: uma variável de referência a um objeto do tipo da classe *Cl*;
- $r.f$: um atributo de instância de r que pode ser do tipo primitivo ou referência;
- *null*: referência a nenhum objeto ou array;
- v: uma variável que pode ser p, a, *a[]*, r, $r.f$ ou *Cl.s*;
- t: um parâmetro que pode ser c ou v;
- e: uma expressão simples ou complexa;

Tabela 5.1: Exemplos de sentenças Java genéricas e suas implicações no fluxo de dados (Franchin, 2007).

Tipo	Sentença	Implicações no fluxo de dados
1	$p = c$	definição de p
2	$p = v$	uso de v e definição de p
3	$v = p$	uso de p e definição de v
4	a = *new type[n]*	definição do array referenciado por a (representado como definição de a) e definição de *a[]*
5	a = *null*	definição de a
6	*a1 = a2*	uso de a_2, definição de a_1 e definição de a_1*[]*
7	*a[n] = c*	uso de a, definição de *a[]*, definição do array referenciado por a (representado como definição de a)
8	*a[n] = v*	uso de v, uso de a, definição de *a[]* e definição do array referenciado por a (representado como definição de a)
9	*v = a[n]*	uso de a, uso de *a[]* e definição de v
10	r = *null*	definição de r
11	r = *new Cl()*	definição do objeto referenciado por r (representado como definição de r) e definição de *r.f*
12	r = *new Cl*$(t_1, \cdots, t_n)$	uso de $t_1, \cdots$, uso de t_n, definição do objeto referenciado por r (representado como definição de r) e definição de *r.f*
13	*r1 = r2*	uso de r_2, definição de r_1 e definição de r_1*.f*
14	*r.f = c*	uso de r, definição de *r.f* e definição do objeto referenciado por r (representado como definição de r)
15	*r.f = v*	uso de v, uso de r, definição de *r.f* e definição do objeto referenciado por r (representado como definição de r)
16	*v = r.f*	uso de r, uso de *r.f* e definição de v
17	*Cl.s = c*	definição de *Cl.s*
18	*Cl.s = v*	uso de v e definição de *Cl.s*
19	*v = Cl.s*	uso de *Cl.s* e definição de v
20	*v = Cl.ms(c)*	definição de v
21	v_1 = *Cl.ms*(v_2)	uso de v_2 e definição de v_1
22	v = *Cl.ms*$(t_1, \cdots, t_n)$	uso de $t_1, \cdots$, uso de t_n e definição de v
23	*v = r.mi(c)*	uso de r e definição de v
24	v_1 = *r.mi*(v_2)	uso de r, uso de v_2 e definição de v_1
25	v = *r.mi*$(t_1, \cdots, t_n)$	uso de r, uso de $t_1, \cdots$, uso de t_n e definição de v
26	$v = e_1$ *op* $\cdots$ e_n	uso dos fatores de e_1, $\cdots$, uso dos fatores de e_n e definição de v

Além das sentenças usadas para ilustrar o modelo de fluxo de dados na Tabela 5.1, podem ser consideradas todas as sentenças válidas na linguagem, como por exemplo: *r.a[n] = p*$_1$ *+ Cl.ms(c,p*$_2$*)*. Neste caso tem-se uso de p_1, uso de p_2, uso de r, uso de a, definição de *a[]*, definição do array referenciado por a e definição do objeto referenciado por r (representado como definição de r). Essa sentença é ilustrada pelas linhas 26 e 15 da Tabela 5.1.

5.5 Critérios de Teste

Os critérios de teste são de fundamental importância, pois fornecem um mecanismo para sistematizar a seleção de casos de teste e a avaliação da qualidade de conjuntos de teste

já definidos (Vincenzi, 2004). Nesta seção são apresentadas as definições dos critérios de teste estrutural de integração contextual, os quais podem ser utilizados para derivar requisitos de testes baseados no fluxo de controle e no fluxo de dados. Após a definição de cada critério é mostrado um exemplo do conjunto de requisitos de teste gerado por ele.

5.5.1 Critérios de Fluxo de Controle

Antes da definição dos critérios, considere: T um conjunto de casos de teste para um programa P (sendo que o grafo $\mathcal{CODU}$ é o grafo de fluxo de controle/dados integrado das unidades de P), e seja Π o conjunto de caminhos exercitados por T. Diz-se que um nó x está incluído em Π se Π contém um caminho $(y_1, \cdots, y_m)$ tal que $x = y_j$ para algum j, $1 \leq j \leq m$. Similarmente, uma aresta (x_1, x_2) é incluída em Π se Π contém um caminho $(y_1, \cdots, y_m)$ tal que $x_1 = y_j$ e $x_2 = y_{j+1}$ para algum $j, 1 \leq j \leq m-1$. Assim, os critérios de fluxo de controle são definidos como segue:

- **todos-nós-integrados-N_d**: Π satisfaz o critério todos-nós-integrados-N_d se todo nó integrado $n \in N'$ de um grafo $\mathcal{CODU}$ com profundidade d previamente definida está incluído em Π. Em outras palavras, este critério requer que cada nó integrado de um grafo $\mathcal{CODU}$ seja exercitado ao menos uma vez por algum caso de teste de T.

- **todas-arestas-integradas-N_d**: Π satisfaz o critério todas-arestas-integradas-N_d se toda aresta integrada $e \in E'$ de um grafo $\mathcal{CODU}$ com profundidade d previamente definida está incluída em Π. Em outras palavras, este critério requer que cada aresta integrada de um grafo $\mathcal{CODU}$ seja exercitado ao menos uma vez por algum caso de teste de T.

Exemplo de Aplicação

Na Tabela 5.2 são apresentados os requisitos de teste para os critérios todos-nós-integrados-N_d e todas-arestas-integradas-N_d para o método `area` da classe `Circle`, apresentado no Capítulo 5, em todos os níveis de profundidade até a profundidade máxima, ou seja, $1 \leq d \leq 4$. Os requisitos são gerados a partir dos $\mathcal{CODU}$s mostrados nas Figuras 5.4, 5.5, 5.6 e 5.7.

5.5.2 Critério de Fluxo de Dados

Em alguns casos, cobrir todos os comandos e desvios de unidades integradas não é suficiente para revelar um erro de integração. Dessa forma, o critério todos-usos foi revisado no contexto do teste estrutural de integração contextual de programas OO e OA para requerer que toda associação entre a definição de uma variável de comunicação e seus subsequentes usos sejam exercitados por um casos de teste, tomando-se como base a abordagem

Tabela 5.2: Requisitos de teste para cada profundidade de cada critério de fluxo de controle, para o método `area`.

d	**todos-nós-integrados-N_d**	**todas-arestas-integradas-N_d**
1	R_n ={(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22}	R_e ={((1).1.0,(1).1.13); ((1).1.13,(1).1.22); ((1).2.0,(1).2.13); ((1).2.13,(1).2.22)}
2	R_n ={(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.6; (2).1.7; (2).1.14; (2).1.22; (2).1.26; (2).2.0; (2).2.6; (2).2.7; (2).2.11; (2).2.18; (2).2.26}	R_e ={((1).1.0,(1).1.13); ((1).1.13,(2).1.0); ((1).2.0,(1).2.13); ((1).2.13,(2).2.0); ((2).1.0,(2).1.6); ((2).1.0,(2).1.7); ((2).1.6,(1).1.22); ((2).1.7,(2).1.14); ((2).1.7,(2).1.22); ((2).1.14,(2).1.22); ((2).1.22,(2).1.26); ((2).1.26,(1).1.22); ((2).2.0,(2).2.6); ((2).2.0,(2).2.7); ((2).2.6,(1).2.22); ((2).2.7,(2).2.11); ((2).2.11,(2).2.18); ((2).2.11,(2).2.26); ((2).2.18,(2).2.26); ((2).2.26,(1).2.22)}
3	R_n ={(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.6; (2).1.7; (2).1.14; (2).1.22; (2).1.26; (2).2.0; (2).2.6; (2).2.7; (2).2.11; (2).2.18; (2).2.26; (3).1.0; (3).1.3; (3).2.0; (3).2.3}	R_e ={((1).1.0,(1).1.13); ((1).1.13,(2).1.0); ((1).2.0,(1).2.13); ((1).2.13,(2).2.0); ((2).1.0,(2).1.6); ((2).1.0,(2).1.7); ((2).1.6,(1).1.22); ((2).1.7,(2).1.14); ((2).1.7,(2).1.22); ((2).1.14,(2).1.22); ((2).1.22,(3).1.0); ((2).1.26,(1).1.22); ((2).2.0,(2).2.6); ((2).2.0,(2).2.7); ((2).2.6,(1).2.22); ((2).2.7,(3).2.0); ((2).2.11,(2).2.18); ((2).2.11,(2).2.26); ((2).2.18,(2).2.26); ((2).2.26,(1).2.22); ((3).1.0,(3).1.3); ((3).1.3,(2).1.26); ((3).2.0,(3).2.3); ((3).2.3,(2).2.11)}
4	R_n ={(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.6; (2).1.7; (2).1.14; (2).1.22; (2).1.26; (2).2.0; (2).2.6; (2).2.7; (2).2.11; (2).2.18; (2).2.26; (3).1.0; (3).1.3; (3).2.0; (3).2.3; (4).1.0; (4).1.5; (4).1.16; (4).1.23; (4).2.0; (4).2.5; (4).2.16; (4).2.23}	R_e ={((1).1.0,(1).1.13); ((1).1.13,(2).1.0); ((1).2.0,(1).2.13); ((1).2.13,(2).2.0); ((2).1.0,(2).1.6); ((2).1.0,(2).1.7); ((2).1.6,(1).1.22); ((2).1.7,(2).1.14); ((2).1.7,(2).1.22); ((2).1.14,(2).1.22); ((2).1.22,(3).1.0); ((2).1.26,(1).1.22); ((2).2.0,(2).2.6); ((2).2.0,(2).2.7); ((2).2.6,(1).2.22); ((2).2.7,(3).2.0); ((2).2.11,(2).2.18); ((2).2.11,(2).2.26); ((2).2.18,(2).2.26); ((2).2.26,(1).2.22); ((3).1.0,(4).1.0); ((3).1.3,(2).1.26); ((3).2.0,(4).2.0); ((3).2.3,(2).2.11); ((4).1.0,(4).1.16); ((4).1.5,(4).1.16); ((4).1.16,(4).1.5); ((4).1.16,(4).1.23); ((4).1.23,(3).1.3); ((4).2.0,(4).2.16); ((4).2.5,(4).2.16); ((4).2.16,(4).2.5); ((4).2.16,(4).2.23); ((4).2.23,(3).2.3)}

de teste de interfaces proposta Linnenkugel e Mullerburg para programas procedurais (Linnenkugel e Mullerburg, 1990).

Portanto, durante o teste de integração, deve-se testar as interfaces entre as unidades que se relacionam em um programa, ou seja, testar as variáveis que influenciam diretamente a comunicação entre as unidades. Essas variáveis são denominadas variáveis de comunicação (Linnenkugel e Mullerburg, 1990). Note que o adendo é considerado como um método compilado, de tal forma que as variáveis de comunicação são aquelas que são passadas como parâmetro para o método em *bytecode* referente ao adendo, ou variáveis que podem ser alteradas dentro desse método e impactar a unidade afetada no retorno. No caso dos adendos, essas variáveis se referem a dados que são capturados no contexto dos pontos de junção. No método em *bytecode* essas variáveis podem ser de qualquer tipo da linguagem Java, isto é, de tipo primitivo ou de referência. Em um programa OO ou OA pode-se identificar as seguintes variáveis de comunicação:

- Parâmetros formais (FP – *Formal Parameters*);
- Parâmetros reais (AP – *Actual Parameters*);
- Atributos estáticos das classes do programa (SF – *Static Field*);

Os atributos de instância (IF – *Instance Field*) declarados em um determinado módulo também podem ser considerados variáveis de comunicação nos casos em que a unidade chamada/entrecortada é um método de instância. Entretanto, eles são tratados como parâmetros reais (AP) e parâmetros formais (FP). Um atributo de instância é um atributo cujo valor é específico ao objeto e não à classe. Em Java, para chamar um método de instância de uma classe é necessário criar um objeto dessa classe e, a partir desse objeto, chamar o método. Todo método de instância tem um parâmetro implícito chamado THIS que é usado para passar a referência do objeto que o está chamando. Deste modo, nos casos em que a unidade chamada é um método de instância, a referência ao objeto, a partir da qual os atributos de instância podem ser acessados, é passada como parâmetro.

O teste estrutural de integração contextual deve considerar somente os caminhos no $\mathcal{C}o\mathcal{DU}$ (ou relações definição-uso) que influenciam diretamente a comunicação entre as unidades afetadas, ou seja:

- para as variáveis de comunicação x que são usadas como entrada, consideram-se os caminhos compostos dos sub-caminhos a partir da última definição de x precedente à chamada até a chamada na unidade chamadora e dos sub-caminhos a partir da entrada na unidade chamada até o uso de x na unidade chamada.

- para as variáveis de comunicação x que são usadas como saída, consideram-se os caminhos compostos dos sub-caminhos a partir da última definição de x na unidade chamada até a saída da unidade chamada e dos sub-caminhos a partir do retorno da chamada na unidade chamadora até o uso de x na unidade chamadora.

Um programa OO e OA consiste de unidades $u_{b.j}$. Para cada unidade $u_{b.j}$ são definidos os seguintes conjuntos:

- SF-IN($u_{b.j}$) é o conjunto de atributos estáticos usados em U; e

- SF-OUT($u_{b.j}$) é o conjunto de atributos estáticos definidos em U.

- FP-IN*($u_{b.j}$) é o conjunto de parâmetros formais de $u_{b.j}$ usados como entrada;

- FP-OUT*($u_{b.j}$) é o conjunto de parâmetros formais de $u_{b.j}$ usados como saída;

- FP-IN($u_{b.j}$) =FP-IN*($u_{1.x}$)$\cup$FP-IN*($u_{2.y}$) $\cup \cdots \cup$FP-IN*($u_{b.j}$) é a união dos conjuntos de parâmetros formais usados como entrada nas unidades pertencentes ao fluxo de execução a partir da primeira interação de u_0 até $u_{b.j}$;

- FP-OUT($u_{b.j}$) =FP-OUT*($u_{1.x}$)$\cup$FP-OUT*($u_{2.y}$) $\cup \cdots \cup$FP-OUT*($u_{b.j}$) é a união dos conjuntos de parâmetros formais usados como entrada nas unidades pertencentes ao fluxo de execução a partir da primeira interação de u_0 até $u_{b.j}$;

Seja $u_{a.i}$ e $u_{b.j}$ duas unidades pertencentes a um programa OO e OA. A interação de $u_{b.j}$ com $u_{a.i}$ é representada por $u_{b.j-a.i}$. Para essa interação são definidos os seguintes conjuntos:

- AP-IN($u_{b.j-0}$) =AP-IN($u_{1.x-0}$)$\cup$AP-IN($u_{2.y-1.x}$)$\cup\cdots\cup$AP-IN($u_{b.j-a.i}$) é a união dos conjuntos de parâmetros reais usados como entrada nas unidades que se relacionam diretamente no fluxo de execução de u_0 até $u_{b.j}$;

- AP-OUT($u_{b.j-0}$) =AP-OUT($u_{1.x-0}$)$\cup$AP-OUT($u_{2.y-1.x}$)$\cup\cdots\cup$AP-OUT($u_{b.j-a.i}$) é a união dos conjuntos de parâmetros reais usados como saída nas unidades que se relacionam diretamente no fluxo de execução de u_0 até $u_{b.j}$;

Para descrever as relações entre parâmetros reais e seus correspondentes parâmetros formais e entre os atributos estáticos usados pelas unidades são definidos dois mapeamentos $IM_{b.j}$ e $OM_{b.j}$. Antes da definição dos mapeamentos é importante ressaltar que ao fazer o mapeamento dos parâmetros e dos atributos estáticos que são do tipo de referência, ocorre tanto o mapeamento deles com seus correspondentes quanto o mapeamento dos seus atributos de instância (caso sejam uma referência a um objeto), ou o mapeamento das suas variáveis agregadas (caso sejam uma referência a um array). Outra observação diz respeito aos atributos estáticos. Eles possuem a mesma identificação tanto na unidade chamadora quanto na unidade chamada. O mapeamento $IM_{b.j}$ relaciona os conjuntos de parâmetros reais usados como entrada nas unidades que se relacionam diretamente no fluxo de execução de u_0 até $u_{b.j}$ com os conjuntos de parâmetros formais usados como entrada nas unidades pertencentes ao fluxo de execução a partir da primeira interação de u_0 até $u_{b.j}$, e cada atributo estático usado como entrada com ele mesmo.

