Teste de Integração Contextual de Programas Orientados a Objetos e a Aspectos: critérios e automação

Vânia de Oliveira Neves

Teste de Integração Contextual de Programas Orientados a Objetos e a Aspectos: critérios e automação

Vânia de Oliveira Neves

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 de Computação e Matemática Computacional.

USP - São Carlos
Novembro/2009

Agradecimentos

Agradeço aos meus pais, Aparecido e Ivanice, que sempre se esforçaram para oferecer o melhor. Obrigada pelo amor, carinho e formação dedicados ao longo da minha vida. Agradeço também ao meu irmão, Fábio, pela amizade e apoio.

Agradeço ao Rafael pelo amor, compreensão, companheirismo e incentivo e aos pais dele, José Roberto e Maria Inêz, pelo apoio, ajuda e carinho.

Ao meu orientador Prof. Dr. Paulo Cesar Masiero pelo muito que me ensinou, por seu profissionalismo e dedicação.

Agradeço aos meus amigos da Unesp: André, Beatriz, Edgar, Ivelize, Lilian e Lucas, por tornar, mesmo à distância, os meus dias mais felizes.

Aos meus amigos do Labes e ICMC/USP: Adriano, Alex Alberto, André Abe, André Endo, André Domingues, Anna Carla, Bruno Cafeo, Chico, Diogo, Draylson, Dusse, Erika, Fabiano, Frotinha, Gabriel, Jorge, José Arnaldo, Kátia, KLB, Leandro, Lucas, Lúcio, Lucía, Marcão, Marcelo Eller, Maria Adelina, Marllos, Mel, Nanico, Nerso, Otávio, Paula, Paulo Henrique, Rafael Messias, Rafael Giusti, Renato Resina, Renato Rodrigues, Ricardo Cerri, Rodolfo, Rodrigo Fraxino, Rodrigo Gondim, Vanessa, Victor e Vinicius e a todos os outros que eu possa ter esquecido mas que não são menos importantes.

Agradeço a todos os meus professores pela minha formação, em especial a professora Maria Lúcia e aos professores do Labes: Ades, Delamaro, Elisa, Ellen, Maldonado, Rosana e Simone. Agradeço também aos funcionários do ICMC, em especial ao sr. Arly pelas "surpresas" no Bloco 1.

Agradeço à Fapesp pelo suporte financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Uma abordagem de teste estrutural de integração contextual para programas OO e OA escritos em Java e AspectJ é apresentada. A finalidade dessa abordagem é descobrir defeitos que possam existir nas interfaces entre uma determinada unidade (método ou adendo) e todas as outras que interagem diretamente com ela, bem como descobrir defeitos que possam ocorrer na hierarquia de chamadas dessas unidades. Para programas OO, esse tipo de teste envolve testar a interação entre métodos; já para programas OA, o teste estrutural de integração nível um (como também pode ser chamado) deve considerar as interações método-método, método-adendo, adendo-adendo e adendo-método. Para efetuar o teste estrutural de integração nível um deve-se considerar todo o fluxo de execução (fluxo de controle e de dados) que ocorre entre uma unidade chamadora e as unidades que interagem diretamente com ela. Para isso é definido o grafo Def-Uso $\mathcal{INIP}$, que é uma abstração formada pela integração dos grafos Def-Uso Orientado a Aspectos ($\mathcal{AODU}$) da unidade chamadora e das unidades que ela chama ou que a afeta. Além disso, são propostos três critérios para derivar os requisitos de teste, dois baseados em fluxo de controle (todos-nós-integrados-N1 e todas-arestas-integradas-N1) e um baseado em fluxo de dados (todos-usos-integrados-N1). A ferramenta JaBUTi/AJ foi estendida para dar apoio à abordagem de teste de integração proposta. Exemplos são apresentados para ilustrar o uso da ferramenta para o teste de profundidade um e também seu uso no contexto de uma abordagem que leva em consideração também o teste de unidades e o teste baseado em conjuntos de junção.

Abstract

A contextual structural integration testing for OO and OA programs written in Java and AspectJ is presented. The purpose of this approach is to discover faults that may exist in the interfaces between a particular unit (method or advice) and all others that interact directly with it, as well as to discover defects that may occur in the call hierarchy of these units. In OO programs, this type of test involves testing the interaction among methods. For OA programs, the structural integration testing at the depth of one (as it can also be called) should consider the method-method, method-advice, advice-advice and advice-method interactions. To perform structural integration testing at the depth of one level the whole execution flow (control and data flow) that occurs among a caller unit and the units that interact directly with it it must be considered. The $\mathcal{INIP}$ Def-Use graph has been defined as an abstraction formed by the integration of the Aspect-Oriented Def-Use ($\mathcal{AODU}$) graphs of the caller unit and of the units that it calls or affects it. Also, three criteria to derive test requirements are proposed, two of which are based on control flow — all-integrated-nodes-N1 and all-integrated-edges-N1 — and one is based on data flow — all-integrated-uses-N1. The tool JaBUTi/AJ was extended to support the proposed integration testing approach. Examples are presented to illustrate the use of the tool for depth 1 testing as well as its use in the context of an approach that also takes into account unit testing and pointcut-based testing.

CAPÍTULO

1

Introdução

1.1 Contextualização

Até o final da década de 60 o desenvolvimento de software era um processo informal e ad hoc, sem o apoio de técnicas e ferramentas, o que levou os programadores a adotarem a abordagem conhecida como "codifica/remenda". O códido produzido era difícil de entender e modificar e as modificações levavam a um código pesado e remendado, chamado de "código espaguete", ocasionando muitos erros e atrasos nos cronogramas. Com o aumento da complexidade do software, aliado ao crescimento da demanda, esse problema se tornou crítico e ocorreu a chamada crise do software. Era evidente que eram necessários métodos mais organizados e práticas mais disciplinadas. Como resultados dessa crise surgiram alguns princípios de projeto e de programação, que empregavam os conceitos de programação estruturada, que possibilitaram o desenvolvimento de código mais legível, fácil de manter e flexível (Boehm, 2006).

Com a evolução desses princípios surgiu a Programação Orientada a Objetos (POO), que atualmente é o paradigma predominante. A POO possibilitou uma melhora significativa no processo de desenvolvimento de software, uma vez que, com ela, é possível o desenvolvimento de sistemas particionados em módulos (classes) capazes de encapsular abstrações de dados juntamente com operações sobre esses dados. Esse paradigma possibilitou uma melhor separação dos diversos interesses de um sistema, com a estruturação de projetos e códigos mais próximos da maneira como os desenvolvedores naturalmente idealizam os sistemas (Elrad et al., 2001b).

Ainda que a POO tenha promovido uma melhora significativa no desenvolvimento de software, nem todos os problemas puderam ser resolvidos. Mais especificamente, as técnicas OO não são suficientes para encaixar determinados tipos de interesses — denominados interesses transversais — dentro de módulos isolados, fazendo com que eles fiquem espalhados ou entrelaçados por todo o sistema. Para tentar solucionar esse problema foi concebida a Programação Orientada a Aspectos (POA), que oferece mecanismos para separar os interesses transversais em módulos distintos (Kiczales et al., 1997). Esses módulos são chamados de aspectos. Uma das linguagens mais conhecidas que apoia a POA é AspectJ.

A evolução significativa da Engenharia de Software, com o estabelecimento de técnicas, métodos e ferramentas, não evita, entretanto, que defeitos sejam introduzidos ao longo do desenvolvimento. As novas abordagens, inclusive, podem introduzir novos tipos de defeitos. Para contornar esse problema, devem ser utilizadas práticas de Garantia de Qualidade de software, entre elas o teste de software, que tem a finalidade de encontrar defeitos presentes em um programa. Técnicas e critérios tem sido desenvolvidos para fornecer uma maneira sistemática para geração e avaliação de conjuntos de teste. As técnicas de teste de software mais conhecidas são: funcional, estrutural e baseada em erros. Elas diferem pela origem da informação utilizada para gerar e avaliar os casos de teste: a técnica funcional (ou caixa-preta) sugere o uso da especificação de requisitos; a técnica estrutural (ou caixa-branca) sugere gerar os casos de teste a partir do conhecimento das características e detalhes internos da implementação; e a técnica baseada em erros sugere que os casos de testes sejam criados a partir do conhecimento dos defeitos típicos inseridos durante o processo de desenvolvimento de software. Essas técnicas são complementares e cada uma propõe um conjunto de critérios de teste que podem ser utilizados tanto para geração de casos de teste quanto para avaliação da adequação desses conjuntos. O teste de software é realizado em três fases: de unidade, de integração e de sistemas. O teste de unidade concentra esforços na menor unidade do programa, o teste de integração procura descobrir defeitos associados às interfaces dessas unidades quando elas interagem com outras unidades e o teste de sistema visa a exercitar o sistema completo.

A técnica de teste estrutural é uma das mais utilizadas e vários critérios para programas orientados a objetos foram propostos por pesquisadores da área (Harrold e Rothermel, 1994; Vincenzi, 2004). Existem hoje algumas propostas na literatura de critérios e ferramentas para o apoio ao teste estrutural de POA e muitas delas são evoluções das abordagens para POO. Este trabalho se insere no contexto do teste de integração estrutural de POA.

1.2 Motivação

Várias pesquisas foram desenvolvidas para o teste estrutural de programas OO e OA, porém poucas exploraram o teste de integração, principalmente para POA. O teste de integração é importante para analisar a relação entre as unidades e seus efeitos quando elas interagem entre si. Segundo Labiche et al. (2000), o comportamento de tais unidades consideradas de forma isolada é, de certo modo, insignificante. Além disso, o fato das unidades apresentarem indícios de conformidade com a especificação e ausência de defeitos não implica no seu correto funcionamento quando integradas.

O uso de ferramentas de teste auxilia na automatização da tarefa de teste, resultando em maior eficiência e redução do esforço necessário para sua realização. Com as ferramentas de teste disponíveis atualmente, ainda há muitas dificuldades para realizar completamente a fase de teste de integração estrutural para programas orientados a aspectos e orientados a objetos.

O grupo de Engenharia de Software do ICMC/USP vem desenvolvendo várias pesquisas sobre o teste estrutural de programas OO e OA e, nesse sentido, foram feitas propostas e desenvolvida a ferramenta JaBUTi/AJ para apoiar o teste de unidades (métodos e adendos) (Lemos, 2005), para apoiar o teste de integração par-a-par entre métodos e adendos (Franchin, 2007) e para apoiar o teste de integração baseado em conjuntos de junção para AspectJ (Lemos, 2009).

Embora o teste de integração par-a-par seja capaz de encontrar defeitos que ocorrem nas interfaces entre o método chamador e o método chamado, ele não é capaz de revelar defeitos que podem ocorrer, por exemplo, nas sequências de chamadas que envolvem mais de um método (ou adendo) chamado. Dessa forma, seria interessante uma proposta de teste estrutural de integração de programas OO e OA que resolvesse esse problema, permitindo que uma unidade seja testada pela técnica estrutural de forma integrada com todas as unidades com as quais ela interage diretamente, seja a chamada de um método ou o entrecorte e execução de um adendo. Ou seja, testar as interações de unidades no contexto de uma unidade dada (nível um (1) de profundidade). Da mesma forma, também é interessante que a ferramenta JaBUTi/AJ seja estendida para dar suporte a esta abordagem, evoluindo para um ambiente mais completo para apoio ao teste estrutural de programas OO e OA.

1.3 Objetivos

O objetivo deste trabalho é propor uma abordagem de teste estrutural de integração contextual de programas orientados a objetos e orientados a aspectos. Essa abordagem é composta pela definição de um grafo para representar o fluxo de controle e de dados de métodos e adendos com

nível um de profundidade, pela definição de dois critérios de fluxo de controle e um de fluxo de dados, além de uma extensão da ferramenta JaBUTi/AJ para apoiar a automação desses critérios e calcular a cobertura alcançada nos testes, seguindo esses critérios.