- $IM_{b.j}$: AP-IN($u_{b.j-0}$) $\cup$ SF-IN($u_{b.j}$) $\longrightarrow$ FP-IN($u_{b.j}$) $\cup$ SF-IN($u_{b.j}$), com
  - AP-IN($u_{b.j-0}$) $\longrightarrow$ FP-IN($u_{b.j}$).

O mapeamento $OM_{b.j}$ relaciona os conjuntos de parâmetros reais usados como saída nas unidades que se relacionam diretamente no fluxo de execução de u_0 até $u_{b.j}$ com os conjuntos de parâmetros formais usados como entrada nas unidades pertencentes ao fluxo de execução a partir da primeira interação de u_0 até $u_{b.j}$, e cada atributo estático usado como saída com ele mesmo.

- $OM_{b.j}$: AP-OUT($u_{b.j-0}$) $\cup$ SF-OUT($u_{b.j}$) $\longrightarrow$ FP-OUT($u_{b.j}$) $\cup$ SF-OUT($u_{b.j}$), com
  - AP-OUT($u_{b.j-0}$) $\longrightarrow$ FP-OUT($u_{b.j}$).

Com base nessas definições e no grafo $\mathcal{CODU}$ das unidades, alguns conjuntos são definidos. Para isso, considere: *def(x)* é o conjunto de variáveis definidas no nó x; *c-uso(x)* é o conjunto de variáveis para as quais existem *c-uso* em x; *p-uso(x, y)* é o conjunto de variáveis para as quais existem *p-usos* na aresta *(x,y)*. Assim, para cada interação $u_{b.j-a.i}$ e uma variável z, foram definidos os seguintes conjuntos:

- DEF-CALLED($u_{b.j}$,z) = $\{x \in u_{b.j} | z \in def(x)$ e existe um caminho livre de definição com relação a z a partir do nó x até o nó de saída de $u_{b.j}\}$, sendo que $z \in$ FP-OUT($u_{b.j}$) ou $z \in$ SF-OUT($u_{b.j}$).

- C-USE-CALLED($u_{b.j}$,z) = $\{x \in u_{b.j} | z \in$ c-uso(x) e existe um caminho livre de definição com relação a z a partir do nó de entrada de $u_{b.j}$ até o nó $x\}$, sendo que $z \in$ FP-IN($u_{b.j}$) ou $z \in$ SF-IN($u_{b.j}$).

- P-USE-CALLED($u_{b.j}$,z) = $\{(x, y) \in u_{b.j} | z \in$ p-uso(x, y) e existe um caminho livre de definição com relação a z a partir do nó de entrada de $u_{b.j}$ até a aresta $(x, y)\}$, sendo que $z \in$ FP-IN($u_{b.j}$) ou $z \in$ SF-IN($u_{b.j}$).

Para a interação $u_{b.j-0}$,z foram definidos os seguintes conjuntos

- DEF-CALLER($u_{b.j-0}$,z) = $\{x \in u_0 | z \in def(x)$ e existe um caminho livre de definição com relação a z a partir do nó x até o nó de interação de $u_{b.j}\}$, sendo que $z \in$ AP-IN($u_{b.j-0}$) ou $z \in$ SF-IN($u_{b.j}$).

- C-USE-CALLER($u_{b.j-0}$,z) = $\{x \in u_0 | z \in$ c-uso(x) e existe um caminho livre de definição com relação a z a partir dos nós de retorno de $u_{b.j}$ até o nó $x\}$, sendo que $z \in$ AP-OUT($u_{b.j-0}$) ou $z \in$ SF-OUT($u_{b.j}$).

- P-USE-CALLER($u_{b.j-0}$,z) = $\{(x, y) \in u_{b.j-0} | z \in$ p-uso(x, y) e existe um caminho livre de definição com relação a z a partir dos nós de retorno de $u_{b.j}$ até a aresta $(x, y)\}$, sendo que $z \in$ AP-OUT($u_{b.j-0}$) ou $z \in$ SF-OUT($u_{b.j}$).

A partir das definições anteriores, o critério todos-usos-integrados-N_d utilizado para derivar requisitos de teste estrutural de integração contextual baseado nas interfaces entre as unidades é definido.

- **todos-usos-integrados-N_d**: Π satisfaz o critério todos-usos-integrados-N_d se:

  1. para cada $z \in$ (AP-IN($u_{b.j-0}$) $\cup$ SF-IN($u_{b.j}$)), Π inclui um caminho livre de definição com relação a z a partir de cada nó $x \in$ DEF-CALLER($u_{b.j-0}, z$) até cada nó $y \in$ C-USER-CALLED($u_{b.j}, IM_{b.j-0}(z)$) e até cada aresta $(x, y) \in$

P-USER-CALLED($u_{b.j}, IM_{b.j-0}(z)$). Em outras palavras, esse critério requer a execução de um caminho livre de definição com relação a cada variável de comunicação a partir de cada definição relevante na unidade chamadora até todo uso computacional e todo uso predicativo nas unidades chamadas.

2. para cada $z \in$ (AP-OUT($u_{b.j-0}$) $\cup$ SF-OUT($u_{b.j}$)), Π inclui um caminho livre de definição com relação a z a partir de cada nó $x \in$ DEF-CALLED($u_{b.j-0}$, $OM_{b.j-0}(z)$) até cada nó $y \in$ C-USER-CALLER($u_{b.j}, z$) e até cada aresta $(x, y) \in$ P-USER-CALLER($u_{b.j}, z$). Em outras palavras, esse critério requer a execução de um caminho livre de definição com relação a cada variável de comunicação a partir de cada definição relevante nas unidades chamadas até todo uso computacional e todo uso predicativo na unidade chamadora.

3. para cada $z \in$ (AP-OUT($u_{b.j-0}$) $\cup$ SF-OUT($u_{b.j}$)) $\cap$ (AP-IN($u_{a.i-0}$) $\cup$ $\cup$ SF-IN($u_{a.i}$)), Π inclui um caminho livre de definição com relação a z a partir de cada nó $x \in$ DEF-CALLER($u_{b.j-0}, OM_{b.j}(z)$) até cada nó $y \in$ $\in$ C-USER-CALLED($u_{a.i}, IM_{a.i}(z)$) e até cada aresta $(x, y) \in$ $\in$ P-USER-CALLED($u_{a.i}, IM_{a.i}(z)$). Em outras palavras, esse critério requer a execução de um caminho livre de definição com relação a cada variável de comunicação a partir de cada definição relevante em uma determinada unidade chamada até todo uso computacional e todo uso predicativo em todas as outras unidades chamadas, sendo que o fluxo de execução obrigatoriamente passe pela unidade-base.

Exemplo de Aplicação

Na Tabela 5.3 são apresentados os requisitos de teste para o critério todos-usos-integrados-N_d para o método `area` da classe `Circle`, apresentado no Capítulo 5, em todos os níveis de profundidade até a profundidade máxima, ou seja, $1 \leq d \leq 4$. Os requisitos são gerados a partir dos $\mathcal{C}o\mathcal{DU}$s mostrados nas Figuras 5.4, 5.5, 5.6 e 5.7.

Tabela 5.3: Requisitos de teste para cada profundidade do critério de fluxo de dados, para o método `area`.

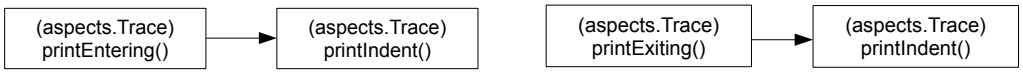
d	**todos-usos-integrados-N_d**
1	R_u ={}
2	R_u ={(S@aspects.Trace.callDepth,(2).1.14,(2).2.18)}
3	R_u ={(S@aspects.Trace.callDepth,(2).1.14,(2).2.18)}
4	R_u ={(S@aspects.Trace.callDepth,(2).1.14,(2).2.18); (S@aspects.Trace.callDepth,(2).1.14,((4).2.16,(4).2.5)); (S@aspects.Trace.callDepth,(2).1.14,((4).2.16,(4).2.23))}

5.6 Estratégia Básica de Teste

Conforme discutido nos capítulos anteriores, a atividade de teste é dividida em três fases: (1) Teste de Unidade, (2) Teste de Integração e (3) Teste de Sistema (Bertolino, 2007). Seguindo a divisão apresentada, a aplicação dos critérios de teste estrutural de integração contextual claramente se encaixam na fase de teste de integração, sendo aplicados após o teste de unidade do sistema. Assim, a estratégia sugerida nesse contexto seria: (1) realizar o teste de cada unidade do sistema em isolamento (utilizando, por exemplo os critérios propostos anteriormente por Vincenzi (2004) e Lemos (2005)); (2) testar a interação entre todas as unidades pertencentes a um fluxo de execução de uma unidade utilizando os critérios propostos nesta dissertação; e (3) enfatizar os interesses transversais testando cada adendo em cada um dos ponto de junção afetados, utilizando os critérios propostos por Lemos e Masiero (2011). Vale ressaltar que no passo (2) é recomendado testar, de forma incremental, a integração de todas as unidades que disparam um fluxo de execução com sua chamada/entrecorte, e em todos os níveis de profundidade, para facilitar a detecção de possíveis erros, bem como a criação de casos de testes específicos para a cobertura de um determinado requisito.

Para exemplificar o passo (2), considere as unidades mostradas no fluxo de execução da Figura 5.2, e repetida na Figura 5.19(d), como sendo as únicas unidades de todo o programa `Shape`. Analisando as interações entre as unidades é possível observar que apenas a unidade `aspects.Trace.printIdent` não chama ou é entrecortada por nenhuma outra unidade. Ou seja, todas as unidades, com exceção da unidade `aspects.Trace.printIdent`, devem ser testadas por meio da abordagem proposta. Além disso, as unidades devem ser testadas de forma incremental com relação à sua profundidade.

Portanto, para testar as unidades `aspects.Trace.printEntering` e `aspects.Trace.printExiting`, que possuem profundidade 1, devemos considerar as integrações entre as unidades mostradas nas Figuras 5.13(a) e 5.13(b), respectivamente.



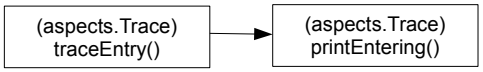
(a) Fluxo de execução da unidade `printEntering` no nível 1 de profundidade.

(b) Fluxo de execução da unidade `printExiting` no nível 1 de profundidade.

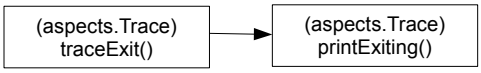
Figura 5.13: Interações dos métodos `printEntering` e `printExiting` no nível 1 de profundidade.

Para testar as unidades `aspects.Trace.traceEntry` e `aspects.Trace.traceExit`, que possuem profundidade 2, devemos considerar primeiramente as integrações no nível 1 de profundidade, como mostrado nas Figuras 5.14(a) e 5.14(b) e, após o teste no nível

1 de profundidade, o nível 2 de profundidade de interação deve ser considerado, como mostrado nas Figuras 5.15(a) e 5.15(b).

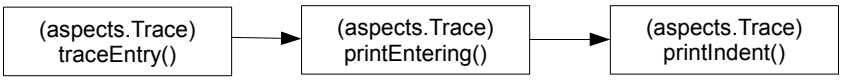


(a) Fluxo de execução da unidade `traceEntry` no nível 1 de profundidade.

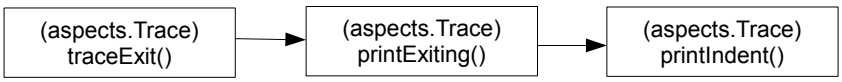


(b) Fluxo de execução da unidade `traceExit` no nível 1 de profundidade.

Figura 5.14: Interações dos métodos `traceEntry` e `traceExit` no nível 1 de profundidade.



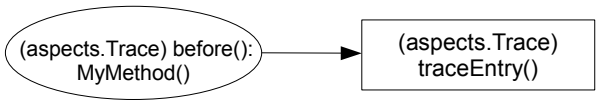
(a) Fluxo de execução da unidade `traceEntry` no nível 2 de profundidade.



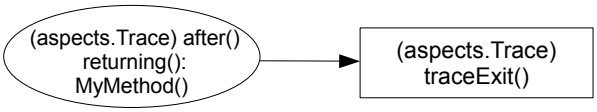
(b) Fluxo de execução da unidade `traceExit` no nível 2 de profundidade.

Figura 5.15: Interações dos métodos `traceEntry` e `traceExit` no nível 2 de profundidade.

O teste dos adendos `aspects.Trace.before():MyMethod` e `aspects.Trace.after() returning():MyMethod`, que possuem profundidade 3, devem considerar primeiramente a integração das unidades na profundidade 1 (Figuras 5.16(a) e 5.16(b)), depois na profundidade 2 ((Figuras 5.17(a) e 5.17(b))) e, por fim, na profundidade 3 (Figuras 5.18(a) e 5.18(b)).

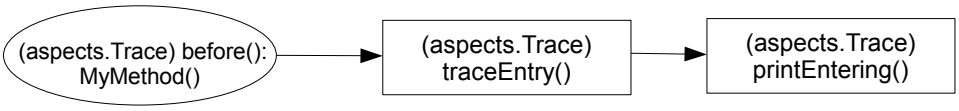


(a) Fluxo de execução do adendo do tipo `before` no nível 1 de profundidade.

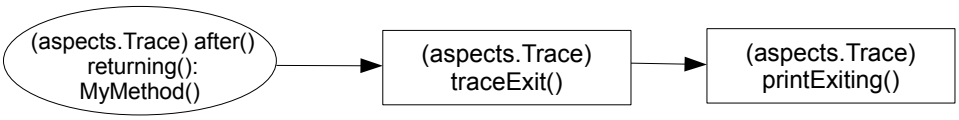


(b) Fluxo de execução do adendo do tipo `after` no nível 1 de profundidade.

Figura 5.16: Interações dos adendos `before` e `after` no nível 1 de profundidade.

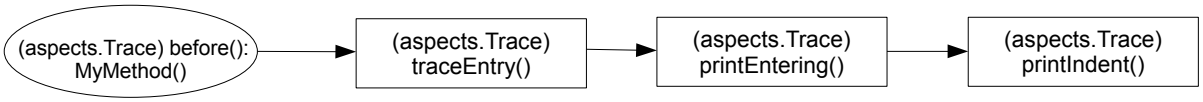


(a) Fluxo de execução do adendo do tipo `before` no nível 2 de profundidade.