1.4 Organização

No Capítulo 2 apresenta-se uma revisão bibliográfica sobre teste de software; no Capítulo 3 é apresentada uma introdução sobre programação orientada a objetos e programação orientada a aspectos, abordando, também, as linguagens Java e AspectJ. No Capítulo 4 apresenta-se uma revisão bibliográfica sobre teste de software de programas orientados a objetos e de programas orientados a aspectos, com enfoque no teste estrutural; no Capítulo 5 é proposta uma abordagem de teste estrutural de integração nível um para programas OO e OA. No Capítulo 6 é descrita a implementação da abordagem proposta na ferramenta JaBUTi/AJ e são discutidos alguns exemplos de uso e uma estratégia para teste. Finalmente, no Capítulo 7 são apresentadas as conclusões finais, as contribuições e trabalhos futuros.

CAPÍTULO

2

Teste de Software

2.1 Considerações Iniciais

Este capítulo tem como objetivo apresentar uma revisão bibliográfica sobre teste de software, com ênfase para a técnica de teste estrutural. Ele está organizado da seguinte forma: na Seção 2.2 apresenta-se uma visão geral de teste de software; na Seção 2.3 apresenta-se um exemplo que será utilizado ao longo deste capítulo; as Seções 2.4, 2.5 e 2.6 tratam, respectivamente, das técnicas de teste funcional, estrutural e baseada em defeitos. Por fim, na Seção 2.8 são discutidas as estratégias de teste. O suporte teórico pare este capítulo advém dos seguintes autores: Delamaro et al. (2007), Pressman (2000) e Myers (2004).

2.2 Definição

Muitos programadores definem a atividade de teste como uma atividade que deve ser realizada para mostrar que erros não estão presentes ou, ainda, acreditam que a atividade de teste tem o propósito de demonstrar que o programa realiza suas funções corretamente. Essas são idéias equivocadas, uma vez que o processo de construção de um software depende da habilidade das pessoas que o constrõem, bem como da competência delas para interpretar o processo e

executá-lo. Dessa forma, é indiscutível que os erros humanos sempre acabam surgindo, mesmo com a utilização das melhores ferramentas, técnicas e métodos.

Desse ponto de vista, Myers (2004) define o teste de software como "o processo de execução de um programa com a intenção de encontrar erros."

Com o intuito de descobrir erros antes da utilização do software e garantir que ele está em conformidade com o que foi especificado, várias atividades têm sido utilizadas (que estão dentro do contexto da grande área denominada Garantia de Qualidade de Software), chamadas de "Verificação, Validação e Teste" (ou "VV&T"), em que Verificação refere-se ao conjunto de atividades que garantam que uma aplicação implementa corretamente uma função específica; e Validação refere-se ao conjunto de atividades que garantam que a aplicação desenvolvida corresponda aos requisitos (Rocha et al., 2001).

Para testar um software, executa-se um programa ou um modelo com a utilização de algumas entradas em particular; em seguida, verifica-se se o resultado obtido corresponde ao esperado. Caso isso não ocorra, é possível que um defeito tenha sido encontrado.

Alguns dos termos utilizados no contexto de teste de software dão a impressão que têm o mesmo significado, mas foram padronizados pela IEEE (IEEE, 1990) com definições diferentes. São eles:

- Defeito (*fault*): passo, processo ou definição de dados incorretos;
- Engano (*mistake*): ação humana que produz um resultado incorreto;
- Erro (*error*): ocasionado pela existência de um defeito durante a execução de um programa, que se caracteriza por um estado inconsistente ou inesperado; e
- Falha (*failure*): produção de uma saída incorreta com relação à especificação. Um erro pode levar a uma falha.

Não se pode afirmar que um software não tem defeitos com base na condução de uma atividade de teste, mas é possível mostrar a presença deles, caso existam. O que se pode obter é um grau maior de confiança no programa, caso o teste tenha sido realizado de forma criteriosa e embasada tecnicamente.

Um dos desafios da atividade de teste é que é impossível testar um programa de forma a encontrar todos os seus defeitos. Para evitar esse problema, foram estabelecidas algumas técnicas de teste, entre as quais o teste funcional, o teste estrutural e o teste baseado em defeitos, apresentados nas próximas seções. Antes da apresentação dessas técnicas, será descrita a especificação de um programa utilizado para exemplificar alguns dos critérios.

2.3 Programa Exemplo

O programa P deverá controlar a venda de produtos e ser capaz de calcular a comissão dos vendedores sobre a venda e o valor líquido de cada venda, que é o valor total descontado da comissão do vendedor. A Figura 2.1 apresenta o diagrama de classes deste sistema.

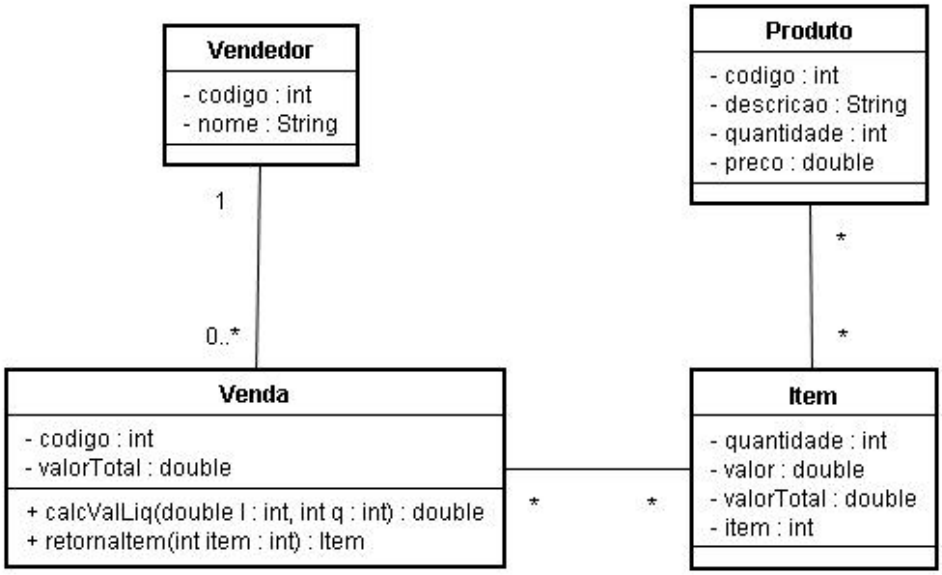

Figura 2.1: Diagrama de classes do sistema exemplo

Ao realizar uma venda, o vendedor e os itens devem ser informados. O valor mínimo de cada item é definido pelo atributo `preco` *da classe* `Produto`*. Cada item pode ser vendido por um preço maior, mas nunca por um preço menor. A comissão do vendedor deverá ser calculada de três diferentes formas, dependendo do lucro obtido com as vendas do mês anterior e da quantidade de itens distintos vendidos:*

Se o lucro obtido pelo vendedor pertencer ao intervalo [0,00;1000,00] e a quantidade de itens vendidos pertencer ao intervalo [0,10], então a comissão será de 10% sobre o valor total da venda.

Se o lucro obtido pelo vendedor pertencer ao intervalo [0,00;1000,00] e a quantidade de itens vendidos pertencer ao intervalo (10, x] | $x \in \mathbb{N}$ *e* $x > 10$*, então a comissão será de 15% sobre o valor total da venda.*

Se o lucro obtido pertencer ao intervalo (1000,00; 5000,00], independente da quantidade de itens vendidos, então a comissão será de 15% sobre o valor total da venda.

Se o lucro obtido pertencer ao intervalo (5000,00, y] | $y \in \mathbb{R}^+$ e $y > 5000,00$ e a quantidade de itens vendidos pertencer ao intervalo [0, 10], então a comissão será de 10% sobre o valor do preço mínimo de cada item, mais 50% sobre a diferença.

Se o lucro obtido pertencer ao intervalo (5000,00, y] | $y \in \mathbb{R}^+$ e $y > 5000,00$ e a quantidade de itens vendidos pertencer ao intervalo (10, x]| $x \in \mathbb{N}$ e $x > 10$, então a comissão será a seguinte: 15% sobre o valor do preço mínimo de cada item mais 50% sobre a diferença.

A Figura 2.2 apresenta o código do método `calculaValorLiquido()`:

```
        public double calculaValorLiquido(double l, int q){
/* 1 */   double comissao = 0;
/* 1 */   double lucroAnterior = l;
/* 1 */   int qtde = q;
/* 1 */   double percFaixa1 = 0.10;
/* 1 */   double percFaixa2 = 0.15;
/* 1 */   double percFaixa3 = 0.5;
/* 1 */   if (lucroAnterior <= 1000){
/* 2 */     if (qtde <= 10)
/* 3 */       comissao = this.getValorTotal()*percFaixa1;
/* 4 */     else
/* 4 */       comissao = this.getValorTotal()*percFaixa2;
/* 4 */   }
/* 5 */   else
/* 5 */     if ((lucroAnterior > 1000) && (lucroAnterior <= 5000)){
/* 6 */       comissao = this.getValorTotal()*percFaixa2;
/* 6 */     }
/* 7 */     else {
/* 7 */       int qtdeAux = qtde;
/* 8 */       while (qtdeAux > 0) {
/* 9 */         Item item = this.retornaItem(qtdeAux);
/* 9 */         double valorItem = item.getProduto().getPreco()*item.getQuantidade();
/* 9 */         if (qtde <= 10) {
/* 10 */          comissao = comissao + valorItem*percFaixa1 +
                        (item.getValorTotal() - valorItem)*percFaixa3;
/* 10 */        }
/* 11 */        else{
/* 11 */          comissao = comissao + valorItem*percFaixa2 +
                            (item.getValorTotal() - valorItem)*percFaixa3;
/* 11 */        }
/* 12 */        qtdeAux--;
/* 12 */      }
/* 12 */    }
/* 13 */  return (this.ValorTotal - comissao);
/* 13 */ }
```

Figura 2.2: Código do método `calculaValorLiquido()`

Como esse programa foi construído com o intuito de exemplificar os critérios de teste, algumas construções podem não ser as mais adequadas.

2.4 Teste Funcional

A técnica de teste funcional é também conhecida como teste caixa-preta, uma vez que ela é indiferente à estrutura e ao comportamento interno do programa. Por esse motivo, o compor-

tamento de um programa em teste só pode ser determinado a partir de suas entradas e saídas relacionadas.

Uma das maneiras de testar um software utilizando essa técnica 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 estão: particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito.

Os critérios de teste funcional baseiam-se na especificação do produto a ser testado. Dessa forma, é fácil perceber que a qualidade desses critérios depende da existência de uma boa especificação de requisitos. Em contrapartida, como não levam em conta detalhes de implementação, esses critérios podem ser aplicados em qualquer fase de teste e em produtos desenvolvidos em qualquer paradigma de programação.

2.4.1 Particionamento em Classes de Equivalência

Como o teste exaustivo é, em geral, impossível de ser aplicado, o testador deve utilizar um subconjunto de todas as possíveis entradas do programa. Dessa forma, um bom subconjunto seria aquele com a maior probabilidade de encontrar a maioria dos erros.

Uma forma de determinar esse subconjunto é dividir o domínio de entrada de um programa em um número finito de classes de equivalência, em que cada elemento se comportaria de forma similar. Ou seja, se um elemento encontrar um erro, todos os outros da mesma classe, também devem fazê-lo; se isso não ocorrer, nenhum outro o encontraria. Há dois passos no projeto de caso de teste pelo particionamento de equivalência: identificação das classes de equivalência e definição dos casos de teste.

Esse critério é recomendado para aplicações em que as variáveis de entrada podem ser identificadas com facilidade, assumindo valores específicos. Porém, o critério não é adequado quando o domínio de entrada é simples, mas o processamento é complexo.

Exemplo: Para a especificação definida acima, o domínio de entrada para P é o produto cartesiano dos conjuntos dos números reais positivos e dos conjuntos dos números naturais; portanto, são entradas válidas os pares $(l, q), |l \in \mathbb{R}^+$ e $q \in \mathbb{N}$. Com base nas faixas definidas na especificação, pode-se particionar o domínio nas seguintes classes válidas:

1. Valores $(l, q)|l \in [0,00; 1000,00]$ e $q \in [0, 10]$
2. Valores $(l, q)|l \in [0,00; 1000,00]$ e $q \in (10, x]$, com $x > 10|x \in \mathbb{N}$

3. Valores $(l, q)|l \in (1000,00; 5000,00]$ e $q \geq 0$

4. Valores $(l, q)|l \in (5000,00; y]$; $q \in [0, 10]$, com $y > 5000,00|y \in \mathbb{R}^+$

5. Valores $(l, q)|l \in (5000,00; y]$; $q \in [10, x]$, com $y > 5000,00|y \in \mathbb{R}^+$ e $x > 10|x \in \mathbb{N}$

No exemplo não são definidas classes inválidas, que são entradas que a implementação deve simplesmente rejeitar, pois, para o caso em questão, quaisquer combinações especificadas de valores do domínio de entrada devem ser processadas.