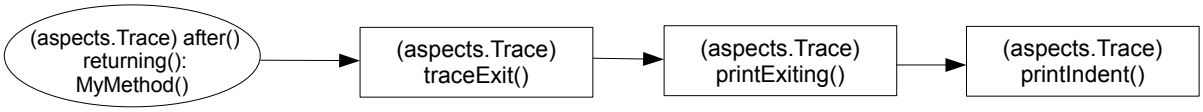


(b) Fluxo de execução do adendo do tipo `after` no nível 2 de profundidade.

Figura 5.17: Interações dos adendos `before` e `after` no nível 2 de profundidade.



(a) Fluxo de execução do adendo do tipo `before` no nível 3 de profundidade.



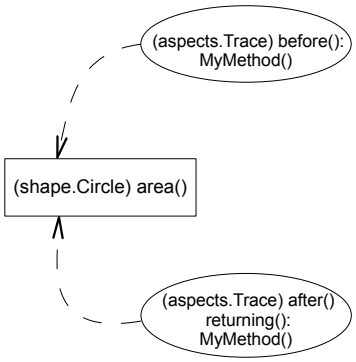
(b) Fluxo de execução do adendo do tipo `after` no nível 3 de profundidade.

Figura 5.18: Interações dos adendos `before` e `after` no nível 3 de profundidade.

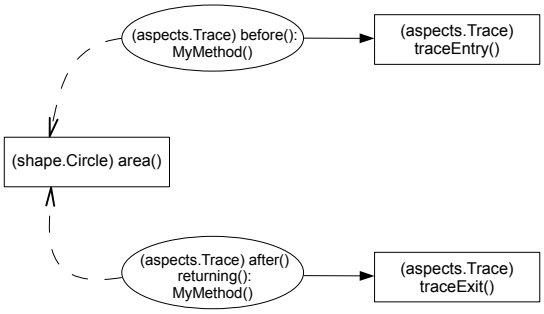
Finalmente, o teste da unidade `shape.Circle.area` deve considerar, na profundidade 1 de interação, o fluxo de execução das unidades mostradas na Figura 5.19(a). Posteriormente, deve-se considerar as unidades até a profundidade 2, como mostrado na Figura 5.19(b). Depois disso, o método `shape.Circle.area` deve ser testado na profundidade 3 com as unidades mostradas na Figura 5.19(c). Por fim, o método deve ser testado em sua profundidade máxima de interação (profundidade 4), como mostrado na Figura 5.19(d).

Portanto, o teste das sete unidades envolvidas no fluxo de execução do método `shape.Circle.area` envolve 16 configurações diferentes se considerarmos o teste em todas as profundidades de interação.

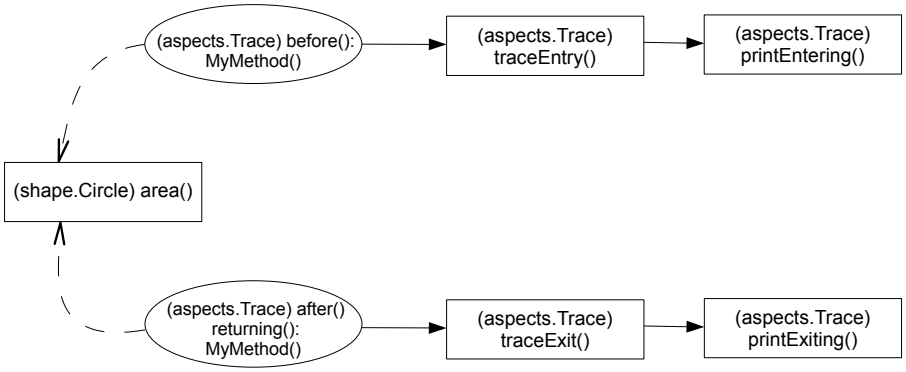
Além disso, como existe uma hierarquia de inclusão entre os critérios de teste estrutural de integração contextual, desconsiderando os caminhos não executáveis, dentro do passo (2) da estratégia apresentada ainda pode ser imposta uma ordem de aplicação dos critérios começando pelo critério todos-nós-integrados-N_d, depois o critério todas-arestas-integradas-N_d e, por fim, o critério todos-usos-integrados-N_d. A ordem de aplicação adotada sugere iniciar o teste com a aplicação do critério mais fraco e incrementar os casos de teste em direção à adequação dos critérios mais fortes.



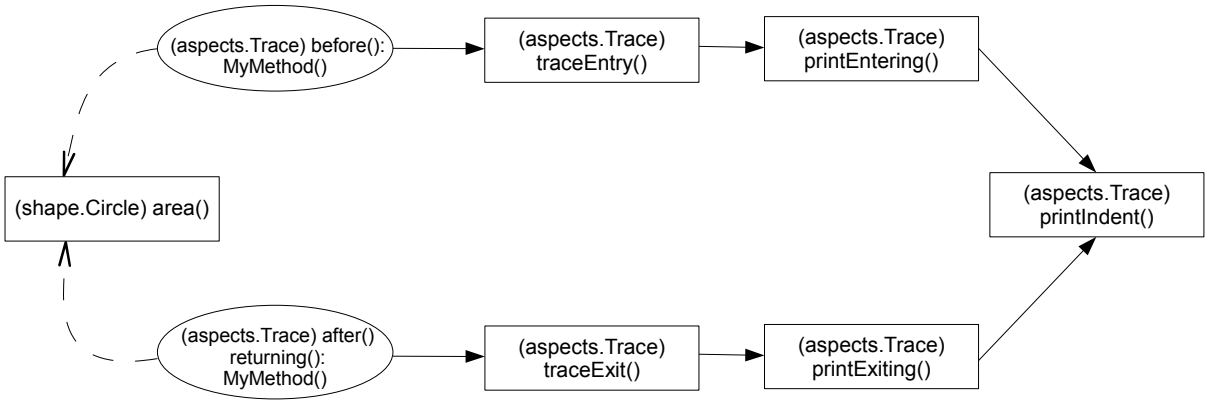
(a) Fluxo de execução do método `area` no nível 1 de profundidade.



(b) Fluxo de execução do método `area` no nível 2 de profundidade.



(c) Fluxo de execução do método `area` no nível 3 de profundidade.



(d) Fluxo de execução do método `area` no nível 4 de profundidade.

Figura 5.19: Interações do método `area` em todos os níveis de profundidade.

5.7 Considerações Finais

Neste capítulo foi apresentada uma abordagem de teste estrutural de integração contextual para programas OO e OA. A abordagem inclui um modelo para a representação de todo o fluxo de execução no contexto de um método sob teste. Com base no modelo, foi definida uma família com três critérios de teste – dois baseados no fluxo de controle e um baseado

no fluxo de dados – e foi apresentado o modelo de fluxo de dados utilizado para identificar o que caracteriza uma definição ou o uso de uma variável em uma sentença ou instrução.

Com a abordagem de teste definida, é necessário implementá-la em uma ferramenta para viabilizar o seu uso. No Capítulo 6 é mostrado o processo de criação da extensão da ferramenta JaBUTi/AJ que possibilita o teste estrutural de integração contextual. Para ilustrar a aplicabilidade da abordagem é apresentado um exemplo de aplicação no teste de um programa OA.

Capítulo

6

Automação do Teste Estrutural de Integração Contextual

6.1 Considerações Iniciais

As técnicas e critérios de teste fornecem ao desenvolvedor uma abordagem sistemática e teoricamente fundamentada, além de constituírem um mecanismo que pode auxiliar a avaliar a qualidade e a adequação da atividade de teste. Com isso, é fundamental o desenvolvimento de ferramentas de teste para o suporte à atividade de teste propriamente dita, uma vez que essa atividade é muito propensa a erros, além de menos produtiva, se aplicada manualmente. Assim, a disponibilidade de ferramentas de teste propicia maior qualidade e produtividade para as atividades de teste (Maldonado et al., 2004).

Deste modo, é importante implementar a abordagem de teste estrutural de integração contextual proposta em uma ferramenta de teste. Para isso, a ferramenta *JaBUTi/AJ*, que é uma ferramenta para apoiar o teste estrutural de programas OO e OA escritos em Java e AspectJ, foi estendida para possibilitar a aplicação automatizada da abordagem proposta neste trabalho.

Na Seção 6.2 é descrita a extensão da ferramenta *JaBUTi/AJ* para o teste estrutural de integração contextual. Na Seção 6.3 é apresentado o modo de utilização da *JaBUTi/AJ* por meio da utilização de um exemplo de uso. Por fim, na Seção 6.4 são apresentadas as considerações finais do capítulo.

6.2 Extensão da Ferramenta JaBUTi/AJ

A ferramenta *JaBUTi/AJ* foi estendida para possibilitar a utilização da abordagem de teste estrutural de integração contextual de programas OO e OA proposta neste trabalho. Para isso, o processo de implementação da extensão da ferramenta foi dividido em quatro etapas: a primeira etapa trata da identificação das unidades que interagem com outras unidades em um programa; a segunda etapa envolve a construção do grafo $\mathcal{C}o\mathcal{DU}$ da unidade sob teste; a terceira etapa tem como objetivo implementar os critérios de teste estrutural de integração contextual, ou seja, os critérios todos-nós-integrados-N_d, todas-arestas-integradas-N_d e todos-usos-integrados-N_d; por fim, a quarta etapa mostra o desenvolvimento do ambiente de teste de integração contextual. As quatro fases de desenvolvimento estão detalhadas a seguir.

6.2.1 Identificação das Unidades

A identificação das unidades é feita pela busca das seguintes instruções no bytecode Java do programa: `invokevirtual`, `invokespecial`, `invokestatic` e `invokeinterface`. Tais instruções identificam interações entre unidades e, por meio delas é possível saber qual unidade está sendo chamada e a qual módulo essa unidade pertence. Além disso, também é possível obter, por meio da análise da cadeia de chamadas do bytecode, o nível de profundidade de interação de cada unidade analisada.

O nome dado para as unidades é formado pela assinatura da unidade no nível de bytecode precedida pelo nome do módulo (incluindo o pacote) em que a unidade foi implementada entre parêntesis. Na Figura 6.1 é mostrado um exemplo de nomeação da unidade `area` do programa apresentado na Figura 5.1.

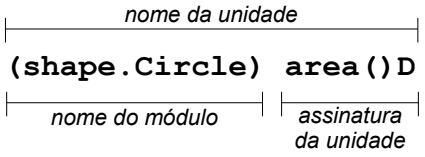


Figura 6.1: Exemplo de nomeação de uma unidade

6.2.2 Construção do Grafo $\mathcal{C}o\mathcal{DU}$

Para construir o grafo $\mathcal{C}o\mathcal{DU}$, a parte da *JaBUTi/AJ* responsável pela construção do grafo $\mathcal{AODU}$ foi estendida. Durante a construção dos grafos $\mathcal{AODU}$ das unidades integradas, os nós de interação são identificados e inseridos em uma estrutura de dados (fila) e é feito o cálculo da profundidade máxima de interação de cada unidade identificada no passo anterior. Dessa forma, o processo de integração dos grafos pode acontecer posteriormente.

Na construção do grafo $\mathcal{CODU}$, o grafo $\mathcal{AODU}$ da unidade-base é gerado para que, posteriormente, os grafos $\mathcal{AODU}$s das unidades que se relacionam diretamente com a unidade-base sejam gerados. Vale ressaltar que os grafos $\mathcal{AODU}$s passam por tratamentos especiais e otimizações antes de serem gerados.

Antes de integrar todos os grafos gerados, as arestas que conectam os nós de interação e os nós seguintes na unidade-base são removidos identificando os lugares aonde os grafos $\mathcal{AODU}$s serão integrados. Após isso, são criadas arestas de integração do nó de interação até o nó de entrada do grafo $\mathcal{AODU}$ da unidade integrada, bem como a partir de cada nó de saída do grafo $\mathcal{AODU}$ integrado até os nós seguintes ao nó de interação da unidade-base. Os nós de interação são removidos da fila que armazena os nós de interação e o processo recomeça com uma outra unidade (uma unidade integrada) como unidade-base. Esse processo é repetido até que não existam mais nós de interação na fila de nós de interação ou até que o nível de profundidade de interação (máximo ou definido pelo testador) tenha sido alcançado.

Com esses passos, a representação do grafo $\mathcal{CODU}$ está pronta para ser usada na geração dos requisitos de teste.

6.2.3 Implementação dos Critérios de Teste

A implementação dos três critérios de teste estrutural de integração contextual (definidos na Seção 5.5) foi feita baseando-se nos critérios anteriormente implementados por Lemos et al. (2007) e Lemos (2009). A implementação dos critérios de teste baseados no fluxo de controle utilizou de forma direta os critérios todos-nós e todas-arestas, necessitando apenas filtrar os requisitos de teste que incluem somente os nós e arestas de grafos integrados.

O critério de teste baseado no fluxo de dados demandou maior esforço se comparado à implementação dos critérios de fluxo de controle. O detalhe mais importante da implementação desse critério está relacionado ao mapeamento das variáveis de comunicação. Utilizando o critério par-a-par implementado por Franchin (2007), no qual os requisitos de teste para cada par de unidades, é feito um mapeamento das variáveis de comunicação e posterior identificação de definições e usos de variáveis em todas as interações de unidades considerando todos os níveis de profundidade. Dessa forma, é possível detectar o caminho feito pelas variáveis de comunicação pelas unidades do grafo.

6.2.4 Implementação do Ambiente de Teste

Para apoiar a abordagem de teste estrutural de integração contextual, um ambiente específico foi adicionado à ferramenta *JaBUTi/AJ*. Esse ambiente utiliza os mesmos módulos selecionados para serem testados e instrumentados no ambiente do teste de unidade e,

a partir desses módulos, as unidades relevantes ao teste de integração contextual são identificadas. Além disso, todas as atividades executadas no ambiente específico para a abordagem de teste estrutural de integração não afetam o teste em outros ambientes. Por exemplo, se um caso de teste for executado no ambiente de teste de unidade, tal execução não afeta as informações do ambiente de teste estrutural de integração contextual. O testador também pode salvar o projeto de teste e a separação ainda será mantida.

O ambiente de teste estrutural de integração contextual apoia as seguintes atividades: verificação dos requisitos de teste gerados para cada critério implementado, importação de casos de teste JUnit, verificação da cobertura obtida pelos casos de teste importados, e visualização dos grafos para cada unidade selecionada.

6.2.5 Esforço de Implementação

Para a extensão da ferramenta *JaBUTi/AJ* responsável pela realização do teste estrutural de integração contextual, as seguintes classes foram implementadas no pacote `criteria` (responsável pela implementação dos critérios de teste disponíveis na ferramenta): `AllCoNodes`, `AllCoEdges` e `AllCoUses`. As classes modificadas foram as seguintes: `AbstractCriterion`, `AllUses` e `Criterion` do pacote `criteria`; `CFG`, `CFGNode`, `DominatorTree` e `GraphNode` do pacote `graph` (responsável pela geração dos grafos na interface); `DialogIntegrationSelection`, `JabutiGUI`, `TableSorterPanel` e `WeightColor` do pacote `gui` (responsável pela interface da ferramenta); `ClassIntegrationData`, `ClassMethod`, `DefUseIntegrationManagement`, `IntegrationParameter`, `TestCase`, `TestingType`, `TestSet` e `UnitMethodData` do pacote `project` (responsável pelo gerenciamento dos projetos de teste); `ToolConstants` do pacote `util` (responsável pela implementação de ações auxiliares da ferramenta); e `InstructionNode` do pacote `verifier` (responsável pelo armazenamento e geração de informações a partir do bytecode). Para resumir o esforço empreendido na extensão da ferramenta, na Tabela 6.1 são listadas as classes criadas / modificadas com uma breve descrição da funcionalidade relacionada, se a classe foi criada ou modificada (C/M) e o número de linhas de código criadas ou modificadas em cada classe (#LDC). Ao todo 22 classes foram modificadas e 3 classes foram criadas. Nenhum pacote foi criado, apenas foram criadas/modificadas classes dentro dos pacotes existentes.

6.3 Exemplo de Uso da Ferramenta

Esta seção descreve brevemente a interface e algumas das funcionalidades da ferramenta *JaBUTi/AJ*. Em seguida, para ilustrar o processo de teste utilizando a ferramenta e evidenciar a aplicabilidade da abordagem proposta, é mostrado um exemplos de uso na Seção 6.3.2.

Tabela 6.1: Arquivos criados/modificados na extensão da ferramenta *JaBUTi/AJ*.

Pacote	Classe/Interface	Descrição	C/M	#LDC
criteria	AbstractCriterion	Classe abstrata que implementa a interface Criterion.	M	139
	AllCoNodes	Classe que implementa o critério todos-nos-integrados-N_d.	C	116
	AllCoEdges	Classe que implementa o critério todas-arestas-integradas-N_d.	C	154
	AllCoUses	Classe que implementa o critério todos-usos-integrados-N_d.	C	220
	AllUses	Classe que implementa o critério todos-usos.	M	7
	Criterion	Interface que define as operações dos critérios implementados.	M	13
graph	CFG	Classe responsável pela montagem do grafo $\mathcal{CODU}$.	M	604
	CFGNode	Classe que define o objeto nó do grafo $\mathcal{CODU}$.	M	67
	DominatorTree	Classe que constrói a árvore de dominadores do $\mathcal{CODU}$.	M	7
	GraphNode	Classe abstrata que representa o nó de um programa.	M	11
gui	DialogIntegrationSelection	Diálogo que mostra as unidades que podem ser testadas, bem como o nível de profundidade de cada uma delas.	M	359
	JabutiGUI	Ambiente do teste estrutural de integração contextual.	M	375
	TableSorterPanel	Diálogo que permite ao usuário selecionar as unidades a serem testadas.	M	50
	WeightColor	Classe que calcula o peso das cores dadas ao grafo $\mathcal{CODU}$ de acordo com o critério de teste selecionado.	M	3
project	ClassFile	Classe que armazena informações de cobertura de uma classe/aspecto.	M	6
	ClassIntegrationData	Classe de infra-estrutura principal. Armazena informações sobre a integração das unidades.	M	750
	ClassMethod	Classe que calcula dados de teste referente a uma unidade sob teste.	M	307
	DefUseIntegrationManagement	Classe responsável por calcular os requisitos de teste baseados no critério de fluxo de dados.	M	432
	IntegrationParameter	Classe que realiza o mapeamento das variáveis de comunicação.	M	127
	TestCase	Classe que representa um caso de teste.	M	225
	TestingType	Classe que define os tipos de teste disponíveis na ferramenta.	M	1
	TestSet	Classe que representa um conjunto de casos de teste.	M	3
	UnitMethodData	Classe que representa uma unidade no teste estrutural de integração contextual.	M	244
util	ToolConstants	Classe que armazena estatísticas usadas pela ferramenta.	M	1
verifier	InstructionNode	Classe que representa o nó de um grafo de instrução utilizado para derivar o grafo $\mathcal{CODU}$.	M	34
Total				4255

6.3.1 A Interface da Ferramenta JaBUTi/AJ

Como exemplo para demostração das interfaces e funcionalidades da ferramenta *JaBUTi/AJ* foi utilizado o programa OA apresentado no Capítulo 5. A interface principal da ferramenta *JaBUTi/AJ* é mostrada na Figura 6.2.

Para testar uma aplicação na ferramenta *JaBUTi/AJ* é necessário, primeiramente, criar um projeto de teste. Nesta etapa o testador escolhe quais classes ou aspectos deseja instrumentar e testar. Após a seleção, a ferramenta constrói os grafos $\mathcal{AODU}$s e deriva os requisitos de teste de unidade de cada módulo selecionado para ser testado. Para cada requisito derivado a ferramenta calcula e atribui diferentes pesos (indicados por diferentes cores) que indicam o requisito que, se coberto, aumentaria ao máximo a cobertura em relação ao critério considerado.

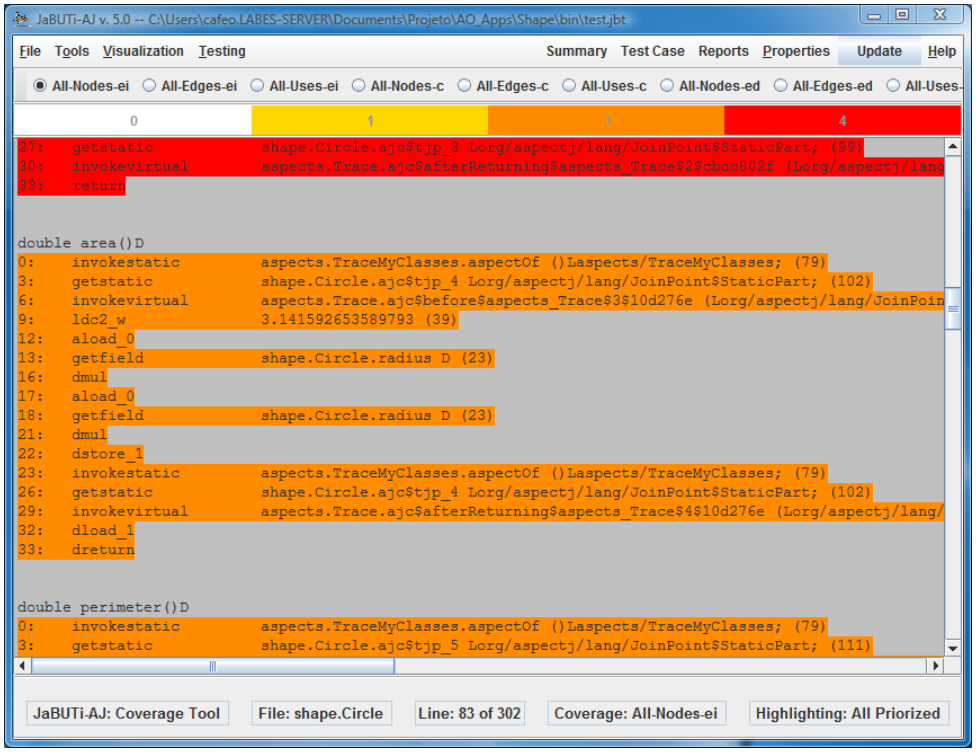


Figura 6.2: Tela principal da ferramenta *JaBUTi/AJ*.

A partir daí, é possível acessar todas as funcionalidades da ferramenta. Inicialmente, ela está no ambiente de teste de unidade, onde podem ser realizados testes em cada unidade, individualmente. Neste ambiente é possível: visualizar o bytecode (como pode ser visto na Figura 6.2) e o código-fonte dos módulos que estão sendo testados, visualizar os grafos $\mathcal{AODU}$ de cada unidade (Figura 6.3(a)), visualizar o conjunto de requisitos de teste derivados de cada critério para cada unidade (Figura 6.3(b)), importar casos de teste (Figura 6.3(c)) para tentar cobrir os requisitos de teste, analisar a cobertura alcançada pelos casos de teste (Figura 6.3(d)) e gerar relatórios.

Além do ambiente para teste de unidade, a ferramenta fornece os seguintes ambientes de teste de integração: par-a-par (intra-módulo e inter-módulo), nível 1 de profundidade, baseado em conjuntos de junção, e contextual. Conforme explicado no Capítulo 5, o teste estrutural de integração contextual integra as unidades que se relacionam com a unidade sob teste até um determinado nível de profundidade (máxima ou escolhida pelo testador). A seleção do ambiente de teste é feita pela opção "Testing" do menu principal.

Da mesma forma que ocorre no ambiente de teste de unidade, no ambiente de teste estrutural de integração contextual é possível: visualizar os grafos $\mathcal{CODU}$ gerados, visualizar o conjunto de requisitos de teste de cada critério de teste estrutural de integração contextual gerados, importar casos de teste para tentar cobrir o conjunto de requisitos derivados pelos critérios, analisar a cobertura alcançada pelos casos de teste e gerar re-

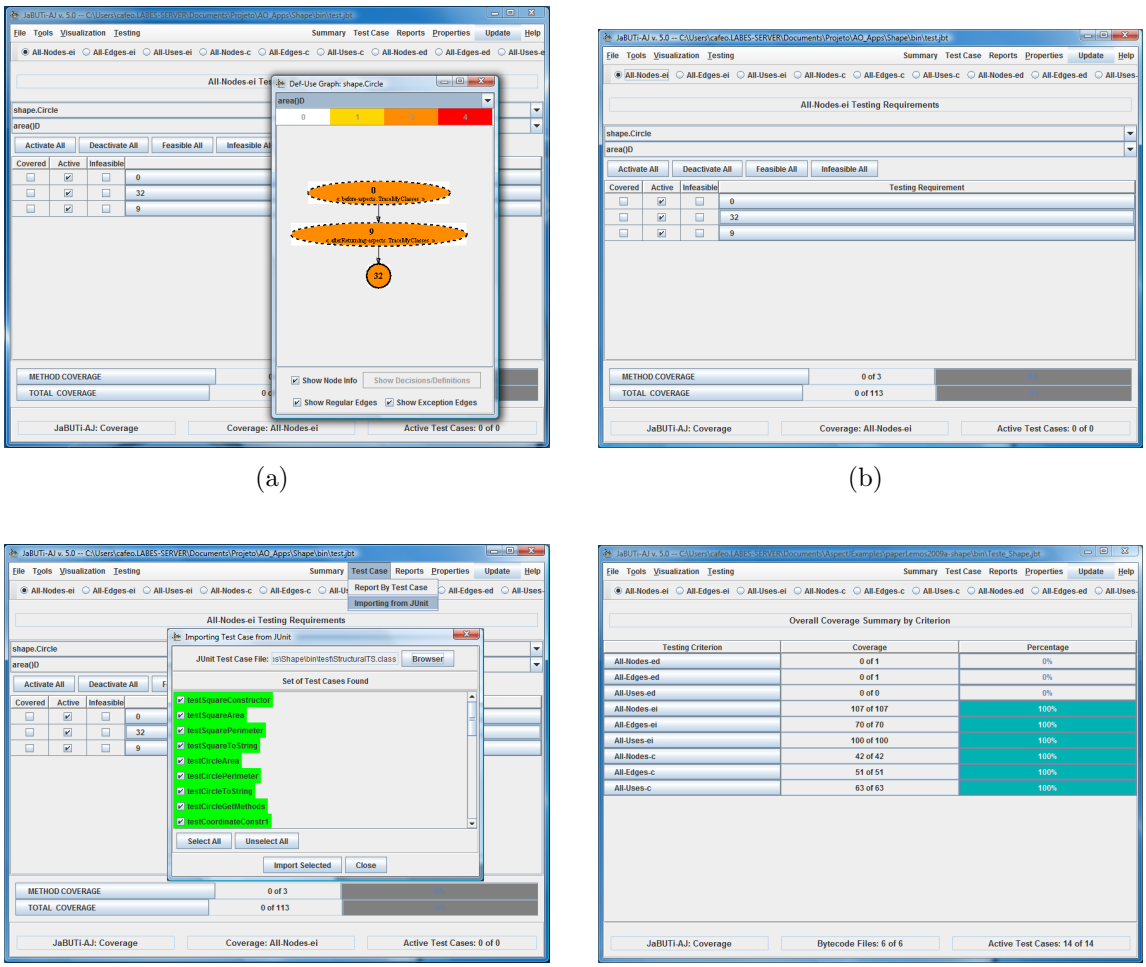


Figura 6.3: Exemplos de interface da ferramenta *JaBUTi/AJ*.

latórios. Além disso, o ambiente de teste de integração contextual permite selecionar as unidades a serem testadas, bem como a profundidade de integração a ser considerada durante os teste (Figura 6.4).

6.3.2 Exemplo de Uso: Programa Shape

Seguindo a estratégia de teste sugerida na Seção 5.6, deve-se primeiramente testar as unidades de cada uma das classes e aspectos presentes no sistema. Para isso, pode-se utilizar a abordagem de teste estrutural de unidade proposta por Lemos et al. (2007). Dessa forma, foi criado um conjunto de casos de teste em JUnit adequado para os critérios de teste estrutural de unidade. O código-fonte dos casos de teste é resumido na Figura 6.5.

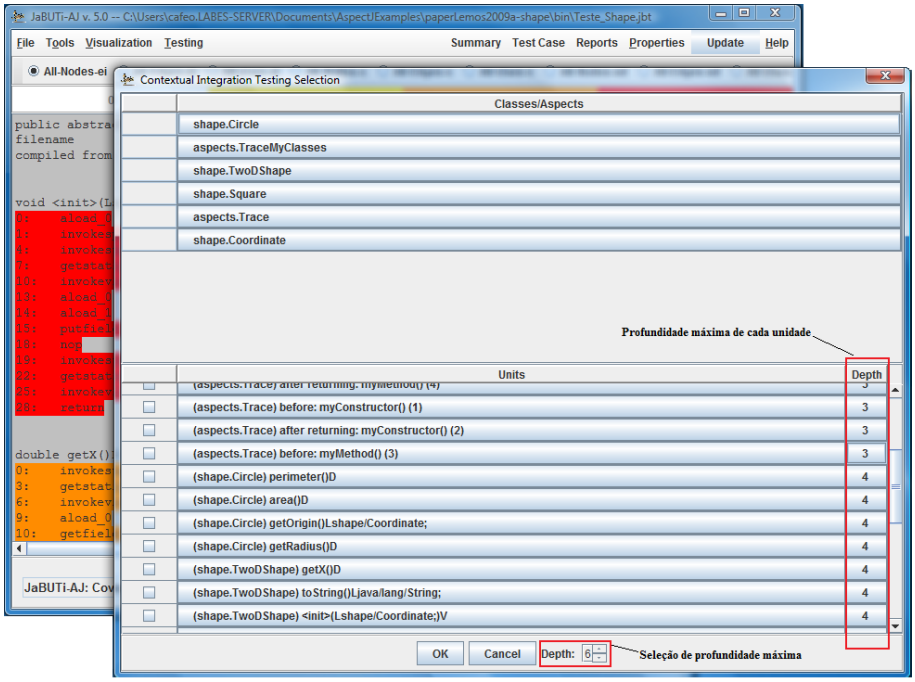


Figura 6.4: Tela de seleção de unidades da ferramenta *JaBUTi/AJ*.

```
20  public class StructuralTS extends TestCase {
21
22    public StructuralTS(String s) {
23      super(s);
24    }
25
26    public void testSquareConstructor() {
27      Square s = new Square(new Coordinate(0,0));
28      assertEquals(1.0, s.area());
29    }
30
31    public void testSquareArea() {
32      Square s = new Square(
33                    new Coordinate(0.0, 0.0), 2);
34      assertEquals(4.0, s.area());
35    }
36
37    public void testSquarePerimeter() {
38      Square s = new Square(2);
39      assertEquals(8.0, s.perimeter());
40    }
41
42    public void testSquareToString() {
43      Square s = new Square();
44      System.out.println(s);
45      assertEquals("Square side = 1.0 @ (0.0, 0.0)",
46                           s.toString());
47    }
48
49    public void testCircleArea() {
50      Circle c = new Circle(10);
51      assertEquals(100 * Math.PI, c.area());
52    }
53
54    public void testCirclePerimeter() {
55      Circle c = new Circle(new Coordinate(0,0), 10);
56      assertEquals(20 * Math.PI, c.perimeter());
57    }
58
59    public void testCircleToString() {
60      Circle c = new Circle();
61      assertEquals("Circle radius = 1.0 @ (0.0, 0.0)",
62                           c.toString());
63    }
64
65    public void testCircleGetMethods() {
66      Circle c = new Circle(new Coordinate(0,0));
67      assertEquals(1.0, c.getRadius());
68      assertEquals(new Coordinate(0,0).toString(),
69                           c.getOrigin().toString());
70    }
71
72    public void testCoordinateConstr1() {
73      Coordinate c = new Coordinate(0,0);
74      Coordinate c1 = new Coordinate(c);
75      assertEquals(c.getX(), c1.getX());
76      assertEquals(c.getY(), c1.getY());
77    }
78
79    public void testCoordinateConstr2() {
80      Coordinate c = new Coordinate();
81      assertEquals(0.0, c.getX());
82      assertEquals(0.0, c.getY());
83    }
84
85    public void testCoordinateDistance() {
86      Coordinate c = new Coordinate();
87      Coordinate c1 = new Coordinate(1, 1);
88      assertEquals(1.0, c.distanceTo(c1), .5);
89    }
90
91    public void testTwoDShape() {
92      Square s1 = new Square(2.0);
93      Square s2 = new Square(
94                    new Coordinate(1.0, 1.0), 2.0);
95      assertEquals(1.0, s1.distance(s2), .5);
96    }
97
98    public void testTrace() {
99      Trace.TRACELEVEL = 0;
100     Trace.initStream(System.err);
101     Square c = new Square(new Coordinate(0,0),2);
102     c.area();
103   }
104
105   public void testTrace2() {
106     Trace.TRACELEVEL = 1;
107     Trace.initStream(System.err);
108     Square c = new Square(new Coordinate(0,0),2);
109     c.area();
110   }
111 }
```

Figura 6.5: Conjunto de casos de teste adequado para os critérios de teste de unidade, para o programa *Shape*.