Dessa forma, com a utilização do critério de particionamento em classes de equivalência, para testar P, basta utilizar um elemento qualquer de cada classe definida acima. Para as classes 1, 2, 3, 4 e 5, são elementos factíveis, respectivamente, (580,00; 3), (720,00; 12), (2500,00; 5), (6500,00; 7) e (7500,00; 12).

2.4.2 Análise do Valor Limite

O critério de análise do valor limite complementa o critério de particionamento em classes de equivalência. De acordo com Pressman (2000), um número maior de erros costuma ocorrer nos limites de entrada. Por isso, ao invés da seleção aleatória dos dados de teste, consideram-se os valores limite desses dados. Outra diferença desse critério para o anterior é que o da análise do valor limite derivam os casos de teste também no domínio de saída.

Exemplo: Considerando as classes definidas acima, alguns dos elementos factíveis para cada classe são:

1. (-0,01;9), (999,99;10)

2. (0,00;10), (1000,01;2.147.483.647)

3. (1000,00; 1), (5000,01,0)

4. (5000,00;-1), (5000,01,-1)

5. (+1.79769313486231570E+308;0),(5000,00,2.147.483.647)

O domínio de saída é o conjunto dos números reais. Dessa forma, deve-se derivar um caso de teste em que a saída seja igual a +1.79769313486231570E+308 e outro em que a saída seja igual a -1.79769313486231570E+308.

Os números 2.147.483.647, +1.79769313486231570E+308 e -1.79769313486231570E+308 são, respectivamente, o maior `int`, o maior e o menor `double` permitido em Java.

2.4.3 Grafo Causa-Efeito

Esse critério, diferentemente dos anteriores, explora as combinações dos dados de entrada. Assim, as causas (condições de entrada) e os efeitos (ações) são identificados na especificação e combinados em um grafo que é uma linguagem formal, na qual a linguagem natural da especificação é traduzida. Segundo Pressman (2000), o processo para derivar os casos de teste é composto pelos seguintes passos:

1. Relacionar as causas e os efeitos e atribuir um identificador a cada um. As causas correspondem a qualquer coisa que provoque uma resposta do sistema e os efeitos, são as saídas ou quaisquer respostas.
2. Desenvolver o grafo causa-efeito, que liga as causas e os efeitos.
3. Converter o grafo em uma tabela de decisão, na qual cada coluna representa um caso de teste
4. Converter as regras da tabela de decisão em casos de teste.

Por meio desse critério, além de exercitar combinações de dados de teste, os resultados esperados são produzidos como parte do processo de criação de teste, ou seja, fazem parte da própria tabela de decisão. Porém, quando o número de causas e efeitos é muito grande, o desenvolvimento do grafo e a conversão dele na tabela de decisão, torna-se mais complexo. Como esse processo é algorítmico, existem ferramentas para auxiliar no seu uso.

2.5 Teste Estrutural

O teste estrutural, também conhecido com teste caixa-branca, utiliza a estrutura lógica do programa para gerar os casos de teste. Essa técnica é tida como complementar à de teste funcional, uma vez que a natureza dos defeitos encontrados por essas técnicas é diferente.

Para representar a estrutura interna do programa é utilizado um Grafo de Fluxo de Controle (GFC) ou Grafo de Programa, a partir do qual os critérios de teste são estabelecidos. A Figura 2.2 apresenta o grafo de fluxo de controle do método `calculaValorLiquido`, apresentado na Figura 2.2. Os critérios de teste estrutural classificam-se em três categorias: com base na complexidade, no fluxo de controle e no fluxo de dados. Cada uma delas é descrita adiante com mais detalhes. Antes disso, são apresentadas algumas definições de conceitos, obtidas de Delamaro et al. (2007), essenciais para o entendimento desses critérios.

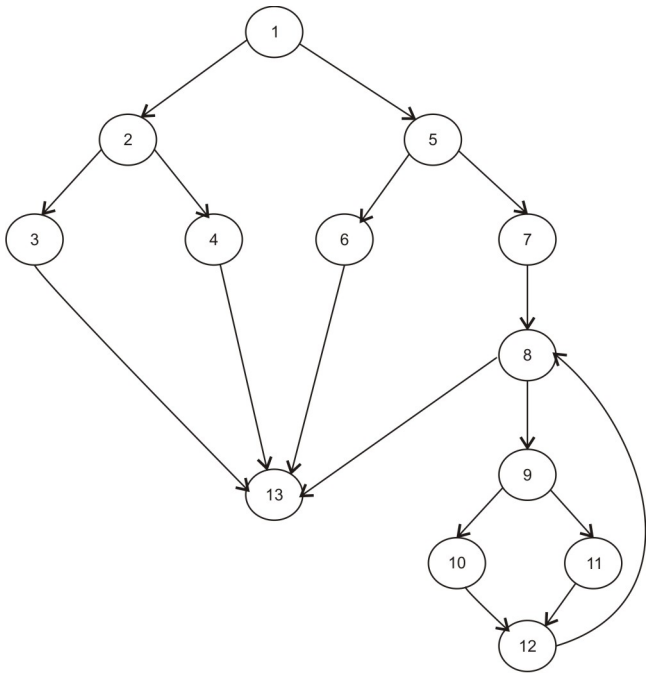

Figura 2.3: Grafo de fluxo de controle do método `calculaValorLiquido()`

Bloco de comandos: um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos na qual a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Dessa forma, todos os comandos de um bloco (com exceção do primeiro) têm um único predecessor e um único sucessor (com exceção do último). Na Figura 2.2, as linhas 1 a 7 representam um bloco de comandos.

Um programa P pode ser representado como um GFC G = (N, E, s), onde:

- N é o conjunto de nós. Os nós representam um bloco indivisível de comandos;
- E representa o conjunto de arestas. As arestas representam o fluxo de controle do programa, ou seja, um possível desvio de um bloco para outro;
- s representa o nó de entrada.

Caminho: é uma sequência finita de nós $(n_1, n_2, \cdots, n_k)$, k $\geq$ 2, tal que existe um arco de n_i para n_i+1 para *i*= 1,2,...,*k*-1.

Caminho simples: um caminho em que todos os nós que o compõem são distintos (com exceção possivelmente do primeiro e do último).

Caminho livre de laço: um caminho em que todos os nós, incluindo o primeiro e o último, são distintos.

Caminho completo: um caminho em que o primeiro nó é o de entrada e o último é o nó de saída do grafo G.

Exemplo: O caminho (1,2,4,13) do grafo da Figura 2.3 é um caminho simples, livre de laço e completo.

Definição de variável: um tipo de ocorrência de uma variável em um programa, em que um valor é armazenado em uma posição de memória. Na Figura 2.2, o comando `double lucroAnterior = l;`, é uma definição de variável.

Indefinição de variável: outro tipo de ocorrência de uma variável em um programa. Ocorre quando não se tem acesso ao seu valor ou sua localização não está definida na memória.

Uso da variável: terceiro e último tipo de ocorrência de uma variável em um programa. Ocorre quando a referência a essa variável não a estiver definindo. Pode ser de dois tipos:

- Uso computacional (c-uso): afeta diretamente uma computação realizada ou permite que o resultado de uma definição anterior possa ser observado. O comando `comissao = this.getValorTotal()*percFaixa1;` na Figura 2.2, corresponde a um uso computacional da variável `percFaixa1`;
- Uso predicativo (p-uso): afeta diretamente o fluxo de controle do programa. O comando `if (lucroAnterior <= 1000)` na Figura 2.2, corresponde a um uso predicativo da variável `lucroAnterior`.

Caminho livre de definição para uma variável *x* dos nós *i* a *j* é um caminho $(i, n_1, \cdots, n_m, j)$, m ≥ 0, que não contem definição de *x* nos nós $n_1, \cdots, n_m$.

Um nó *i* tem uma **definição global** de uma variável *x*, se ocorre uma definição de *x* no nó *i* e existe um caminho livre de definição *i* para algum nó ou para algum arco que contém um c-uso ou um p-uso, respectivamente, da variável *x*.

C-uso global: é um c-uso da variável *x* em um nó *j* em que não existe uma definição de *x* no nó *j* precedendo esse c-uso.

C-uso local: é um c-uso da variável *x* em um nó *j* em que existe uma definição de *x* no nó *j* precedendo esse c-uso.

Complexidade ciclomática: métrica de software que proporciona uma medida quantitativa da complexidade lógica de um programa.

du-caminho em relação à variável *x* é um caminho $(n_1, n_2, \cdots, n_j, n_k)$ em que n_1 tem uma definição global de *x* e: (1) n_k tem um c-uso de *x* e $(n_1, n_2, \cdots, n_j, n_k)$ é um caminho simples livre de definição com relação a *x*; ou (2) (n_j, n_k) tem um p-uso de *x* e $(n_1, n_2, \cdots, n_j, n_k)$ é um caminho livre de definição com relação a *x* e $n_1, n_2, \cdots, n_j$ é um caminho livre de laço.

A seguir serão brevemente apresentados os principais critérios associados às classificações mencionadas anteriormente. É importante ressaltar que os critérios serão tratados no nível de unidade, uma vez que foram inicialmente propostos para o teste de unidades de programas procedimentais. As estratégias de teste de unidade, integração e de sistema são descritas na seção seguinte.

2.5.1 Critérios Baseados na Complexidade

Dentre os critérios dessa categoria está um dos mais conhecidos e também um dos primeiros critérios de teste estrutural definido, o de McCabe (ou teste do caminho básico). Esse critério utiliza a complexidade ciclomática do programa como guia para definir um conjunto básico de caminhos de execução. Além disso, oferece um limite máximo para o número de testes que deve ser realizado para garantir que todas as instruções sejam executadas pelo menos uma vez.

2.5.2 Critérios Baseados em Fluxo de Controle

Os critérios dessa categoria utilizam características do controle da execução do programa para derivar os requisitos de teste. Os mais conhecidos são:

- Todos-nós: exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC; ou seja, que cada comando do programa seja executado ao menos uma vez. A cobertura desse critério é o mínimo esperado em uma atividade de teste. Considerando o grafo de programa apresentado como exemplo na Figura 2.3, alguns dados de entrada (e seus respectivos nós) necessários para satisfazer esse critério estão listados na Tabela 2.1;

- Todas-arestas (ou todos-arcos): requer que cada aresta do grafo, isto é, cada desvio do fluxo de controle do programa, seja exercitada pelo menos uma vez. No grafo de exemplo apresentado na Figura 2.3, os dados de entrada que fazem com que este critério seja satisfeito são listados na Tabela 2.2;

- Todos-caminhos: requer que todos os caminhos possíveis do programa sejam executados. Executar todos os caminhos de um programa é uma tarefa impraticável (embora desejável) uma vez que o número de caminhos é muito grande ou até mesmo infinito. Tomando por base o grafo de programa da Figura 2.3, o número de caminhos neste exemplo é infinito devido a existência de um *loop* (por exemplo, os caminhos (8,9,10,12,8) e (8,9,11,12,8)).

Tabela 2.1: Conjunto de casos de teste que satisfazem o critério todos-nós

Entrada	Nós
(800,5)	1,2,3,13
(800,11)	1,2,4,13
(1100, 10)	1,5,6,13
(5200, 1)	1,5,7,8,9,10,12,8,13
(5200, 11)	1,5,7,8,9,11,12,8,13

Tabela 2.2: Conjunto de casos de teste que satisfazem o critério todas-arestas

Entrada	Arestas
(800,5)	(1-2),(2-3),(3-13)
(800,11)	(1-2),(2-4),(4-13)
(1100, 10)	(1-5),(5-6),(6-13)
(5200, 1)	(1-5),(5-7),(7-8),(8-9),(9-10),(10-12),(12-8),(8-13)
(6400, 11)	(1-5),(5-7),(7-8),(8-9),(9-11),(11-12),(12-8),(8-13)

2.5.3 Critérios Baseados em Fluxo de Dados

Uma das motivações para a introdução dos critérios baseados em fluxo de dados foi que é impraticável executar todos os caminhos de um programa. Além disso, os critérios de fluxo de controle todos-nós e todas-arestas são pouco eficazes para revelar a presença de defeitos, mesmo para programas pequenos (Delamaro et al., 2007).