Para verificar se o conjunto é adequado para os critérios de teste de unidade pode ser utilizado o ambiente de teste de unidade da ferramenta *JaBUTi/AJ*. Na Figura 6.6 é mostrada uma tela da ferramenta com a cobertura para os critérios de teste de unidade implementados, após a execução do conjunto de casos de teste inicial. Apesar do conjunto ser adequado para os critérios de teste estrutural de unidade, ou seja, obter 100% de cobertura, nenhum defeito é revelado a partir de sua execução. O conjunto obtêm 0% de cobertura para os critérios dependentes de exceção porque não existem requisitos dependentes de exceção no sistema testado.

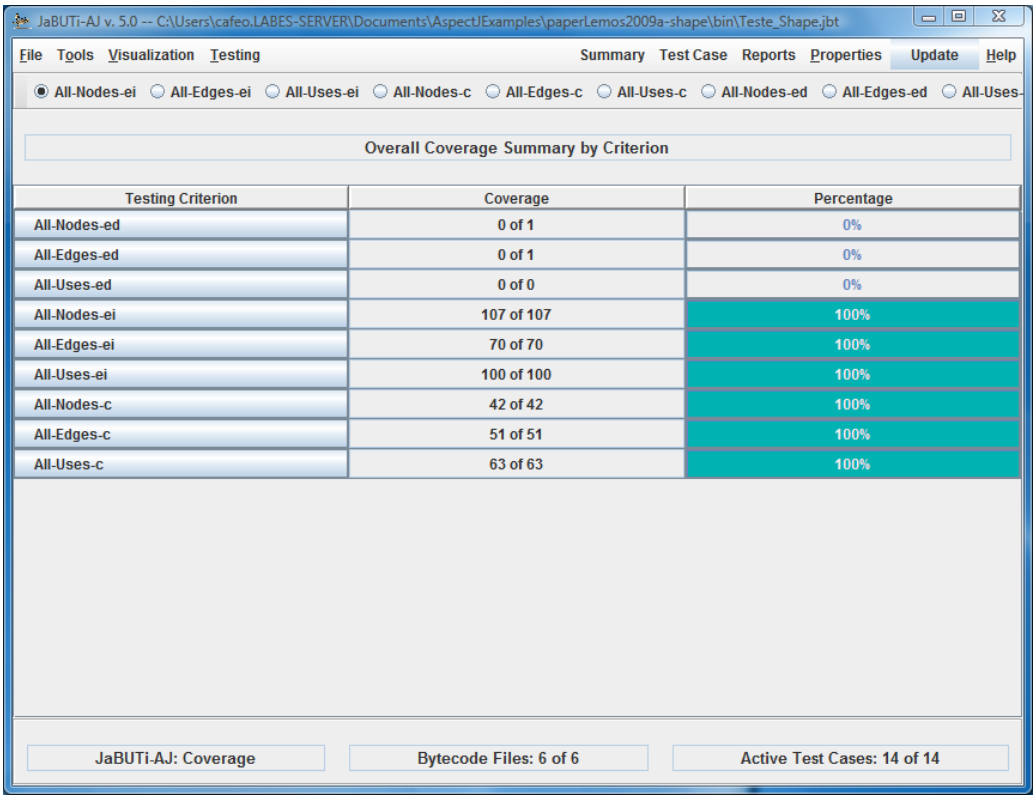


Figura 6.6: Cobertura do programa *Shape* para os critérios de unidade implementados na *JaBUTi/AJ*.

Para efetuar o teste de integração contextual do programa *Shape* na ferramenta *JaBUTi/AJ* é necessário, primeiramente, criar um projeto de teste. Para isso, o testador deve fornecer o nome de uma classe da aplicação a partir da qual os demais aspectos e classes estão relacionados. Para que o projeto abranja um número maior de classes e aspectos foi escolhida a classe `ExampleMain`. Em seguida, a ferramenta abre a janela de gerenciador de projetos (Figura 6.7) na qual os aspectos e classes da aplicação *Shape* são exibidos. Nesta janela o testador seleciona quais classes ou aspectos deseja instrumentar e testar, além de informar um nome para o projeto de teste.

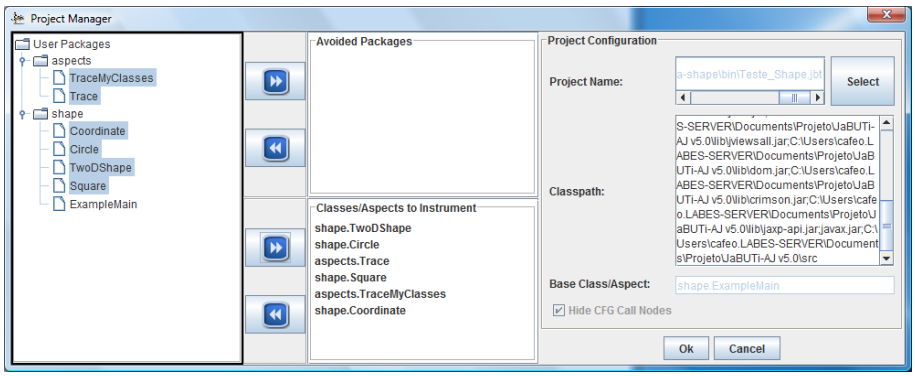


Figura 6.7: Gerenciador de projeto da ferramenta *JaBUTi/AJ*.

Depois de criado o projeto, a ferramenta está pronta para iniciar os testes. Como a ferramenta está inicialmente no teste de unidade, deve-se mudar para o ambiente de teste estrutural de integração contextual. Em seguida, a ferramenta exibe a janela mostrando as unidades e suas profundidades de interação. Dessas unidades, o testador pode escolher quais unidades deseja testar, bem como a profundidade máxima a ser considerada durante o teste.

Como exemplo, foi selecionada a unidade `area` da classe `Circle`. Observe que o método `area` é entrecortado por dois adendos, um do tipo `before` e um do tipo `after returning`, por meio do conjunto de junção definido por `MyMethod` no aspecto `Trace`. O adendo do tipo `before` chama o método `traceEntry` que por sua vez chama o método `printEntering` que chama o método `printIndent`. O adendo do tipo `after returning` chama o método `traceExit` que por sua vez chama o método `printExiting` que chama o método `printIndent`.

Após a seleção da unidade e do nível de profundidade, a ferramenta constrói para cada unidade selecionada um grafo $\mathcal{CODU}$ e calcula os requisitos de teste de cada critério de teste estrutural de integração contextual. Vale ressaltar que, de acordo com a estratégia de teste proposta no Capítulo 5, a unidade `area` deve ser testada incrementalmente em todos os níveis de profundidade.

Supondo que o testador esteja seguindo a estratégia de teste proposta, os casos de teste adequados para os critérios de teste estrutural de unidade (Lemos et al., 2007) são importados e a cobertura dos requisitos do teste de integração é analisada. A execução de um caso de teste percorre um determinado caminho no fluxo do programa. A partir do caminho percorrido, a ferramenta verifica quais nós, quais arestas e quais pares def-uso foram exercitados. Com base nos requisitos de teste derivados pelos critérios de teste de integração e não cobertos pelos casos de teste já existentes, o testador pode então criar conjuntos de casos de teste adequados para cobrir esses requisitos.

A classe `StructuralTS` (Figura 6.5), escrita em JUnit, contém os casos de teste adequados para os critérios de teste estrutural de unidade. Nas Tabelas 6.2, 6.3, 6.4 e 6.5 são mostrados os requisitos cobertos e os não cobertos da unidade `area` após a execução dos casos de teste da classe `StructuralTS` para cada nível de profundidade. As coberturas dos grafos $\mathcal{C}o\mathcal{DU}$s da unidade `area`, considerando os níveis 1, 2, 3 e 4 de profundidade, são mostradas nas Figuras 6.8, 6.9, 6.10 e 6.11, respectivamente. Esses grafos estão exibindo informações sobre a cobertura do critério todos-nós-integrados-N_d. Como pode-se observar nas tabelas e grafos apresentados, a execução do conjunto de casos de teste cobriu parcialmente os requisitos da unidade `area` nas profundidades 2, 3 e 4.

Vale ressaltar que a cobertura em todas as profundidades do método `area` é mostrada nesta seção apenas para ilustrar a utilização do caso de teste inicial diretamente em cada nível. Seguindo a estratégia de teste proposta no Capítulo 5, o caso de teste inicial deve ser incrementado com casos de teste que cubram os requisitos não cobertos nível por nível. Ou seja, nesse caso devem ser criados casos de teste que exercitem os requisitos não cobertos do nível 2 de profundidade e, posteriormente, a cobertura para os níveis 3 e 4 devem ser analisadas.

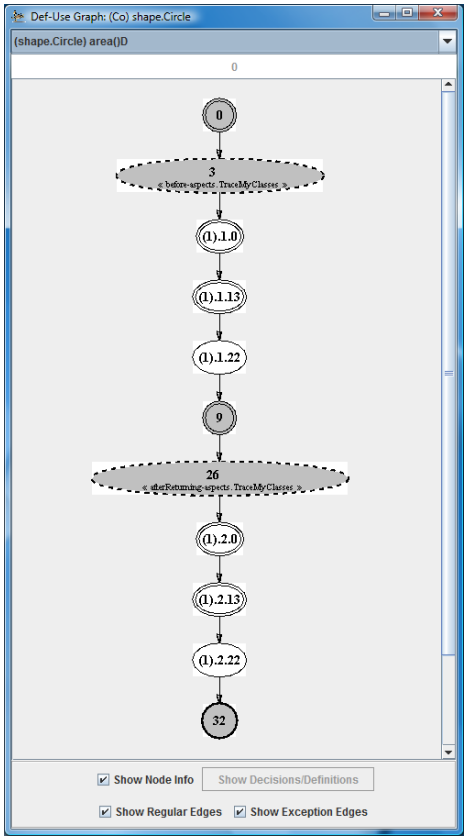


Figura 6.8: Cobertura do grafo $\mathcal{C}o\mathcal{DU}$ do método `area` no nível 1 de profundidade.

Tabela 6.2: Cobertura dos requisitos do método `area` (Nível 1) após a execução do conjunto de casos de teste `StructuralTS`.

Critério	Requisitos não cobertos	Requisitos cobertos	Cobertura
todos-nós-integrados-N_1	–	(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22	100%
todas-arestas-integradas-N_1	–	((1).1.0,(1).1.13); ((1).1.13,(1).1.22); ((1).2.0,(1).2.13); ((1).2.13,(1).2.22)	100%
todos-usos-integrados-N_1	–	–	0%

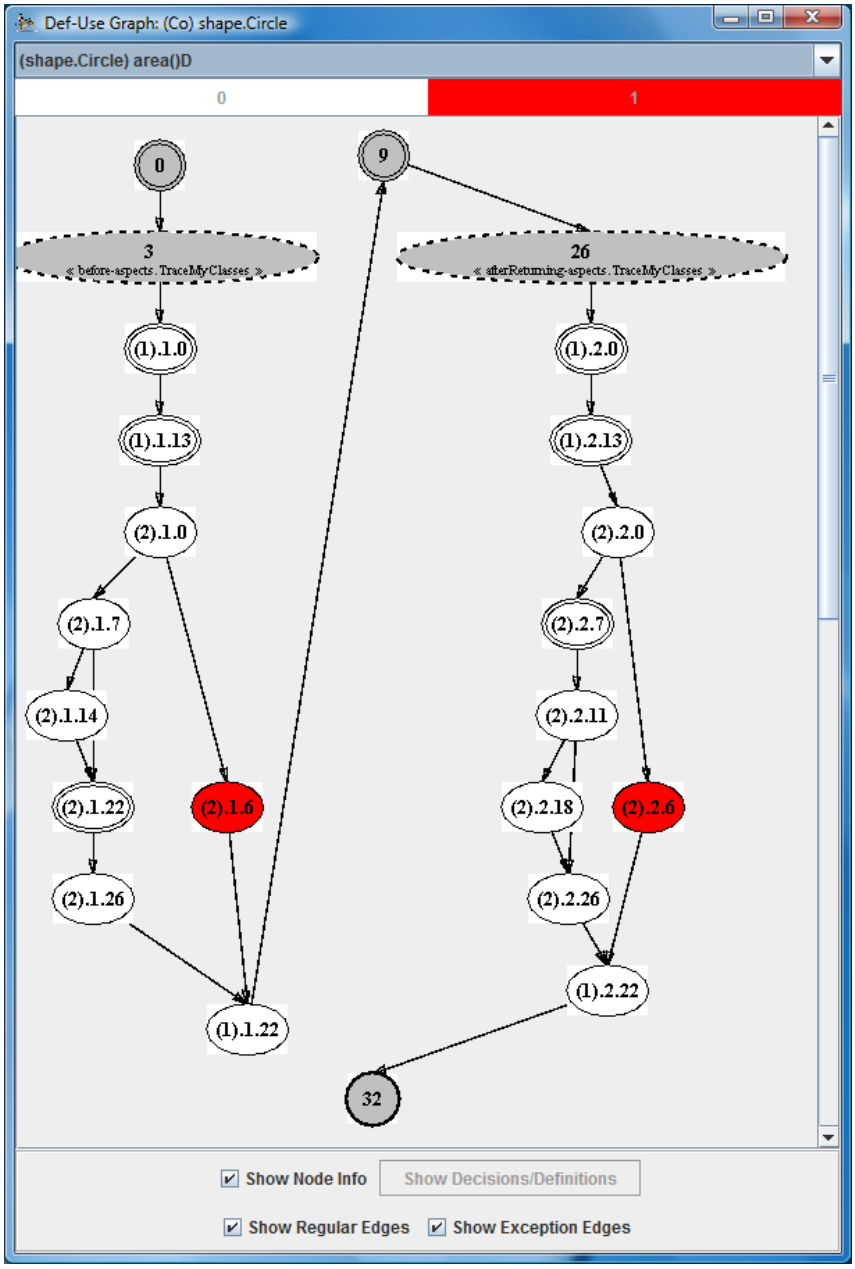


Figura 6.9: Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 2 de profundidade.