Os critérios dessa categoria utilizam a análise do fluxo de dados do programa para derivar os requisitos de teste. Eles requerem que sejam testadas as interações que envolvem definições de uma variável e seus usos posteriores.

Uma família de critérios baseados em fluxo de dados é proposta por Rapps e Weyuker (Rapps e Weyuker, 1985). Para derivar os requisitos de teste dessa família foi proposta uma extensão do GFC, o **grafo Def-Uso**, na qual são adicionadas informações a respeito do fluxo de dados do programa.

Para obter o grafo Def-Uso, a partir do GFC associa-se a cada nó i os conjuntos c-uso(i) e def(i), e a cada arco (i,j) o conjunto p-uso(i,j), onde:

- c-uso(i): conjunto de variáveis com c-uso global no bloco i;
- def(i): conjunto de variáveis com definições globais no bloco i;
- p-uso(i,j): variáveis com p-uso no arco (i, j).

Além disso, foram definidos os conjuntos:

- dcu(x,i): conjunto dos nós j, tal que x $\in$ c-uso(j), x $\in$ def(i) e existe um caminho livre de definição com relação a x do nó i para o nó j.

- dpu(x,i): conjunto dos arcos (j, k) tal que $x \in$ p-uso (j,k), x $\in$ def(i) e existe um caminho livre de definição com relação a x do nó i para o arco (j, k).

Ainda utilizando o código-fonte do programa listado na Seção 2.3, construiu-se um novo diagrama com as informações referentes ao fluxo de dados. O resultado é exibido na Figura 2.4.

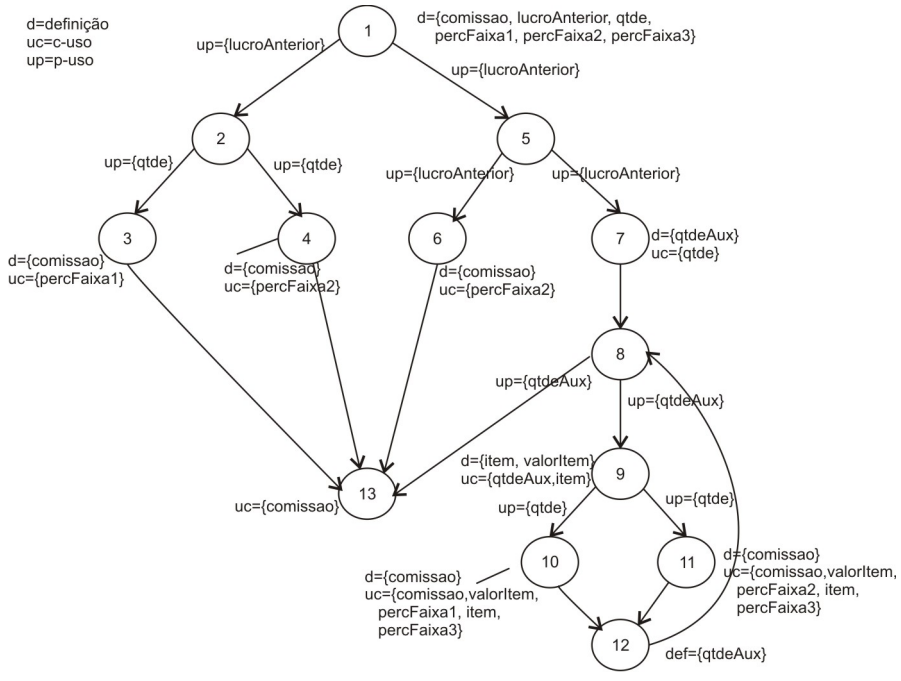

Figura 2.4: Grafo de fluxo de dados do método `calculaValorLiquido()`

Os critérios definidos por essa família são:

- **Todas-definições:** requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um c-uso ou por um p-uso;

- **Todos-usos:** requer que todas as associações entre uma definição de variável e seus subsequentes usos (c-usos e p-usos) sejam exercitadas pelos casos de teste por pelo menos um caminho livre de definição, ou seja, um caminho em que a variável não é redefinida. Dentre as variações desse critérios estão: critérios todos-p-usos, todos-p-usos/alguns-c-usos e todos-c-usos/alguns-p-usos.

- **Todos-du-caminhos:** requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos dessa variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

A maior parte dos critérios baseados em fluxo de dados estabelece que para requerer um determinado elemento (caminho, associação, etc.) exista a ocorrência explícita de um uso de variável. Esse aspecto determina as principais limitações desses critérios e motivou a definição da família de critérios Potenciais Usos (Maldonado, 1991). Os critérios que fazem parte dessa família são:

Todos-potenciais-usos: requer para todo nó *i* e variável *x* para qual exista definição em *i* que pelo menos um caminho livre de definição com relação a *x* do nó *i* para todo nó e para todo arco possível de ser alcançado a partir de *i* seja exercitado;

Todos-potenciais-usos/du: requer para todo nó *i* e variável *x* para qual exista definição em *i* que pelo menos um potencial-du-caminho [1] com relação a *x* do nó *i* para todo nó e para todo arco possível de ser alcançado a partir de *i* seja exercitado;

Todos-potenciais-du-caminhos: requer para todo nó *i* e variável *x* para qual exista definição em *i* que todo potencial-du-caminho com relação a *x* do nó *i* para todo nó e para todo arco possível de ser alcançado a partir de *i* seja exercitado;

2.6 Teste Baseado em Defeitos

Nessa técnica, os requisitos de teste são gerados pela utilização de defeitos típicos do processo de implementação de software. Com a utilização dele é possível medir a qualidade de um conjunto de testes, considerando sua efetividade ou sua habilidade em detectar defeitos (Zhu et al., 1997). Dentre os critérios dessa técnica, destacam-se:

Semeadura de defeitos: nesse critério são introduzidos defeitos artificiais no programa a ser testado de forma aleatória e desconhecida para o testador. Supõe-se que eles representem os defeitos naturais do programa em termos de dificuldade de detecção. Após a introdução dos defeitos artificiais, o programa é testado e os defeitos naturais e artificiais descobertos são contados separadamente. Esse critério pode ser utilizado para medir a qualidade do software. A razão entre o número de defeitos encontrados e o número total de defeitos introduzidos pode ser considerada uma medida de teste de adequação. Uma desvantagem desse critério é que não é fácil introduzir defeitos artificiais equivalentes em dificuldade de detecção em relação aos defeitos naturais (Zhu et al., 1997). Além disso, os defeitos não são uniformes.

Teste de mutação: também conhecido como análise de mutantes, esse critério tem por objetivo alterar o programa que está sendo testado por diversas vezes para construir uma coleção de programas alternativos, chamados de programas mutantes, como se estivessem introduzindo defeitos no programa original. Dado um conjunto de casos de teste para esse programa, cada

[1] Análogo ao du-caminho, mas a ocorrência explícita do uso da variável não é requerida

mutante é executado sobre cada membro do conjunto. A execução para quando o mutante e o programa produzem respostas diferentes (neste caso, diz-se que o mutante "morreu"); ou quando os casos de teste se esgotam (nesse caso, diz-se que o mutante está "vivo"). As razões para que um mutante permaneça vivo são que ou os dados de teste são inadequados ou que o programa mutante é equivalente ao original. Embora esse critério tenha sido originalmente desenvolvido para o teste de unidade, é possível adaptá-lo para outras fases do processo de software (Zhu et al., 1997).

2.7 Fases de Teste

A primeira das fases é a de **teste de unidade**, na qual os testes são realizados no menor módulo do sistema para encontrar erros de lógica de programação e na implementação desses módulos. A fase subsequente é a de **teste de integração** cujo objetivo é testar a integração entre esses módulos, procurando encontrar defeitos em suas interfaces. Por último, é realizado o **teste de sistema** cujo objetivo é assegurar que o sistema e os demais componentes que o compõem funcionem adequadamente e que a função/desempenho global seja obtida.

2.7.1 Teste de Unidade

Conforme dito anteriormente, o teste de unidade concentra esforços na menor unidade do software que, no caso do paradigma procedimental, é o procedimento ou subrotina. Já no paradigma orientado a objetos não há um consenso. Alguns autores consideram a classe como a menor unidade enquanto outros consideram os métodos.

Para realizar o teste de unidade é necessária a implementação de *drivers* e/ou *stubs* para cada unidade de teste. Os *drivers* são responsáveis pela ativação e coordenação do teste da unidade. Dessa forma, ele recebe os dados de teste, passa-os para a unidade a ser testada, obtém os resultados e apresenta-os ao testador. Os *stubs* simulam o comportamento da unidade chamada, servindo para dessa forma, substituir uma unidade utilizada (chamada) pelo módulo que está sendo testado.

2.7.2 Teste de Integração

O objetivo dessa etapa é verificar se as unidades anteriormente testadas individualmente continuarão funcionando quando integradas. Nessa etapa, tenta-se encontrar os defeitos relacionados às interfaces.

Há duas formas de realizar o teste de integração. Em uma delas, chamada integração não-incremental (ou *big-bang*), as unidades são todas combinadas antecipadamente e, dessa forma, o programa todo é testado. No entanto, nessa abordagem a correção dos erros é muito difícil tendo em vista a grande amplitude do programa. Outra forma é a chamada integração-incremental, em que o programa é integrado e testado em pequenas partes, facilitando a identificação e correção de defeitos. Há algumas abordagens de teste de integração-incremental, entre as quais estão (Pressman, 2000):

Integração *top-down*: os módulos são integrados de cima para baixo, levando-se em conta a hierarquia de controle. Dessa forma, o primeiro módulo a ser testado é o de controle principal. A vantagem dessa abordagem é que ela testa, logo no início, as principais funções de controle. Entretanto, pode ser que seja solicitado um processamento nos níveis inferiores da hierarquia, havendo, então necessidade de se ter *stubs*.

Integração *bottom-up*: abordagem que inicia a integração nos níveis mais baixos da estrutura de controle. A principal vantagem dela é que os *stubs* não são necessários, embora o programa não exista como entidade até que o último módulo seja adicionado.

Um melhor ajuste dessas abordagens seria a combinação das duas estratégias (chamada de teste sanduíche), na qual se utiliza a integração *top-down* nos níveis superiores e a *bottom-up* nos níveis inferiores.

2.7.3 Teste de Sistemas

Etapa das mais difíceis, consiste de uma série de testes diferentes e com finalidades diferentes para verificar se todos os elementos do sistema (hardware, bancos de dados, etc.) foram integrados corretamente e realizam as funções atribuídas a eles. Entre os tipos de teste estão:

- **Teste de recuperação:** há sistemas que precisam recuperar-se de falhas e voltar o processamento dentro de um tempo determinado. Esse tipo de teste força o sistema a falhar para verificar se a recuperação é executada adequadamente.
- **Teste de segurança:** verifica se todos os mecanismos de proteção do sistema o protegerão, de fato, dos acessos indevidos.
- **Teste de estresse:** realizado para confrontar os programas com situações anormais. O sistema é executado com o objetivo de exigir recursos em quantidade, frequência ou volume anormais.
- **Teste de desempenho:** realizado para verificar o desempenho em tempo de execução do software.

2.8 Considerações Finais

Neste capítulo foi apresentada uma revisão bibliográfica dos principais conceitos envolvendo teste de software. Foram vistas as técnicas de teste funcional (e os critérios particionamento em classes de equilavência, análise do valor limite e grafo causa-efeito), estrutural (e os critérios baseados na complexidade, em fluxo de controle e em fluxo de dados) e baseada em defeitos (e os critérios semeadura de defeitos e análise de mutantes). Por fim, foram apresentadas as estratégias de teste de unidade, integração e de sistema.