Tabela 6.3: Cobertura dos requisitos do método `area` (Nível 2) após a execução do conjunto de casos de teste `StructuralTS`.

Critério	Requisitos não cobertos	Requisitos cobertos	Cobertura
todos-nós-integrados-N_2	(2).1.6; (2).2.6	(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.7; (2).1.14; (2).1.22; (2).1.26; (2).2.0; (2).2.7; (2).2.11; (2).2.18; (2).2.26	88%
todas-arestas-integradas-N_2	((2).1.0,(2).1.6); ((2).1.6,(1).1.22); ((2).1.7,(2).1.22); ((2).2.0,(2).2.6); ((2).2.6,(1).2.22); ((2).2.11,(2).2.26)	((1).1.0,(1).1.13); ((1).1.13,(2).1.0); ((1).2.0,(1).2.13); ((1).2.13,(2).2.0); ((2).1.0,(2).1.6); ((2).1.0,(2).1.7); ((2).1.6,(1).1.22); ((2).1.7,(2).1.14); ((2).1.7,(2).1.22); ((2).1.14,(2).1.22); ((2).1.22,(2).1.26); ((2).1.26,(1).1.22); ((2).2.0,(2).2.6); ((2).2.0,(2).2.7); ((2).2.6,(1).2.22); ((2).2.7,(2).2.11); ((2).2.11,(2).2.18); ((2).2.11,(2).2.26); ((2).2.18,(2).2.26); ((2).2.26,(1).2.22)	70%
todos-usos-integrados-N_2	–	(S@aspects.Trace.callDepth,(2).1.14, (2).2.18)	100%

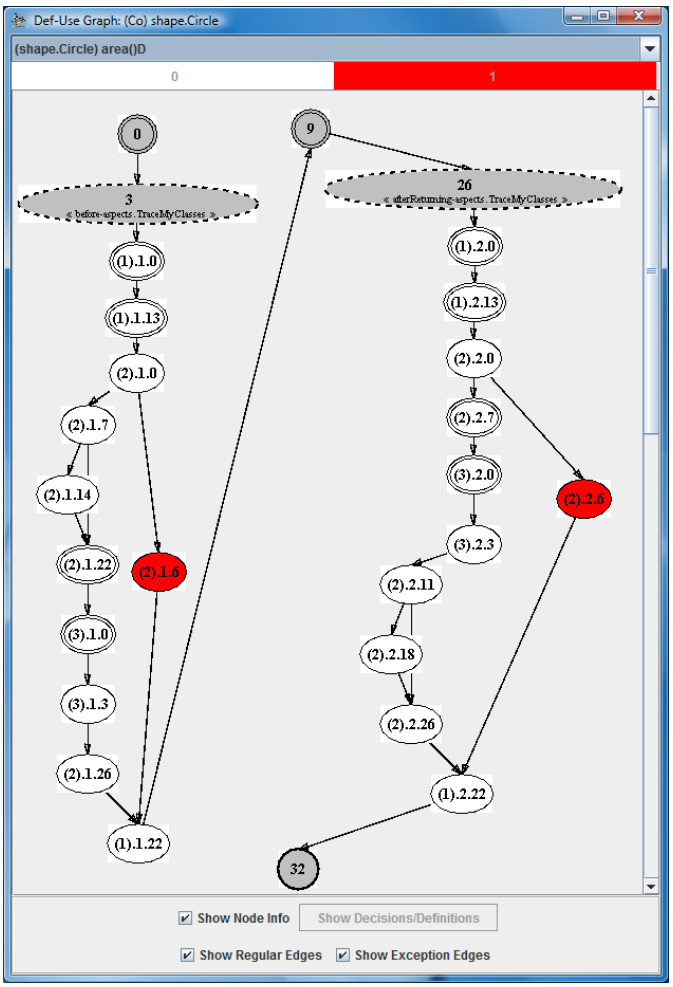


Figura 6.10: Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 3 de profundidade.

Tabela 6.4: Cobertura dos requisitos do método `area` (Nível 3) após a execução do conjunto de casos de teste `StructuralTS`.

Critério	Requisitos não cobertos	Requisitos cobertos	Cobertura
todos-nós-integrados-N_3	(2).1.6; (2).2.6	(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.7; (2).1.14; (2).1.22; (2).1.26; (2).2.0; (2).2.7; (2).2.11; (2).2.18; (2).2.26; (3).1.0; (3).1.3; (3).2.0; (3).2.3	90%
todas-arestas-integradas-N_3	((2).1.0,(2).1.6); ((2).1.6,(1).1.22); ((2).1.7,(2).1.22); ((2).2.0,(2).2.6); ((2).2.6,(1).2.22); ((2).2.11,(2).2.26)	((1).1.0,(1).1.13); ((1).1.13,(2).1.0); ((1).2.0,(1).2.13); ((1).2.13,(2).2.0); ((2).1.0,(2).1.7); ((2).1.7,(2).1.14); ((2).1.14,(2).1.22); ((2).1.22,(3).1.0); ((2).1.26,(1).1.22); ((2).2.0,(2).2.7); ((2).2.7,(3).2.0); ((2).2.11,(2).2.18); ((2).2.18,(2).2.26); ((2).2.26,(1).2.22); ((3).1.0,(3).1.3); ((3).1.3,(2).1.26); ((3).2.0,(3).2.3); ((3).2.3,(2).2.11)	75%
todos-usos-integrados-N_3	–	(S@aspects.Trace.callDepth,(2).1.14, (2).2.18)	100%

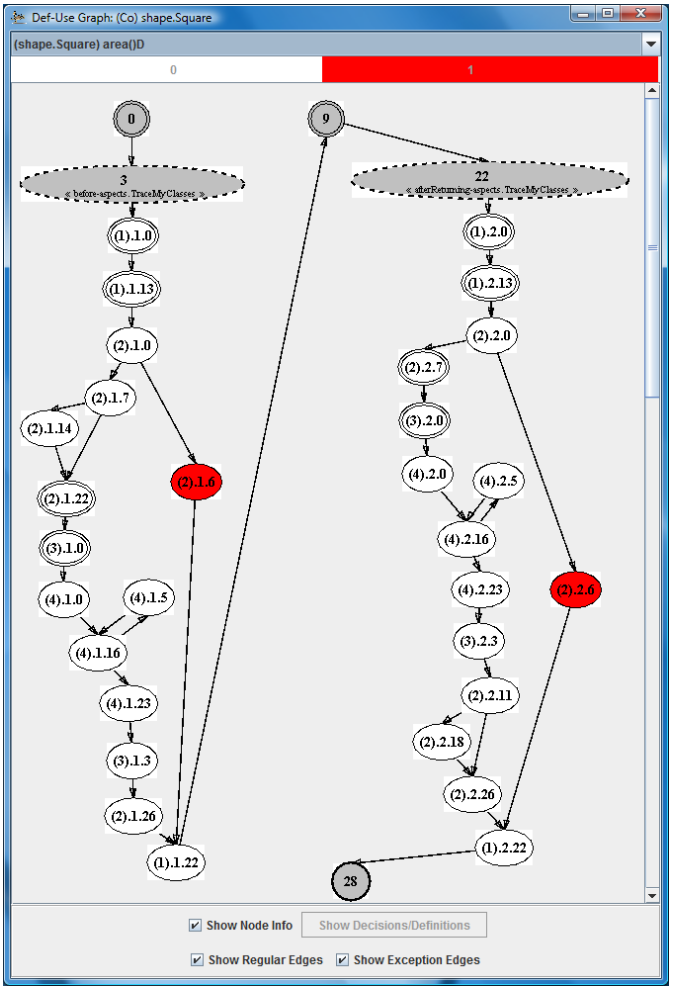


Figura 6.11: Cobertura do grafo $\mathcal{CODU}$ do método `area` no nível 4 de profundidade.

Tabela 6.5: Cobertura dos requisitos do método `area` (Nível 4) após a execução do conjunto de casos de teste `StructuralTS`.

Critério	Requisitos não cobertos	Requisitos cobertos	Cobertura
todos-nós-integrados-N_4	(2).1.6; (2).2.6	(1).1.0; (1).1.13; (1).1.22; (1).2.0; (1).2.13; (1).2.22; (2).1.0; (2).1.7; (2).1.14; (2).1.22; (2).1.26; (2).2.0; (2).2.7; (2).2.11; (2).2.18; (2).2.26; (3).1.0; (3).1.3; (3).2.0; (3).2.3; (4).1.0; (4).1.5; (4).1.16; (4).1.23; (4).2.0; (4).2.5; (4).2.16; (4).2.23	93%
todas-arestas-integradas-N_4	((2).1.0,(2).1.6); ((2).1.6,(1).1.22); ((2).1.7,(2).1.22); ((2).2.0,(2).2.6); ((2).2.6,(1).2.22); ((2).2.11,(2).2.26)	((1).1.0,(1).1.13); ((1).1.13,(2).1.0); ((1).2.0,(1).2.13); ((1).2.13,(2).2.0); ((2).1.0,(2).1.7); ((2).1.7,(2).1.14); ((2).1.14,(2).1.22); ((2).1.22,(3).1.0); ((2).1.26,(1).1.22); ((2).2.0,(2).2.7); ((2).2.7,(3).2.0); ((2).2.11,(2).2.18); ((2).2.18,(2).2.26); ((2).2.26,(1).2.22); ((3).1.0,(4).1.0); ((3).1.3,(2).1.26); ((3).2.0,(4).2.0); ((3).2.3,(2).2.11); ((4).1.0,(4).1.16); ((4).1.5,(4).1.16); ((4).1.16,(4).1.5); ((4).1.16,(4).1.23); ((4).1.23,(3).1.3); ((4).2.0,(4).2.16); ((4).2.5,(4).2.16); ((4).2.16,(4).2.5); ((4).2.16,(4).2.23); ((4).2.23,(3).2.3)	82%
todos-usos-integrados-N_4	–	(S@aspects.Trace.callDepth,(2).1.14, (2).2.18); (S@aspects.Trace.callDepth, (2).1.14, ((4).2.16,(4).2.5)); (S@aspects.Trace.callDepth,(2).1.14, ((4).2.16,(4).2.23))	100%

Após a execução do conjunto de casos de teste adequados para o critério de teste estrutural de unidade, são criados mais 2 casos de teste (Figura 6.12) para cobrir os requisitos não cobertos da unidade `area` nas profundidades 2, 3 e 4.

```
1  public void testTrace1_Circle() {
2    Trace.TRACELEVEL = 0;
3    Trace.initStream(System.err);
4    Circle c = new Circle(new Coordinate(0,0));
5    c.area();
6  }
7
8  public void testTrace2_Circle() {
9    Trace.TRACELEVEL = 1;
10   Trace.initStream(System.err);
11   Circle c = new Circle(new Coordinate(0,0));
12   c.area();
13 }
```

Figura 6.12: Conjunto de casos de teste adicionais para a cobertura dos requisitos não cobertos do método `area` nas profundidades 2, 3 e 4.

Após a execução dos casos de teste adicionais mostrados na Figura 6.12, a cobertura de 100% dos requisitos derivados pelos critérios todos-nós-integrados-N_1, todas-arestas-integradas-N_1, todos-usos-integrados-N_1, todos-nós-integrados-N_2, todas-arestas-integradas-N_2 e todos-usos-integrados-N_2, todos-nós-integrados-N_3, todas-arestas-integradas-N_3, todos-usos-integrados-N_3, todos-nós-integrados-N_4, todas-arestas-integradas-N_4 e todos-usos-integrados-N_4. Dessa forma, o testador pode interroper a geração dos casos de teste e concluir que o teste da unidade `area`, utilizando a abordagem de teste estrutural de integração contextual, foi realizado com sucesso. A

partir daí o testador pode utilizar outras técnicas (por exemplo, o teste baseado em conjuntos de junção) com a finalidade de descobrir defeitos que ainda possam estar presentes no programa.

6.4 Considerações Finais

Neste capítulo foi apresentada uma implementação da abordagem de teste estrutural de integração contextual para programas OO e OA. Essa abordagem foi implementada na ferramenta *JaBUTi/AJ*, e estende versões anteriores da ferramenta (Franchin, 2007; Lemos, 2009; Lemos et al., 2007; Neves, 2009). É importante ressaltar que a ferramenta permite realizar as atividades básicas que devem ser apoiadas por uma ferramenta de teste estrutural: a geração e visualização de um modelo de representação do fluxo de controle e de dados; a instrumentação dos programas a serem testados; a execução dos casos de teste; e a análise de cobertura com base nos critérios definidos e na execução dos casos de teste.

Também foi mostrado o uso dos critérios definidos no Capítulo 5. No entanto, é necessário verificar o custo de aplicação dos critérios. No próximo capítulo é apresentado um estudo da aplicação da abordagem proposta com o intuito de avaliar o custo de aplicação dos critérios de teste propostos.

Capítulo

7

Avaliação da Abordagem de Teste Estrutural de Integração Contextual

7.1 Considerações Iniciais

Neste capítulo é apresentada uma avaliação do custo de aplicação dos critérios de teste estrutural de integração apresentados. A finalidade desta avaliação é investigar se o custo de aplicação dos critérios é viável para permitir seu uso na prática. Para isso foram analisados oito sistemas OA, comumente utilizados em artigos relacionados ao teste de programas OA, que foram testados utilizando critérios de teste estrutural de unidade e os critérios propostos.

Na Seção 7.2 é descrito o estudo do custo de aplicação dos critérios propostos mostrando a caracterização do estudo, os resultados obtidos e as limitações do estudo. Na Seção 7.3 são apresentadas as considerações finais do capítulo.

7.2 Avaliação do Custo de Aplicação dos Critérios de Teste Estrutural de Integração Contextual

Nesta seção é avaliado o custo para adequar um conjunto de casos de teste inicialmente adequado para o teste de unidade de um programa OA aos critérios de teste estrutural de

integração contextual. Considera-se que o programa OA está adequadamente testado do ponto de vista das unidades se o conjunto de casos de teste obtém 100% de cobertura para os critérios todos-nós, todas-arestas e todos-usos adaptados para o teste de programas OO e OA (Lemos et al., 2007), e que essa cobertura continua sendo obtida após a integração das unidades de um programa. Indiretamente, investiga-se também a utilidade da abordagem, mostrando se na maioria dos casos realmente são necessários mais casos de teste para obter uma cobertura de 100% dos critérios de teste de integração contextual, ou se a cobertura dos critérios de teste de unidade é suficiente para cobrir também os critérios propostos. Vale ressaltar que em todas as fases do teste foram desconsiderados os caminhos não executáveis no cálculo da cobertura.

7.2.1 Caracterização do Estudo

Conduziu-se um estudo exploratório para obter evidências da aplicabilidade dos critérios de teste estrutural de integração contextual propostos nesta dissertação. Nesse estudo foram avaliadas duas questões referentes à abordagem proposta nesta dissertação: (1) em quais situações as aplicações necessitam de casos de teste adicionais para cobrir os requisitos gerados pelos critérios de teste estrutural de integração contextual, considerando que as unidades dessas aplicações já foram testadas (100% de cobertura para os critérios todos-nós, todas-arestas, todos-usos)?; e (2) Quando os casos de teste adicionais são realmente necessários, qual é o esforço necessário para tornar o conjunto de casos de teste inicial (adequado para o teste de unidade) adequado para os critérios de teste estrutural de integração contextual? A primeira questão está relacionada à utilidade da abordagem proposta, ou seja, se o conjunto de casos de teste gerado pelo teste de unidade é suficiente para cobrir os requisitos gerados pelos critérios aqui propostos, esses critérios não são úteis. A segunda questão está relacionada com o custo e a aplicabilidade da abordagem, ou seja, critérios que requerem um grande número de casos de teste adicionais podem ser impraticáveis.

As hipóteses do experimento foram que, com respeito à questão (1), a maioria das aplicações necessitariam de casos de teste adicionais para a cobertura dos requisitos de teste gerados com a aplicação dos critérios de teste estrutural de integração contextual. Entretanto, com respeito à questão (2), a hipótese foi de que o número de casos de teste adicionais seria pequeno, pois segundo um estudo conduzido por Souter e Pollock, programas OO geralmente possuem unidades com poucos comandos e com fluxo de controle entre unidades simples (Souter e Pollock, 2000). Ou seja, com as propriedades de unidades descrita por Souter e Pollock, a maioria das interfaces presentes nessas unidades (chamadas a métodos e entrecortes) são exercitadas e apenas poucos caminhos deixariam

de ser cobertos no contexto de execução da unidade sob teste. No entanto, vale ressaltar que tais caminhos podem conter defeitos não descobertos durante o teste de unidade.

Inspirado em um estudo conduzido por Lemos (2009) para avaliar o critério de teste par-a-par de programas OO e OA, foram utilizadas 8 aplicações de diferentes fontes para a condução da avaliação de custo, sendo que algumas dessas aplicações já foram utilizadas em outros experimentos (Lemos, 2009; Programming Tools Group and Sable Research Group, 2011; Rinard et al., 2004; Xu e Rountev, 2007). As seguintes aplicações foram obtidas da distribuição do AspectJ Team (2011) e também utilizadas como benchmark por Programming Tools Group and Sable Research Group (2011): uma aplicação de componentes Java Bean utilizando aspectos (*bean*); uma aplicação 2D que modela figuras e contém um aspecto de rastreamento (*shape*); uma implementação do padrão sujeito-observador utilizando aspectos (*subj-obs*); e uma simulação de um sistema de telefonia utilizando aspectos (*telecom*). Além disso, foram obtidas: uma aplicação de um tutorial de AspectJ para aplicações Enterprise apresentado por R. Bodkin (2005) que simula um serviço de música (*music*); uma aplicação do trabalho de Rinard et al. (2004) que implementa uma pilha com um aspecto que previne que números negativos sejam passados como parâmetros (*stack*); uma aplicação do trabalho de Lemos (2009) que implementa o cálculo de uma função de massa de probabilidade (*math*); e uma aplicação do trabalho de Zhou et al. (2004) que simula uma aplicação bancária (*banking*).

Seguindo a estratégia de teste apresentada no Capítulo 5, primeiramente foram testadas as unidades isoladamente, guiando-se pelo critério de teste de unidade definido por Lemos et al. (2007) e utilizando a ferramenta *JaBUTi/AJ*. Depois disso, utilizando a extensão da ferramenta *JaBUTi/AJ* com a implementação do teste estrutural de integração contextual, foram adicionados casos de teste aos conjuntos iniciais para adequá-los aos critérios de fluxo de controle e de dados propostos nesta dissertação. Nas duas etapas procurou-se obter conjuntos de casos de teste otimizados, ou seja, conjuntos de casos de teste com o mínimo número de casos de teste possível.

7.2.2 Resultados

As seguintes informações foram coletadas para cada programa: número de classes (#C); número de aspectos (#A); número de métodos (#m); número de adendos (#a); número de unidades testadas (#uT), ou seja, número de métodos (#m) e número de adendos (#a); porcentagem de unidades testadas no programa (%uT); profundidade máxima (P); número de linhas de código (#LDC); número de casos de teste requeridos para cobrir os critérios de teste de unidade proposto por Lemos (2005) (#CTu); número de casos de teste adicionados ao conjunto inicial para cobrir os critérios propostos na profundidade máxima (#CTi); e porcentagem de casos de teste adicionais em relação ao número inicial

de casos de teste (%ad.CT). Considera-se o número de casos de teste adicionais como uma medida de esforço para cobrir os critérios de fluxo de controle baseados em conjuntos de junção. Na Tabela 7.1 são mostrados os resultados do estudo.

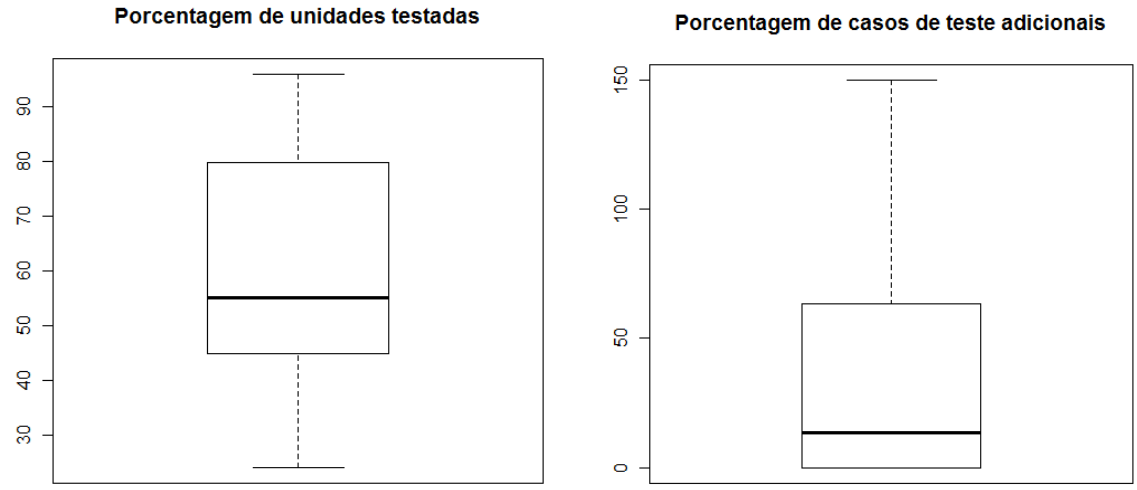
Tabela 7.1: Dados coletados a partir do teste de oito programas OA.

Programa	#C	#A	#m	#a	#uT	%uT	P	#LDC	#CTu	#CTi	%ad.CT
banking	1	2	5	4	6	66,7	4	21	5	3	60
bean	1	1	13	2	9	60	4	66	5	0	0
math	1	1	3	1	2	50	2	69	12	8	66,7
music	11	2	35	3	15	39,5	4	206	22	4	18,2
stack	4	2	10	2	6	50	4	39	5	0	0
shape	5	1	44	4	46	95,8	6	189	14	21	150
subj-obs	5	2	13	1	13	92,9	2	46	6	0	0
telecom	6	3	77	6	20	24,1	3	289	23	2	8,7
Média	4,25	1,75	25	2,88	14,63	59,87	3,63	115,6	11,5	4,75	41,3

Analisando a Tabela 7.1 com o auxílio da Figura 7.1(a), na qual é mostrado um gráfico do tipo *box-plot* que representa o conjunto referente à porcentagem de unidades testadas (%uT), pode-se notar que, em média, o número de unidades testadas foi de aproximadamente 60% do total de unidades (métodos e adendos), o que mostra que a maioria das unidades é testada pela abordagem de teste proposta. Dessa forma, é possível dizer que existem evidências para afirmar que a maioria dos programas terão grande parte de suas unidades testadas pela abordagem proposta.

No entanto, também é importante notar que a dispersão nessa amostra é relativamente alta, já que existem alguns dados de programas que caracterizam *outliers* no gráfico, ou seja, comportamentos que não seguem o padrão observado dentro do conjunto de dados analisado. Um exemplo disso é a porcentagem de unidades testadas no programa `Telecom`. Apenas 24% das unidades foram testadas de forma integrada enquanto a média obtida para o conjunto de programas é de aproximadamente 60%. Isso acontece devido às características específicas do programa `Telecom` que possui vários métodos que não interagem com outras unidades, sendo a maioria deles métodos que realizam cálculos simples e métodos dos tipos *get* e *set*. Outro exemplo de *outlier* nessa distribuição é a porcentagem de unidades testadas do programa `Shape`, no qual 95,8% das unidades existentes no programa foram testadas. Nesse programa a grande maioria dos métodos é entrecortado por pelo menos um adendo e, por isso, há um número alto de unidades testadas no programa.

Portanto, analisando o comportamento padrão do conjunto de dados e as características dos programas que representam os *outliers* do gráfico da Figura 7.1(a) fica evidente que, apesar de haver um certo padrão no número de unidades testadas, a dispersão acentuada é dependente das características dos programas testados.



(a) *Box-plot* da porcentagem de unidades testadas.

(b) *Box-plot* da porcentagem de casos de teste adicionais.

Figura 7.1: *Box-plots* de distribuições coletadas com a condução do experimento.

Também na Tabela 7.1 é possível notar, com o auxílio das colunas #CTi e %ad.CT, que foi necessária a criação de casos de teste adicionais na maioria das aplicações para alcançar a cobertura das unidades em todas as profundidades. Esse resultado confirma a hipótese apresentada de que a maioria das aplicações requer conjuntos de casos de teste mais robustos para os critérios de teste estrutural de integração contextual.

Além disso, é possível observar por meio das colunas #CTi e %ad.CT e do gráfico do tipo *box-plot* da Figura 7.1(b), que o número adicional médio de casos de teste necessários para cobrir os requisitos adicionais gerados pelos critérios propostos não é tão grande (41,3% – 4,75 – casos de teste adicionais na média) se considerarmos que aproximadamente 60% das unidades dos programas foram testadas de forma integrada em todas as profundidades de interação.

Como no caso da análise da porcentagem das unidades testadas, a análise da porcentagem de casos de teste adicionais necessários para cobrir os requisitos gerados pelos critérios de teste estrutural de integração contextual, apesar de apresentar um padrão na distribuição, também apresenta *outliers*. Nesse caso, podemos notar que a média de casos de teste adicionais foi de 41,3% e que na Figura 7.1(b) a maioria da porcentagem de casos adicionais está próxima dos 50% nos casos que exigiram casos de teste adicionais. Porém, em uma das aplicações (*shape*) o número de casos de teste adicionais foi maior que o número de casos de teste inicial (150% de casos de teste adicionais em relação ao número de casos de teste de unidade), pois dois dos adendos apresentam comandos condicionais e afetam o programa em diferentes pontos de junção. Nesse caso, foi necessário gerar uma quantidade maior de casos de teste para cobrir os requisitos gerados pela abordagem.

Com os dados apresentados com relação ao custo de adequação do conjunto de casos de teste de unidade para o teste estrutural de integração contextual, é possível obter indícios da aplicabilidade dos critérios propostos. No entanto, assim como no caso das unidades testadas, as características do programa em teste podem influenciar diretamente no número de casos de teste adicionais necessários para a aplicação dos critérios propostos. Portanto, seria conveniente ampliar o experimento com um número maior de programas testados, com diferentes tamanhos e características.

7.2.3 Limitações do Estudo

Estudos empíricos apresentam limitações que devem ser consideradas na interpretação dos resultados obtidos. No caso do estudo apresentado nesta seção, uma das limitações está relacionada com a representatividade do conjunto de aplicações selecionadas afetando a capacidade de generalização dos resultados obtidos. As aplicações consideradas no estudo são de pequeno porte e aplicações de médio e grande porte ou aplicações comerciais com diferentes características podem levar a resultados diferentes dos obtidos. Além disso, como a extensão da ferramenta *JaBUTi/AJ* se trata de um protótipo, não está claro como a ferramenta se comportaria no teste de grades aplicações. No entanto, em versões anteriores da *JaBUTi*, programas de médio porte foram testados com sucesso e, portanto, há fortes indícios de que a extensão implementada da ferramenta *JaBUTi/AJ* pode testar aplicações maiores do que as usadas no experimento.

Por exemplo, em aplicações no qual existam unidades com vários comandos condicionais e que interajam com várias unidades, é mais provável que seja necessário gerar uma quantidade maior de casos de teste para cobrir os requisitos de teste de integração contextual. Essa conclusão foi obtida em um estudo conduzido por Lemos e Masiero para avaliar o custo de aplicação de diferentes critérios de integração, no qual foi mostrado que o número de casos de teste adicionais está diretamente relacionado com a complexidade, o número e as interações de uma unidade (Lemos e Masiero, 2011). Portanto, já que a grande maioria das unidades em um programa OO têm um fluxo de controle simples, os critérios de teste propostos podem ser considerados aplicáveis na maioria dos casos (Souter e Pollock, 2000).

Um outro caso problemático é quando um adendo apresenta comandos condicionais e afeta um programa em diferentes pontos de junção. Nesse caso, mais uma vez é provável que seja necessário gerar uma quantidade maior de casos de teste para cobrir os requisitos de teste de integração contextual. No entanto, já que adendos são estruturalmente similares a métodos, também é possível considerar que poucas aplicações possuem adendos com um fluxo de controle complexo. Por exemplo, a aplicação *Shape*, apresentada no Capítulo 5 contém adendos que possuem comandos condicionais e que atuam diversas vezes em

diferentes pontos de junção do programa. Se não existissem comandos condicionais nos adendos o número de casos de teste adicionais cairia para 12 em vez de 21. Além disso, segundo Lesiecki (2005), adendos devem conter apenas o código essencial, com o propósito de aumentar a sua testabilidade.

Por fim, um outro problema que afeta de maneira geral o teste estrutural é a geração de requisitos inalcançáveis. Na abordagem proposta esses requisitos podem ser gerados quando a cobertura de certas estruturas (nós, arestas, ou pares def-uso) requer que um caminho seja exercitado, mas esse caminho não pode ser alcançado por nenhum caso de teste. Essa questão é problemática no teste estrutural, pois é um problema indecidível saber se um caminho no programa é ou não é executável. Entretanto, para auxiliar a resolução dessa questão a ferramenta *JaBUTi/AJ* permite ao testador indicar os requisitos que são inalcançáveis e desconsiderá-los do teste.

7.3 Considerações Finais

Neste capítulo foi apresentado um estudo para avaliar o custo de aplicação dos critérios propostos nesta dissertação. Apesar das limitações, os estudos apresentam evidências da aplicabilidade dos critérios. Ressalta-se que para uma validação mais rigorosa dos critérios propostos outros estudos experimentais podem ser conduzidos, principalmente em direção à avaliação de eficácia dos critérios propostos.

No próximo capítulo são apresentadas as conclusões finais deste trabalho e os trabalhos futuros relacionados com esta dissertação.