CAPÍTULO

3

Linguagens de Programação OO e OA

3.1 Considerações Iniciais

As técnicas de programação têm evoluído com o passar do tempo, iniciando-se nas linguagens de baixo nível — como linguagens de montagem — para chegar às abordagens de alto nível, como o paradigma orientado a objetos. À medida que a tecnologia avança, a habilidade de se alcançar uma clara separação de interesses aumenta. Nesse contexto surgiu a técnica de programação orientada a aspectos (Elrad et al., 2001b).

Este capítulo apresenta uma revisão bibliográfica da programação orientada a objetos e da programação orientada a aspectos. Na Seção 3.2 é apresentada uma introdução da programação orientada a objetos e, baseada no trabalho de Gosling e McGilton (1996), são discutidas as principais características da linguagem Java; na Seção 3.3 é apresentada uma introdução à programação orientada a aspectos e são abordadas as principais características de AspectJ e na Seção 3.4 são apresentadas as considerações finais.

3.2 Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) possibilita uma melhor aproximação entre modelagem de aplicações e o mundo real, uma vez que simula a realidade utilizando objetos. Uma

aplicação OO pode ser vista como vários objetos interagindo entre si por meio de troca de mensagens, que estimulam alguns comportamentos no objeto receptor. Esses comportamentos, por sua vez, são consumados quando uma operação é executada.

Cada objeto é criado a partir de uma classe e, dessa forma, pode-se dizer que um objeto é uma instância de uma classe, que é composta por um conjunto de dados (atributos) que a descrevem e por um conjunto de operações (métodos) que podem ser realizadas sobre esses dados. Uma classe pode ser entendida como uma abstração de seus objetos, podendo conter várias instâncias, embora cada uma tenha suas próprias características, definidas pelos valores de seus atributos (que define o estado do objeto).

Outros conceitos importantes relacionados ao paradigma OO são encapsulamento, herança e polimorfismo.

Encapsulamento: é a capacidade de o objeto tornar seus dados acessíveis somente por meio de suas próprias operações. No encapsulamento, a interface e a implementação da classe são separadas sintaticamente, de modo que, além de garantir a ocultação de informação, protege os dados contra utilizações arbitrárias, que não respeitam os objetivos iniciais da classe. Outra vantagem é que qualquer alteração feita nos métodos da classe não atinge as demais classes dependentes dela, a menos que haja alteração na interface; essa vantagem possibilita o reuso.

Herança: uma classe pode ser definida a partir de outra classe já existente. Isso permite que a classe criada (subclasse) herde todos os atributos e métodos da classe existente (superclasse), possibilitando, ainda, que novos atributos e operações específicos sejam definidos na subclasse. Esse mecanismo permite criar uma hierarquia de classes, sendo a classe que está no topo da hierarquia a mais genérica e tendo todos os métodos e atributos comuns às demais, ao passo que a classe de nível mais baixo é a mais especializada. Há linguagens de programação que permitem que uma subclasse herde características de mais de uma superclasse, o que é chamado de herança múltipla. A herança também possibilita o reuso.

Polimorfismo: significa "várias formas". Em relação à OO, possibilita que uma operação seja implementada por um ou mais métodos e, para isso, todos os métodos precisam ter o mesmo identificador. A escolha de qual implementação será executada é feita em tempo de execução por meio da técnica de "*late binding*" ou "*run-time binding*".

3.2.1 Java

O paradigma orientado a objetos influenciou o projeto de várias linguagens de programação. Algumas linguagens tradicionais incorporaram conceitos orientados a objetos, como C e Pascal, e deram origem a linguagens híbridas, C++ e Object Pascal, respectivamente. No entanto, é muito fácil o desvio do paradigma orientado a objetos, quando se usa uma linguagem híbrida.

Simula foi a primeira linguagem de programação que teve objetos e classes como conceito central. Ela influenciou o desenvolvimento da Smalltak — a primeira linguagem totalmente orientada a objetos. Por causa da evolução e disseminação de linguagens de programação como Smalltalk, novas linguagens, metodologias e ferramentas apareceram. Entre essas linguagens surgiu a Java, cuja sintaxe é parecida com C e C++ (Capretz, 2003).

A plataforma Java tem dois componentes: A Java Application Programming Interface (Java API) e a Java Virtual Machine (JVM). A primeira é uma coleção de pacotes Java com funcionalidades bastante úteis. Já a JVM é o componente responsável por uma das principais características de Java: sua independência de hardware e de sistema operacional. Isso é possível porque o compilador Java não gera "código de máquina", no sentido de instruções de hardware nativas. Em vez disso ele gera *bytecodes*, que são interpretados pela JVM e estão disponíveis para diferentes sistemas operacionais. Uma vez que isso ocorreu, a JVM envia ao harsware um comando para executar a instrução de fato. A Figura 3.1 exemplifica esse processo (Lindholm e Yellin, 2008). Inicialmente o código fonte do programa (arquivos com extensão .java) é compilado e, então, gerado um arquivo que contém os *bytecodes* (arquivos com extensão .class) que, por sua vez, são interpretados pela JVM.

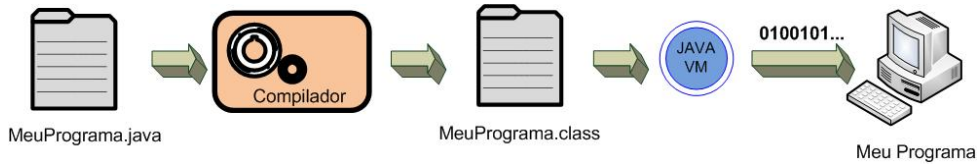

Figura 3.1: Processo de uso da linguagem Java(adaptada de Sun Microsystems (2008))

3.2.2 Características da Linguagem Java

Exceto pelos tipos primitivos, tudo em Java é objeto. Há três tipos de tipos primitivos: tipos numéricos (`byte`, `short`, `int`, `long`, `float` e `double`), tipo caractere (`char`) e tipo booleano (`boolean`, que assume valores `true` e `false`). Cada tipo de dados primitivo possui uma classe correspondente (*wrapper class*): `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Caracter` e `Boolean`. Para trabalhar com textos, Java possui a classe `String`.

Os tipos de referência (classe e array, por exemplo) possuem valores que são referências a objetos criados dinamicamente. Por exemplo, um valor do tipo classe referencia uma instância de classe.

Arrays

Arrays são objetos Java com um determinado número de variáveis (elementos do array) não-nomeadas que são acessadas pelo seu índice. Os arrays podem ser de qualquer tipo e todos os seus elementos são desse tipo. A declaração de um array é feita da forma:

```
<Tipo> [ ] <NomeDoArray>; ou <Tipo> <NomeDoArray> [ ];
```

Exemplo: `String Vendedores []`

Esse código declara um array sem valores iniciais, mas não aloca dinamicamente a quantidade de memória a ser alocada. Para isto, a sintaxe do código é:

```
<Tipo> [ ] <NomeDoArray> = new <Tipo> [NumeroElementos];
```

Exemplo: `String [] Vendedores = new String [15]`

Também é possível realizar a declaração de um array de modo simultâneo com sua iniciação. Por exemplo:

```
String [] Vendedores = {``José'', ``Maria'', ``João''}
```

Java não tem uma declaração específica para declarações de arrays multidimensionais. No entanto, isso é resolvido declarando arrays de arrays.

Gerenciamento de Memória e Coletor de Lixo (*Garbage Collection*)

A responsabilidade pelo gerenciamento da memória em Java é do seu coletor de lixo e não do programador, como é comum em outras linguagens. O coletor de lixo monitora os objetos criados e, quando eles não estiverem mais em uso, desaloca-os da memória.

Classes e Objetos

Conforme dito na seção anterior, em uma classe são definidos os atributos (estado) e os métodos (comportamentos) de um objeto específico que serão construídos a partir dela. Os atributos e métodos são chamados de membros da classe. Uma classe por si só não é um objeto, ela pode ser considerada um modelo que define como um objeto se parecerá e se comportará quando for criado (ou instanciado) a partir de sua especificação. A sintaxe de declaração é a seguinte:

```
<Modificador> class <NomeDaClasse> [extends <NomeDaSuperClasse>] [implements <Interface>]
```

Exemplo: `public class Vendedor extends Pessoa`

`<Modificador>` é formado pelas palavras-chave de controle de acesso e, opcionalmente, pelas palavras-chave `abstract` ou `final`. A primeira define que a classe será abstrata e a segunda que a classe não poderá ser derivada, ou seja, não poderá ser superclasse

de nenhuma classe. A palavra reservada `extends` cria relação de herança com a classe `<NomeDaSuperClasse>`. A palavra reservada `implements` indica que a classe que está sendo declarada implementa a(s) interface(s) incluída(s) na sua declaração. Controle de acesso, classes abstratas, herança e interfaces são discutidos mais adiante, nesta seção.

Um objeto é uma instância de uma classe criada em tempo de execução. A sintaxe da declaração de um novo objeto é:

```
<NomeDaClasse> <VariávelRef> = new <NomeDaClasse> (<ListaParâmetros>)
```

Exemplo: `Vendedor vendedor = new Vendedor(``José'');`

Com isso, foi criado um objeto que será referenciado por `<VariávelRef>` e é uma instância da classe `<NomeDaClasse>`. O acesso aos atributos e métodos dos objetos é conseguido utilizando o operador ponto ".".

Construtores de Classes: ao declarar uma classe em Java, pode-se, opcionalmente, declarar seu construtor (que deve ter o mesmo nome da classe). Ele é um método especial, invocado no momento da criação de um objeto e que determina quais ações devem ser executadas nesse momento.

Subclasses e Superclasses

A relação de herança entre as classes é conseguida pela utilização da palavra chave `extend` na declaração da classe. Uma subclasse é a classe que está sendo declarada a partir de outra classe (a superclasse). Todas as classes em Java são subclasses da classe `Object`, a mais genérica de todas.

Interfaces

Uma interface é um tipo de referência que contém apenas constantes e métodos abstratos. Não há implementação em uma interface, ela é feita na classe que implementa a interface. A sintaxe de declaração de uma interface é a seguinte:

```
                                                    }
interface <NomeInterface> [extends <NomeSuperInterface>] {
...
```

Em Java não existe herança múltipla, mas uma classe pode implementar muitas interfaces.

Controle de Acesso

Ao declarar uma classe em Java, pode-se indicar o nível de acesso permitido aos seus membros. Há quatro níveis de acesso e apenas três podem ser declarados explicitamente: `public`, `protected` e `private`. Os membros declarados como `public` estão disponíveis para qualquer outra classe em qualquer lugar. Membros declarados como `protected` são acessíveis

somente para as subclasses daquela classe. Membros declarados como `private` são acessíveis somente dentro da classe em que eles foram declarados e não estão disponíveis para as suas subclasses. O quarto nível de acesso não tem nome e é obtido se nada for declarado. Ele indica que os membros da classe são acessíveis somente dentro do mesmo pacote, mas inacessíveis a objetos fora do pacote.

Pacote

São coleções de classes e interfaces relacionadas umas com as outras, de alguma forma.

Classes e métodos abstratos

São classes que têm, pelo menos, um método não implementado (método abstrato). Classes abstratas dependem para existir da definição, de pelo menos, uma classe derivada.

3.3 Programação Orientada a Aspectos

Segundo Elrad et al. (2001a), a POO possibilitou uma melhora significativa no processo de desenvolvimento de software, uma vez que com ela é possível obter uma melhor separação de interesses na implementação e no projeto de sistemas. No entanto, alguns desses interesses não podem ser encaixados em módulos isolados e as técnicas OO não são suficientes para capturá-los, obtendo-se uma clara separação desses tipos de requisitos, para que eles fiquem espalhados ou entrelaçados por todo o sistema. Esses interesses, geralmente requisitos não funcionais, são chamados de transversais (*crosscuting concerns*). Como exemplo de interesses transversais podem-se mencionar: políticas de sincronização, controle de acesso, mecanismos de tolerância a falhas e funcionalidades de QoS (Delamaro et al., 2007; Elrad et al., 2001a; Kiczales et al., 1997).