Capítulo

8

Conclusão

8.1 Considerações Finais

Neste trabalho foi proposta uma abordagem de teste estrutural de integração contextual de programas orientados a objetos e a aspectos. A abordagem inclui um modelo de fluxo de controle e de dados baseado no *bytecode* Java – chamado Grafo Def-Uso Contextual (ou *Contextual Def-Use graph*) – que é uma abstração formada pela integração dos grafos Def-Uso Orientados a Aspectos ($\mathcal{AODU}$) da unidade sob teste com todas as unidades que interagem direta ou indiretamente com ela até um nível de profundidade de interação máximo ou definido pelo testador. Com base nesse modelo foi definida uma família de critérios de teste que avalia um conjunto de casos de teste no que diz respeito à cobertura das interfaces entre as unidades disparadas no fluxo de execução pela unidade sob teste. A família contém três critérios de teste, dois baseados no fluxo de controle (todos-nós-integrados-N_d e todas-arestas-integradas-N_d) e um baseado no fluxo de dados (todos-usos-integrados-N_d).

A ferramenta *JaBUTi/AJ* foi estendida para dar apoio à abordagem de teste estrutural de integração contextual proposta. Com essa extensão é possível efetuar testes de unidade (Lemos, 2005), teste de integração par-a-par (Franchin, 2007), teste de integração nível um (Neves, 2009), teste de integração baseado em descritores de conjuntos de junção (Lemos, 2009), e também teste estrutural de integração contextual de programas OO e OA escritos em Java e AspectJ com apoio da ferramenta.

Para avaliar a aplicabilidade da abordagem de teste proposta foi realizado um estudo experimental. O estudo avalia os critérios propostos no que diz respeito ao custo de aplicação em programas OO e OA. A seguir, na Seção 8.2 são apresentadas as principais contribuições deste trabalho e, por fim, na Seção 8.3 são apresentados alguns trabalhos futuros que podem ser realizados a partir desta dissertação.

8.2 Contribuições

A contribuição principal desta dissertação é uma abordagem de teste estrutural de integração contextual de programas OO e OA. Essa abordagem proposta inclui as seguintes partes relevantes:

1. Um modelo de representação da integração de todas as unidades pertencentes ao fluxo de execução disparado por uma unidade sob teste (o grafo $\mathcal{C}o\mathcal{DU}$);

2. Uma família de três critérios de teste estrutural de integração contextual de programas OO e OA (os critérios todos-nós-integrados-N_d, todas-arestas-integradas-N_d e todos-usos-integrados-N_d); e

3. Uma extensão da ferramenta *JaBUTi/AJ* que implementa os critérios propostos;

Também foi realizada uma avaliação sobre o custo de aplicação dos critérios de teste estrutural de integração apresentados com a finalidade de mostrar se o custo de aplicação dos critérios propostos é viável na prática. Apesar das limitações descritas no Capítulo 7, o resultado da avaliação apresentou evidências da aplicabilidade dos critérios.

8.3 Trabalhos Futuros

Como trabalhos futuros decorrentes desta dissertação, destacam-se principalmente: o refinamento do modelo definido, dos casos especiais e dos critérios de teste; a sugestão das unidades a serem testados, bem como a profundidade de interação a ser considerada durante o teste; a extensão dos critérios e modelo definidos para o teste de comportamentos excepcionais presentes em programas OO e OA; e a otimização da ferramenta *JaBUTi/AJ* para, por exemplo, testar programas de médio e grande porte ou apoiar a geração automática de casos de teste. Além disso, como foi realizado apenas um estudo exploratório sobre o custo de aplicação dos critérios propostos, poderiam ser realizados outros estudos com o objetivo de obter avaliações mais rigorosas sobre o custo de aplicação e avaliações de eficácia dos critérios, especialmente com validade estatística.

Para uma avaliação mais rigorosa dos critérios propostos nesta dissertação, seria necessária a condução de experimentos controlados, nos quais uma quantidade maior de programas de diversos propósitos fosse utilizada. Em particular, um estudo que verificasse a habilidade dos critérios em encontrar defeitos a partir do uso de uma quantidade maior de programas com defeitos de diversos tipos. Trabalhos como os de Alexander et al. (2004); Offutt e Irvine (1995) e Ferrari et al. (2008) podem ser explorados nesse sentido, com a investigação dos tipos de erros específicos de programas OO e OA que podem ser encontrados com o auxílio dos critérios propostos.

Referências

Alexander, R. The real costs of aspect-oriented programming. *IEEE Software*, v. 20, n. 6, p. 92–93, 2003.

Alexander, R. T.; Bieman, J. M.; Andrews, A. A. *Towards the systematic testing of aspect-oriented programs*. Relatório Técnico, Colorado State University and Washington State University, (Acessado em 15/02/2011), 2004.
Disponível em `http://www.cs.colostate.edu/~rta/publications/CS-04-105.pdf`

AspectJ Team The AspectJ Programming Guide. On-line, (Acessado em 02/05/2011), 2011.
Disponível em `http://www.eclipse.org/aspectj`

Bernardi, M. L.; Lucca, G. A. D. Testing aspect oriented programs: an approach based on the coverage of the interactions among advices and methods. *Quality of Information and Communications Technology, International Conference on the*, v. 0, p. 65–76, 2007.

Bertolino, A. Software testing research: Achievements, challenges, dreams. In: *2007 Future of Software Engineering (FOSE '07)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 85–103.

Bodkin, R. Commercialization of AOSD: The road ahead. In: *Proceedings of the Workshop on Commercialization of AOSD Technology – in conjunction with AOSD 2003*, 2003.
Disponível em `http://www.jpmdesign.de/conferences/aosd/2003/papers/AOSD_Commercialization_Position_2003_final.pdf`

Booch, G. *Object-oriented analysis and design with applications*. 2nd ed. Redwood City, Calif.: Benjamin–Cummings, 1994.

Budd, T. A. Mutation analysis: ideas, examples, problems and prospects. In: *Computer Program Testing*, North-Holland, 1981, p. 129–148.

Capretz, L. F. A brief history of the object-oriented approach. *SIGSOFT Software Engineering Notes*, v. 28, n. 2, p. 6, 2003.

Coady, Y.; Kiczales, G. Back to the future: a retroactive study of aspect evolution in operating system code. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*, New York, NY, USA: ACM, 2003, p. 50–59.

Copeland, L. *A practitioner's guide to software test design.* Norwood, MA, USA: Artech House, Inc., 2003.

Czarnecki, K.; Eisenecker, U.; Czarnecki, K. *Generative programming: methods, tools, and applications.* Addison-Wesley Professional, 2000.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software.* Elsevier, 2007.

DeMillo, R. A.; McCracken, W. M.; Martin, R. J.; Passafiume, J. F. *Software testing and evaluation.* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1987.

van Deursen, A.; Marin, M.; Moonen, L. *A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw.* Relatório Técnico SEN-R0507, Stichting Centrum voor Wiskundeen Informatica, Amsterdam, Netherlands, 2005.

Dijkstra, E. W. *A discipline of programming.* Prentice Hall, 1976.

Domingues, A. L. S. *Avaliação de critérios e ferramentas de teste para programas OO.* Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 2001.

Elrad, T.; Filman, R. E.; Bader, A. Aspect-oriented programming: introduction. *Communications of the ACM*, v. 44, n. 10, p. 29–32, 2001.

Ferrari, F. C.; Maldonado, J. C. *Teste de software orientado a aspectos: uma revisão sistemática.* Relatório Técnico 291, ICMC/USP, 2007.

Ferrari, F. C.; Maldonado, J. C.; Rashid, A. Mutation testing for aspect-oriented programs. In: *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, Washington, DC, USA: IEEE Computer Society, 2008, p. 52–61.

Franchin, I. G. *Teste estrutural de integração par-a-par de programas orientados a objetos e a aspectos: critérios e automatização.* Dissertação de mestrado, ICMC-USP, São Carlos, SP, 2007.

Frankl, P. G.; Weyuker, E. J. Testing software to detect and reduce risk. *Journal of Systems and Software*, v. 53, n. 3, p. 275–286, 2000.

Gradecki, J. D.; Lesiecki, N. *Mastering AspectJ: aspect-oriented programming in Java.* New York, NY, USA: John Wiley & Sons, Inc., 2003.

Harrold, M. J.; Rothermel, G. Performing data flow testing on classes. In: *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, New York, NY, USA: ACM, 1994, p. 154–163.

Horgan, J. R.; London, S. Data flow coverage and the c language. In: *Proceedings of the symposium on Testing, analysis, and verification*, New York, NY, USA: ACM, 1991, p. 87–97.

Howden, W. E. *Functional program testing and analysis.* New York, NY, USA: McGraw-Hill, Inc., 1986.

IEEE IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. Getting started with AspectJ. *Communications of the ACM*, v. 44, n. 10, p. 59–65, 2001a.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. An overview of AspectJ. In: *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, London, UK: Springer-Verlag, 2001b, p. 327–353.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J. Aspect-oriented programming. In: *in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS*, 1997, p. 220–242.

Kung, D.; Gao, J.; Hsia, P.; Toyoshima, Y.; Chen, C.; Kim, Y.-S.; Song, Y.-K. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, v. 38, n. 10, p. 75–87, 1995.

Laddad, R. *AspectJ in action: practical aspect-oriented orogramming.* Greenwich, CT, USA: Manning Publications Co., 2003.

Lemos, O. A. L. *Teste de programas orientados a aspectos: uma abordagem estrutural para aspectJ.* Dissertação de Mestrado, ICMC - USP, 2005.

Lemos, O. A. L. *Teste estrutural de integração de programas orientados a aspectos: uma abordagem baseada em conjuntos de junção para AspectJ.* Tese de Doutoramento, ICMC-USP, 2009.

Lemos, O. A. L.; Ferrari, F. C.; Masiero, P. C.; Lopes, C. V. Testing aspect-oriented programming pointcut descriptors. In: *WTAOP '06: Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, New York, NY, USA: ACM, 2006, p. 33–38.

Lemos, O. A. L.; Maldonado, J. C.; Masiero, P. C. Data flow integration testing criteria for aspect-oriented programs. In: *1st Brazilian Workshop on AOSD (WASP 2004) in conjunction with SBES*, 2004.

Lemos, O. A. L.; Masiero, P. C. A pointcut-based coverage analysis approach for aspect-oriented programs. *Information Sciences*, v. 181, p. 2721–2746, 2011.

Lemos, O. A. L.; Vincenzi, A. M. R.; Maldonado, J. C.; Masiero, P. C. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, v. 80, n. 6, p. 862–882, 2007.

Lesiecki, N. Unit test your aspects. On-line, (Acessado em 16/12/2009), 2005. Disponível em `http://www-128.ibm.com/developerworks/java/library/j-aopwork11/`

Linnenkugel, U.; Mullerburg, M. Test data selection criteria for (software) integration testing. In: *Proceedings of the First International Conference on Systems Integration*, 1990, p. 709–717.

Maldonado, J. C. *Critérios potenciais usos: uma contribuição ao teste estrutural de software.* Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, SP, 1991.

Maldonado, J. C.; Barbosa, E. F.; Vincenzi, A. M. R.; Delamaro, M. E.; Souza, S. R. S.; Jino, M. *Introdução ao teste de software.* Relatório Técnico 65, ICMC-USP, 2004.

McCabe, T. J. A complexity measure. In: *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, p. 407.

Microsystems, S. The java tutorial. On-line, (Acessado em 06/10/2009), 2009. Disponível em `http://java.sun.com/docs/books/tutorial/index.html`

Mortensen, M.; Alexander, R. T. *Adequate testing of aspect-oriented programs.* Relatório Técnico CS 01-110, Department of Computer Science, Colorado State University, 2004.

Mortensen, M.; Alexander, R. T. An approach for adequate testing of AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP'2005) – held in conjunction with AOSD'2005*, 2005, p. 1–6.

Mortensen, M.; Ghosh, S.; Bieman, J. Aspect-oriented refactoring of legacy applications: An evaluation. *Software Engineering, IEEE Transactions on*, v. PP, n. 99, p. 1, 2010.

Myers, G. J.; Sandler, C.; Badgett, T.; Thomas, T. M. *The art of software testing.* John Wiley & Sons, 2004.

Neves, V. O. *Teste de integração contextual de programas orientados a objetos e a aspectos: critérios e automação.* Dissertação de mestrado, ICMC-USP, São Carlos, SP, 2009.

Offutt, J.; Irvine, A. Testing object-oriented software using the category-partition method. In: *Seventeenth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95)*, Santa Barbara, CA, USA, 1995, p. 293–304.

Pande, H. D.; Landi, W. A.; Ryder, B. G. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, v. 20, n. 5, p. 385–403, 1994.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, v. 15, p. 1053–1058, 1972.

Pressman, R. *Software engineering: a practitioner's approach.* 6 ed. McGraw-Hill Science, 2005.

Programming Tools Group and Sable Research Group abc: The AspectBench compiler for AspectJ. On-line, (Acessado em 23/02/2011), 2011.
Disponível em `http://www.sable.mcgill.ca/benchmarks/`

R. Bodkin, R. L. Enterprise AspectJ tutorial using eclipse. On-line, (Acessado em 02/05/2011), 2005.
Disponível em `http://www.eclipsecon.org/2005/presentations/EclipseCon2005_EnterpriseAspectJTutorial9.pdf`

Rapps, S.; Weyuker, E. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. SE-11, n. 4, p. 367–375, 1985.

Rapps, S.; Weyuker, E. J. Data flow analysis techniques for test data selection. In: *ICSE '82: Proceedings of the 6th International Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, p. 272–278.

Rinard, M.; Salcianu, A.; Bugrara, S. A classification system and analysis for aspect-oriented programs. *SIGSOFT Softw. Eng. Notes*, v. 29, p. 147–158, 2004.

Roper, M. *Software testing*. New York, NY, USA: McGraw-Hill, Inc., 1995.

Sommerville, I. *Software engineering (8th edition) (International Computer Science Series)*. Addison Wesley, 2007.

Souter, A. L.; Pollock, L. L. Omen: A strategy for testing object-oriented software. *SIGSOFT Software Engineering Notes*, v. 25, p. 49–59, 2000.

Venners, B. *Inside the java virtual machine*. McGraw-Hill Professional, 1999.

Vilela, P. R. S.; Maldonado, J. C.; Jino, M. C. Data flow based integration testing. In: *Proceedings of the 13rd Brazilian Symposium on Software Engineering*, Florianópolis, SC, Brazil: SBCPress, 1999, p. 393–409.

Vincenzi, A. M. R. *Orientação a objeto: Definição, implementação e análise de recursos de teste e validação*. Tese de Doutoramento, ICMC-USP, 2004.

Wallace, D. R.; Fujii, R. U. Software verification and validation: an overview. *IEEE Software*, v. 6, n. 3, p. 10–17, 1989.

Weyuker, E. J. The complexity of data flow criteria for test data selection. *Information Processing Letters*, v. 19, n. 2, p. 103–109, 1984.

Xie, T.; Zhao, J. A framework and tool supports for generating test inputs of AspectJ programs. In: *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, New York, NY, USA: ACM, 2006, p. 190–201.

Xu, G.; Rountev, A. Regression test selection for AspectJ software. In: *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Washington, DC, USA, 2007, p. 65–74.

Zhao, J. Tool support for unit-testing of aspect-oriented software. In: *OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, Seattle, WA, USA, 2002.

Zhao, J. Data-flow-based unit testing of aspect-oriented programs. In: *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, Dallas, TX, USA, 2003, p. 188–197.

Zhou, Y.; Ziv, H.; Richardson, D. J. Towards a practical approach to test aspect-oriented software. In: *Proceedings of the 2004 Workshop on Testing Component-based Systems(TECOS 2004)*, 2004, p. 1–16.

Zhu, H.; Hall, P. A. V.; May, J. H. R. Software unit test coverage and adequacy. *ACM Computing Surveys*, v. 29, n. 4, p. 366–427, 1997.