Um exemplo de interesse transversal é apresentado na Figura 3.2. O diagrama de classes da UML refere-se a um editor de figuras simples. Uma `Figura` consiste de `ElementoDeFigura`, que pode ser `Ponto` ou `Linha`. A `Tela` é onde as figuras são desenhadas e, quando um elemento da figura se mover, é necessário que ela seja atualizada. Uma solução utilizando POO seria chamar o método de atualização da `Tela` (`Tela.atualiza()`) no corpo dos métodos `setX`, `setY` e `move` da classe `Ponto` e no corpo dos métodos `setP1`, `setP2` e `move` da classe `Linha`, conforme mostra a Figura 3.3. Note que o interesse de atualização de tela não pertence a nenhuma das duas classes, mas entrecorta ambas.

A Programação Orientada a Aspectos (POA) surgiu como proposta para melhorar a separação de interesses. A POA é uma técnica (e não um paradigma) utilizada em conjunto com os paradigmas de programação existentes (Kiczales et al., 2001). Segundo Lemos et al. (2007),

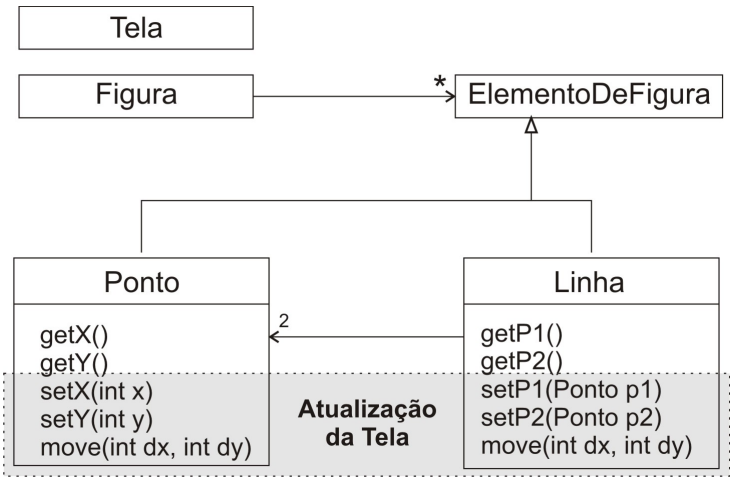

Figura 3.2: Exemplo de interesse transversal (Franchin, 2007)

```
public 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();
  }

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

```
public class Linha implements ElementoDeFigura {
  private Ponto p1, p2;

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

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

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

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

Figura 3.3: Solução OO para o exemplo da Figura 3.2 (Franchin, 2007)

a POA possibilita a criação de módulos isolados, os aspectos, que têm a capacidade de afetar de forma transversal vários outros módulos do sistema. Um único aspecto pode contribuir para a implementação de diversos outros módulos que implementam as funcionalidades básicas (chamados de código-base). Após a programação das classes e dos aspectos, ocorre a combinação desses elementos.

A estrutura da implementação baseada em POA consiste de: linguagem de componentes (módulos básicos), na qual se programam os componentes (ou código-base); linguagem de aspectos, na qual se programam os aspectos; combinador de aspectos, cuja tarefa é combinar os

programas escritos em linguagem de componentes e os escritos em linguagem de aspectos; programas escritos em linguagem de componente, que implementam os componentes, utilizando a linguagem de componente e, por último, programas escritos em linguagem de aspectos, que implementa os aspectos utilizando linguagem de aspectos (Kiczales et al., 1997).

O combinador de aspectos processa os componentes e a linguagem de aspectos, compondo-os para produzir a funcionalidade desejada do sistema. Para isso, alguns conceitos são fundamentais:

- **Pontos de junção:** são pontos específicos na execução do programa em que o comportamento adicional será definido.
- **Conjuntos de junção:** identificam os diversos pontos de junção em um sistema. As regras de combinação podem ser definidas a partir da especificação desses pontos.
- **Adendo:** são construções similares a métodos e definem as ações que deverão ser executadas antes, depois ou no lugar dos pontos de junção identificados.

O processo que une os componentes e os aspectos em um programa executável é chamado de combinação (*weaving*). A combinação dos aspectos pode ser realizada de forma estática ou dinâmica.

3.3.1 AspectJ

AspectJ é a linguagem de programação orientada a aspectos mais utilizada atualmente. Ela é uma extensão para a linguagem Java e foi projetada para ser compatível com ela. Assim, AspectJ contém todas as construções de Java.

AspectJ oferece suporte para o entrecorte estático e dinâmico. O primeiro torna possível alterar a estrutura estática das classes, interfaces e outros aspectos, e definir novas operações em tempo de compilação. As categorias de entrecorte estático são: declarações inter-tipos, modificações da hierarquia de classes e introdução de avisos e erros de compilação. Já no entrecorte dinâmico, é possível definir código adicional para ser executado em pontos bem definidos durante a execução do programa. Ele é baseado nas construções de pontos de junção, conjuntos de junção e adendos. O AspectJ possibilita codificar os aspectos, conjuntos de junção e adendos. Vale ressaltar que os pontos de junção não são codificados na linguagem de aspectos, uma vez que eles são pontos bem definidos na linguagem de componentes.

Conjuntos de Junção: A sintaxe de um conjunto de junção no AspectJ é definida da seguinte forma:

```
<tipo de acesso> pointcut <nomeDoConjuntoDeJunção> ({<lista de parâmetros}) :

                                {tipoDeConjuntoDeJunção [&& | ||] };

<tipo de acesso> = public | private [abstract]

<tipoDeConjuntoDeJunção> = [!] Call | execution | target | args | cflow |cflowbelow |

             staticinicialization | within | if | adviceexecution | preinitialization
```

Na Tabela 3.1 são relacionados alguns tipos de conjunto de junção disponíveis em AspectJ, com as respectivas sintaxes.

Tabela 3.1: Tipos de conjuntos de junção e suas sintaxes

Tipo	Sintaxe
Execução do método	execution(AssinaturaDeMétodo)
Chamada do método	call(AssinaturaDeMétodo)
Execução do construtor	execution(AssinaturaDeConstrutor)
Chamada a construtor	call(AssinaturaDeConstrutor)
Iniciação de classe	staticinicialization(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	inicialization(AssinaturaDeConstrutor)
Pré-iniciação de objeto	preinicialization(AssinaturaDeConstrutor)
Execução de advices	adviceexecution()

Outros tipos de conjunto de junção presentes em AspectJ são os que capturam pontos de junção baseados nos tipos de objeto em tempo de execução. Podem ser de dois tipos: `this`; o objeto corrente, e `target`, o objeto no qual um método é chamado. Há também o tipo de conjunto de junção `args`, que captura pontos de junção baseados nos tipo dos parâmetros do ponto de junção. Os tipos de conjunto de junção `cflow` e `cflowbelow` retornam os pontos de junção na execução do fluxo de outro ponto de junção; o segundo não retorna o ponto corrente. O `withincode` corresponde aos pontos de junção contidos em um método ou construtor; já o `within` corresponde aos pontos de junção contidos em um tipo especifico. Por fim, o `if` permite que uma condição dinâmica faça parte de um conjunto de junção (Gradecki e Lesiecki, 2003).

Um conjunto de junção pode ser composto como combinação de vários outros. Para isso, utilizam-se os operadores lógicos binários 'E' (&&) e 'OU' (||). Para evitar que determinados pontos de junção sejam selecionados, utiliza-se o operador de negação (!).

É possível utilizar caracteres coringas quando for necessário capturar pontos de junção com características em comum. Esses caracteres são: '*' e '..' e denotam qualquer número de

caracteres (o primeiro exclui o caractere '.'; + denota qualquer subclasse ou subinterface de um determinado tipo (que em AspectJ refere-se a uma classe, interface, tipo primitivo ou aspecto).

Adendos: Há três tipos de adendos no AspectJ:

- **Anteriores (before):** o adendo é executado antes de cada ponto de junção.
- **Posteriores**, com três tipos:
  - **o after ():** o adendo é executado depois de cada ponto de junção;
  - **o after returning():** o adendo é executado depois de cada ponto de junção que retorna normalmente;
  - **o after throwing:** o adendo é executado depois de cada ponto de junção que lança uma exceção.
- **De contorno (around):** o adendo é executado no lugar de cada ponto de junção. Com a utilização do método especial `proceed()` ele pode continuar com a execução normal ou causar uma exceção com o contexto alterado.

Além do `this` os adendos têm acesso a outras variáveis especiais que carregam informação sobre o ponto de junção capturado:

- *thisJoinPoint:* encapsula informações estáticas e dinâmicas;
- *thisJoinPointStaticPart:* guarda somente informações do contexto estático;
- *thisEnclosingJoinPointStaticPart:* guarda informações estáticas do ponto de junção mais próximo, acima do interceptado.

Aspectos: Têm a declaração similar de uma classe em Java. A declaração de aspectos, é feita com a palavra chave `aspect`, inclui declarações de adendos, conjuntos de ponto de junção e todas as outras declarações permitidas nas declarações de classe.

Para ilustrar a aplicação de um aspecto, a classe `Venda` do exemplo da Seção 2.3 foi modificada para a classe apresentada na Figura 3.4. O adendo `around` da Figura 3.5 é disparado quando um ponto de junção definido no conjunto de junção é alcançado. Nesse caso, ele será disparado toda vez que houver a chamada dos métodos: `retornaComissaoFaixa1()`, `retornaComissaoFaixa2()` ou `retornaComissaoFaixa3(int qtde)`. Esse adendo aplicará um desconto de 10% sobre a comissão do vendedor. A sequência de execução dos adendos é a seguinte: primeiro o adendo `before`, que imprime a mensagem *Aplicando*

desconto à comissão...; em seguida, o adendo `around` executa no lugar do ponto de junção e por meio do `proceed()` invoca o ponto de junção que, por sua vez, retorna o valor da comissão. O comando seguinte do adendo `around`, aplica a comissão de 10%. Finalmente, o adendo `after` é executado, imprimindo a mensagem *Desconto aplicado.*.

```
public double retornaComissaoFaixa1(){
  return this.getValorTotal()*0.10;
}

public double retornaComissaoFaixa2(){
  return this.getValorTotal()*0.15;
}

public double retornaComissaoFaixa3(int qtde){
  int qtdeAux = qtde;
  double comissao = 0;
  while (qtdeAux > 0) {
    Item item = this.retornaItem(qtdeAux);
    double valorItem = item.getProduto().getPreco()*item.getQuantidade();
    if (qtde <= 10) {
      comissao = comissao + valorItem*0.10 +
                 (item.getValorTotal() - valorItem)*0.5;
    }
    else{
      comissao = comissao + valorItem*0.15 +
                 (item.getValorTotal() - valorItem)*0.5;
    }
    qtdeAux--;
  }
  return comissao;
}

public double calculaValorLiquido(float l, int q){
  double lucroAnterior = l;
  int qtde = q;
  if (lucroAnterior <= 1000){
    if (qtde <= 10)
      return (this.ValorTotal - this.retornaComissaoFaixa1());
    else
      return (this.ValorTotal - this.retornaComissaoFaixa2());
    }
  else
    if ((lucroAnterior > 1000) && (lucroAnterior <= 5000)){
      return (this.ValorTotal - this.retornaComissaoFaixa2());
    }
  else return (this.ValorTotal - this.retornaComissaoFaixa3(qtde));
}
```

Figura 3.4: Classe `Venda` modificada.

O compilador de AspectJ gera *bytecodes* com os aspectos combinados com o código base. Dessa forma, os métodos em que os adendos definem comportamento adicional são sintaticamente inconscientes (*oblivious*) da existência dos aspectos.

```
public aspect aspecto {
  pointcut descontaComissao() :call(* Venda.retornaComissao*(..));

  double around() :descontaComissao(){
    double result = 0;
    result = proceed();
    return result - result*0.10;
  }

  before() :descontaComissao(){
    System.out.println("Aplicando desconto à comissão...");
  }

  after() :descontaComissao(){
    System.out.println("Desconto aplicado.");
  }
}
```

Figura 3.5: Exemplo de um programa AspectJ.

3.3.2 O Processo de Combinação no AspectJ

O processo de combinação do AspectJ acontece, nas versões atuais, no nível de *bytecode* e produz, como resultado, *bytecode* Java puro (Hilsdale e Hugunin, 2004).

Os adendos `after` e `before` modificam o *bytecode* de forma similar: chamadas estáticas aos adendos são incluídas em locais específicos no código afetado. Quando um adendo é associado a um conjunto de junção do tipo `execution`, a chamada do adendo é incluída no *bytecode* do método entrecortado. Por outro lado, quando um adendo é associado a um conjunto de junção do tipo `call`, a chamada do adendo é incluída antes ou depois de cada uma das chamadas ao método entrecortado (Coelho et al., 2009).

A combinação dos adendos `around` é diferente dos adendos `after` e `before`, por causa da execução do método `proceed`. O modo como o adendo `around` é combinado depende de, na hora da compilação, usa ou não a diretiva `-XnoInline`, além do tipo de conjunto de junção ao qual esse adendo está associado. Dessa forma, quando um método é afetado por um adendo `around`, se esse adendo estiver associado a um conjunto de junção do tipo `execution`, o bloco de *bytecode* correspondente ao método original é extraído e substituído por uma chamada ao método `around`. Se o adendo estiver associado a um conjunto de junção do tipo `call`, o bloco de *bytecode* referente a cada uma das chamadas ao método entrecortado é substituído pela chamada ao método `around`.

Um novo método que capta todos os argumentos originais do método entrecortado é criado na classe afetada (no nível de *bytecode*). Se o `around` estiver associado por um conjunto de junção do tipo `execution`, esse novo método encapsula o método que foi entrecortado. Se o `around` estiver associado a um conjunto de junção do tipo `call`, esse novo método conterá uma chamada ao método entrecortado.

Além disso, se ao compilar foi utilizada a diretiva `-XnoInline`, uma nova classe `AroundClosure` é criada. Essa classe contém um método chamado `run` que, por sua vez, contém somente uma chamada para o novo método gerado. Qualquer chamada ao `proceed` no corpo do adendo é representado por uma chamada ao método `run` de `AroundClosure`. Por outro lado, se ao compilar a diretiva `-XnoInline` não for utilzada, a classe `AroundClosure` não é criada e é realizada uma cópia do código do adendo `around` para a classe afetada. Dessa forma, a chamada ao método `run` é substituída pela chamada ao novo método.

O processo de combinação no AspectJ pode acontecer em três diferentes tempos: tempo de compilação, tempo de pós-compilação e tempo de carga da classe na máquina virtual Java (LTW). Os arquivos `.class` produzidos por esses processos são os mesmos, independentemente da abordagem escolhida (AspectJ Team, 2005).

3.3.3 Outras Linguagens OA

Há outras linguagens que implementam aspectos, entre as quais destacam-se: HyperJ, CaesarJ, AspectC++, AspectC, aoPHP, Jiazzi e AspectWerks. Existem também propostas para POA independente de linguagem. A linguagem HyperJ (Tarr et al., 2002) foi desenvolvida pela IBM, com o objetivo de permitir que cada requisito seja implementado em uma dimensão distinta, de modo que o ambiente de compilação faça a composição, de forma semelhante à feita em AspectJ. CaesarJ (Mezini e Ostermann, 2009) é uma linguagem OA para Java que unifica aspectos, classes e pacotes em um única construção, favorendo melhor modularidade e auxiliando no desenvolvimento de componentes reusáveis. A linguagem AspectC++ (Spinczyk et al., 2008) é uma implementação de POA em C++, com projeto baseado na linguagem AspectJ. A linguagem aoPHP (Saunders et al., 2008) é uma extensão para POA do PHP. A linguagem Jiazzi (McDirmid e Hsieh, 2003) é uma extensão de Java, que permite a compilação separada de módulos denominados *units*. As *units* funcionam de forma semelhante a aspectos em AspectJ, podendo alterar a estrutura e o comportamento de classes. AspectWerkz (Bonér e Vasseur, 2008) implementa transversalidade estática e dinâmica, permitindo que o processo de costura de código seja feito durante a compilação ou durante a execução. A sua implementação é flexível, permitindo que código transversal seja implementado como combinação de código Java com anotações e arquivos XML (Couto, 2006).

3.4 Considerações Finais

Neste capítulo foram discutidas a programação orientada a objetos e a programação orientada a aspectos e apresentadas as linguagens de programação Java e AspectJ. O paradigma orientado a objetos possibilitou que os sistemas fossem construídos com uma arquitetura mais organizada, na qual os interesses são implementados em classes separadas. No entanto, com esse paradigma não foi possível obter uma clara separação de todos os interesses (os interesses transversais). Com o objetivo de resolver o problema, surgiu a programação orientada a aspectos, que implementa os interesses transversais em módulos separados, chamados de aspectos.

CAPÍTULO

4

Teste de Software OO e OA

4.1 Considerações Iniciais

Tanto com a Programação Orientada a Objetos quanto com a Programação Orientada a Aspectos tornou-se possível fazer melhores projetos de software. No entanto, com elas também surgiram novos defeitos e, dessa forma, a tarefa de teste permanece importante. Neste capítulo apresenta-se uma revisão bibliográfica referente ao teste de software de programas orientados a objetos e de programas orientados a aspectos, com enfoque no teste estrutural. Ele está organizado da seguinte forma: na Seção 4.2 apresentam-se as abordagens para teste estrutural de programas orientados a objetos; na Seção 4.3 discutem-se as abordagens para teste estrutural de programas orientados a aspectos; na Seção 4.4 apresentam-se as estratégias de ordenação de classes e aspectos; na Seção 4.5 descrevem-se algumas das ferramentas de teste existentes para apoio de POO e de POA; na Seção 4.6 há uma tabela comparativa entre as abordagens consideradas e, por fim, na Seção 4.7 estão as considerações finais deste capítulo.

4.2 Teste Estrutural de Programas Orientados a Objetos

As primeiras abordagens para teste estrutural de programas orientados a objetos foram adaptadas das abordagens para o paradigma procedimental. No entanto, o paradigma OO tem determinadas características que não estão presentes no paradigma procedimental, conforme discutido na Seção 3.2, apresentando novos desafios para o teste de software.

O fato de não haver um consenso em relação a qual seria a menor unidade de teste para um programa OO (conforme já dito neste texto, alguns autores consideram o método, outros consideram a classe) acarreta uma abordagem de teste diversa, em que o teste de unidade e o teste de integração devem ser vistos de uma perspectiva diferente. A Tabela 4.1 apresenta os tipos de teste OO aplicados em cada fase, de acordo com a abordagem adotada.

Tabela 4.1: Relação entre as fases de teste e o teste de programas OO (adaptado do trabalho de Domingues (2002))

	Menor Unidade: Método
Fase	*Teste de Software Orientado a Objetos*
Unidade	Intramétodo
Integração	Intermétodo, Intraclasse e Interclasse
Sistema	Toda a aplicação
	Menor Unidade: Classe
Fase	*Teste de Software Orientado a Objetos*
Unidade	Intramétodo, Intermétodo e Intraclasse
Integração	Interclasse
Sistema	Toda a aplicação

4.2.1 Abordagem de Harrold e Rothermel (1994)

Uma das primeiras abordagens para teste estrutural de programas orientados a objetos foi proposta por Harrold e Rothermel (1994) e se aplicava ao teste de fluxo de dados em classes. Para testar métodos individuais (teste intramétodo) e métodos de uma mesma classe que interagem entre si (teste intermétodo), a técnica é similar às existentes no paradigma procedimental (teste intraprocedimental e teste interprocedimental, respectivamente). No entanto, no paradima OO, métodos acessíveis externamente podem ser invocados em qualquer ordem. Dessa forma, as interações de fluxo de dados devem ser consideradas e, para isso, foi definido o Grafo de Chamadas de Classe (*Class Call Graph*). Neste grafo os nós representam os métodos e as arestas, as chamadas entre os métodos. Como exemplo, considere-se a implementação parcial da classe `SymbolTable` codificada em C++ e apresentada na Figura 4.1; o seu Grafo

de Chamadas de Classe correspondente pode ser visto na Figura 4.2. As linhas pontilhadas representam chamadas de fora a métodos públicos dessa classe.

```
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)

Dessa forma, nesse trabalho (em que a classe é considerada como a menor unidade de testes) foram definidos três níveis de teste: o **intramétodo** e o **intermétodo** que, como já dito anteriormente são análogos aos do paradigma procedimental; e o **intraclasse**, cuja definição é a seguinte:

- **intraclasse:** testa as interações de métodos públicos quando eles são chamados em várias sequências. Considerando que o conjunto das possíveis chamadas de sequência de métodos públicos é infinito, pode-se testar somente um subconjunto desse conjunto. Utilizando novamente a implementação da Figura 4.1 como exemplo, o teste intraclasse pode ser realizado pela seleção de sequências de teste como <`SymbolTable, AddToTable, GetFromTable`> e <`SymbolTable, AddToTable, AddToTable`>.

Com isso, há três tipos de pares def-uso na classe, correspondentes aos níveis de teste acima: **os pares def-uso intramétodo**, **pares def-uso intermétodo** e os **pares def-uso intraclasse**. Os dois primeiros são semelhantes aos do paradigma procedimental. O par def-uso intraclasse é definido como segue: considerando C uma classe a ser testada, seja *d* uma sentença contendo

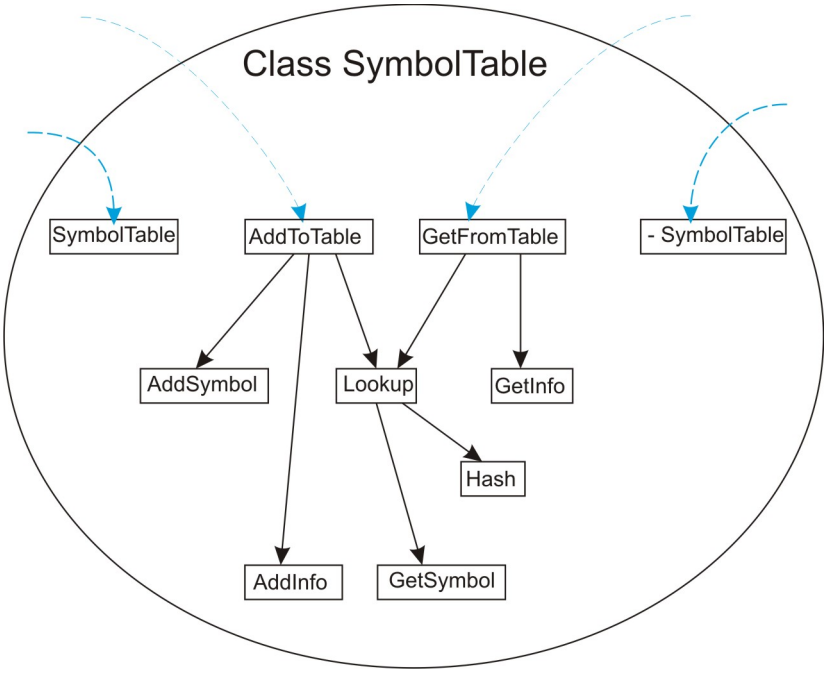

Figura 4.2: Grafo de chamadas de classe para a classe `SymbolTable` (Harrold e Rothermel, 1994)

uma definição de uma variável *v*, e *u* representando uma sentença que contém um uso dessa variável. Seja M_0 um método público de C, e seja $\{M_1, M_2, \cdots, M_n\}$ um conjunto de métodos em C chamados, direta ou indiretamente, quando M_0 é invocado. Seja N_0 um método público em C (possivelmente o mesmo método que M_0), e seja $\{N_1, N_2, \cdots, N_n\}$ um conjunto de métodos em C chamados, direta ou indiretamente, quando N_0 é invocado. Suponha que d está em algum método em $\{M_0, M_1, M_2, \cdots, M_n\}$, e u está em algum método em $\{N_0, N_1, N_2, \cdots, N_n\}$. Se existir um programa P que chama M_0 e N_0, tal que em P, (*d*,*u*) é um par def-uso, e tal que depois que *d* é executado e antes que *u* seja executado, a chamada de M_0 termina; então, (*d*,*u*) é um par def-uso interclasse. Um exemplo é a sequência `<AddToTable, GetFromTable>` da Figura 4.1. 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`. Dessa forma, a definição de uma informação na tabela (linha 73) e uso da tabela (linha 82) gera um par def-uso intraclasse.

Para computar as informações de fluxo de dados requeridas para o teste intramétodo e para o teste intermétodo, os pesquisadores afirmam que pode ser utilizado o algoritmo de Pande et al. (1994), que constrói um grafo de fluxo de controle interprocedimental para um programa que combina o grafo de fluxo de controle para procedimentos individuais. Para os pares def-uso intraclasse esse algoritmo não pode ser usado diretamente. Dessa forma, os pesquisadores

desenvolveram uma nova representação do grafo, o Grafo de Controle de Fluxo de Classes (*CCFG — Class Control Flow Graph*), que *conecta* todos os métodos de uma classe.

O algoritmo para a construção desse grafo funciona da seguinte forma: inicialmente, tendo como entrada uma classe C, é gerado o Grafo de Chamada de Classes G. Em seguida, G é envolvido em um *frame* — que permite que as chamadas aos métodos sejam realizadas em qualquer sequência para a computação dos pares def-uso intraclasse. Esse *frame* contem cinco nós: *frame entry* e *frame exit*, que representam a entrada e a saída do *frame*, respectivamente; *frame loop*, que facilita o sequenciamento dos métodos; e os nós *frame call* e *frame return*, que representam a chamada e o retorno de qualquer método público, respectivamente. Esse frame também pode conter quatro arestas: (*frame entry, frame loop*), (*frame loop, frame call*), (*frame loop, frame exit*) e (*frame return, frame loop*). Nesse ponto da construção do CCFG, o frame e o grafo de chamada de classe não estão conectados. Cada nó de G representa um método M e, assim, na sequência, cada nó é então substituído pelo Grafo de Fluxo de Controle de M; as arestas de G também são atualizadas. Na Figura 4.3 é apresentado o CCGF parcial da classe `SymbolTable`. Nela, apenas os nós e os grafos correspondentes aos métodos `AddToTable`, `GetFromTable` e `Lookup` são expandidos. Os *Cs* contidos na figura correspondem às chamadas de métodos e os *Rs* correspondem ao retorno deles.

Para realizar o teste de integração de classes, Harrold e Rothermel (1994) definem um quarto nível de teste: o interclasse, que considera os pares (*d*,*u*) tal que *d* e *u* estão em classes diferentes. Nesse nível de teste pode-se computar os pares usando as interações do CCFG.

4.2.2 Abordagem de Vincenzi (2004)

Vincenzi (2004) apresenta uma abordagem de fluxo de controle e de dados intramétodo para programas orientados a objetos. Ele considera o método como a menor unidade a ser testada e, para representar o fluxo de controle e de dados, o grafo Def-Uso ($\mathcal{DU}$) é utilizado. No entanto, antes de construir o $\mathcal{DU}$, constrói-se o *Grafo de Instruções de fluxo de dados ($\mathcal{IG}$)* para cada método. Informalmente, um $\mathcal{IG}$ é um grafo no qual cada nó contém uma única instrução de *bytecode* e as arestas conectam as instruções que podem ser executadas em sequência. Ele também contém informações a respeito do tipo de acesso (definição ou uso) que as instruções de *bytecode* fazem.

Para lidar com o mecanismo de tratamento de exceção de Java, as representações de $\mathcal{IG}$ e/ou $\mathcal{DU}$ refletem o fluxo de controle durante a execução normal do programa e durante a ocorrência de exceções. Para representar fluxo de controle regular e de exceção, dois tipos de arestas são utilizados: as arestas regulares, que representam o fluxo de controle regular (isto é, que

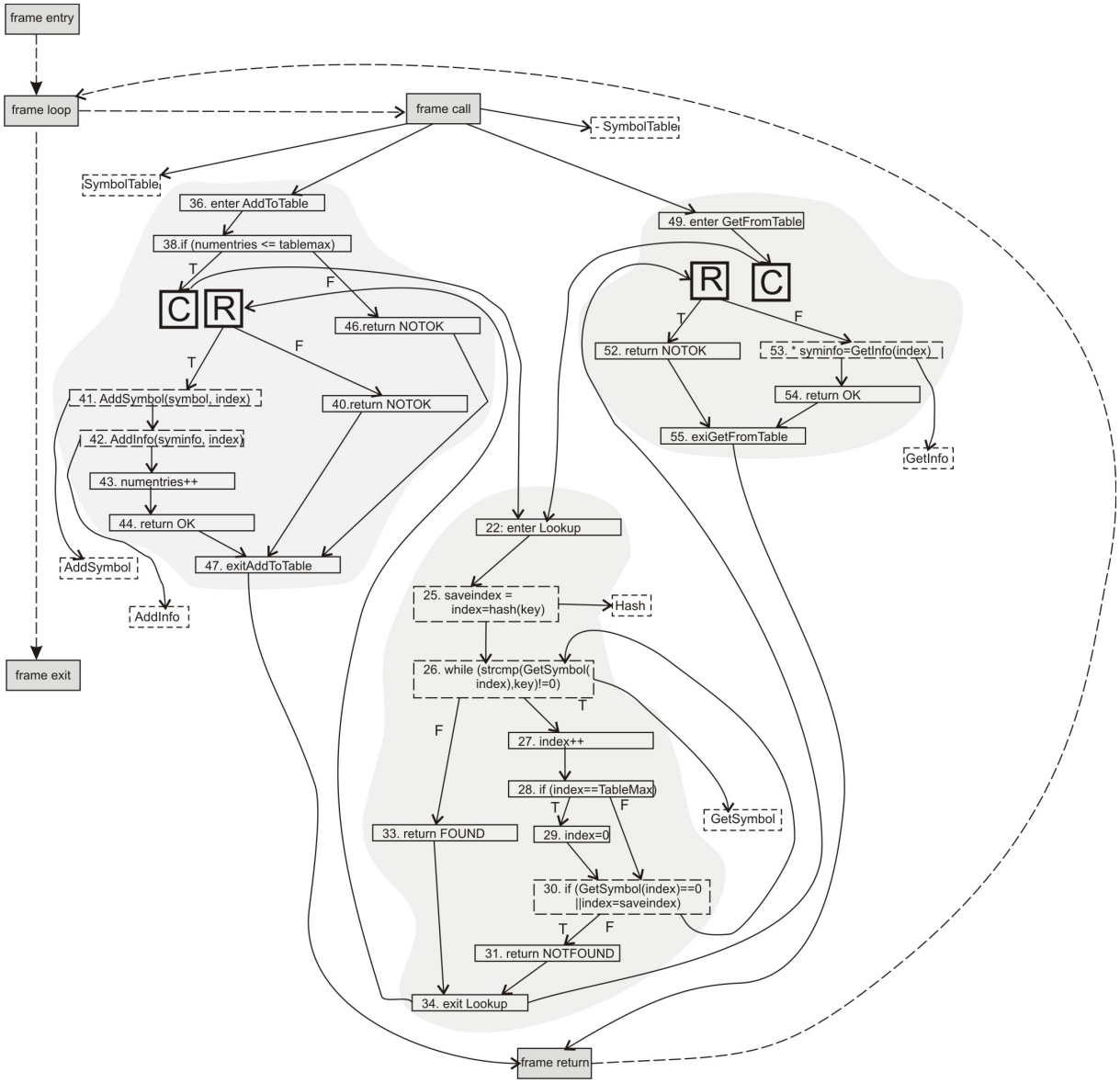

Figura 4.3: Grafo de fluxo de controle de classe para a classe `SymbolTable` (Harrold e Rothermel, 1994)

é definida por sentenças da linguagem); e as arestas de exceção, que representam o fluxo de controle quando uma exceção é lançada.

A Figura 4.4 exemplifica essa abordagem. O código fonte de um método simples `dummy` e o seu respectivo *bytecode* são mostrados nas Figuras 4.4(a) e 4.4(b), respectivamente. O grafo $\mathcal{IG}$ correspondente ao conjunto de instruções de *bytecode* da Figura 4.4(b) é apresentado na Figura 4.4(c), com seus respectivos conjuntos de variáveis definidas (*def(i)*) e usadas (*uso(i)*) em cada nó. A Figura 4.4(d) mostra o grafo $\mathcal{DU}$ correspondente ao grafo $\mathcal{IG}$ da Figura 4.4(c).

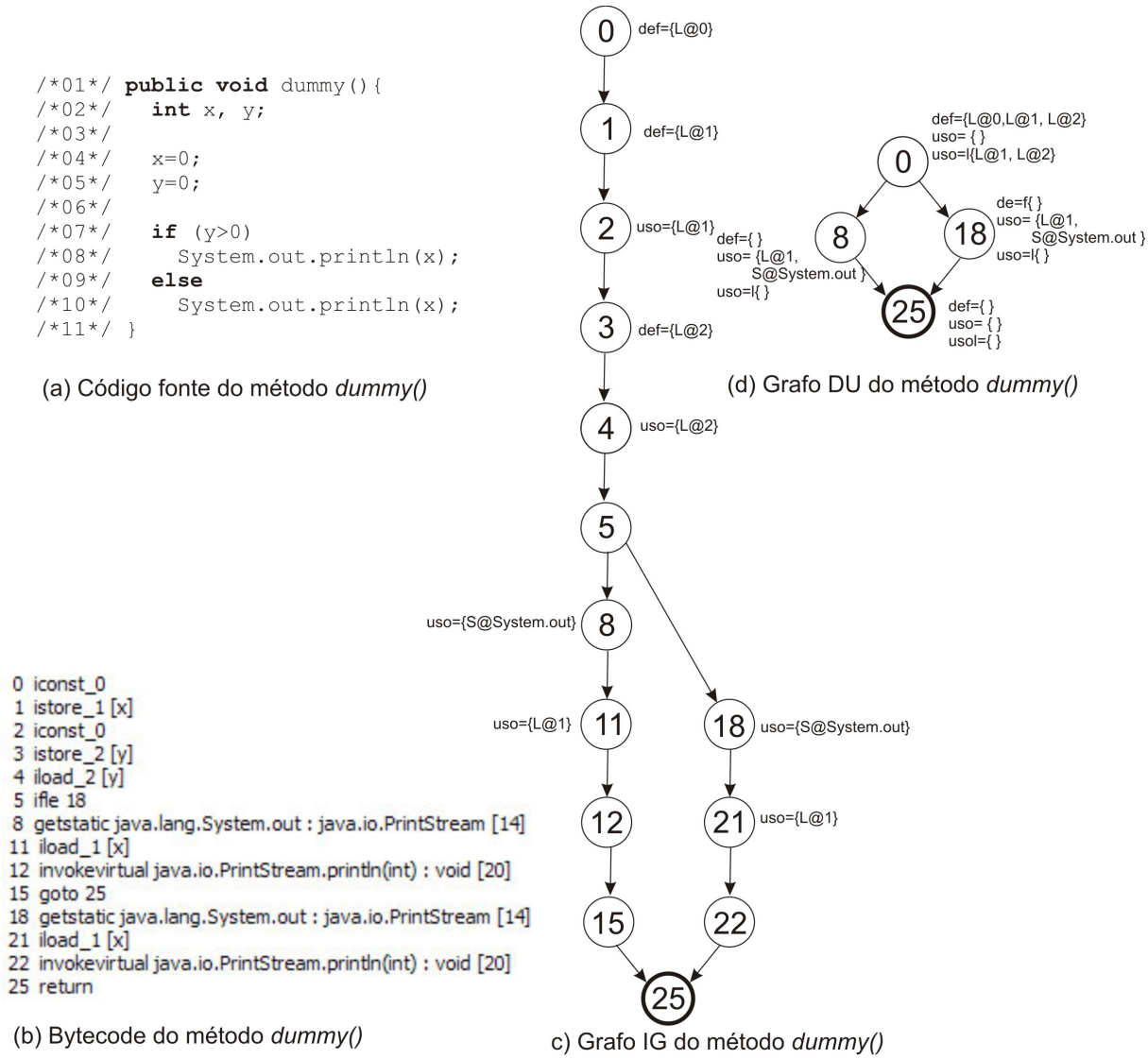

Figura 4.4: Exemplo dos grafos $\mathcal{IG}$ e $\mathcal{DU}$, adaptados de Vincenzi (2004)

A partir do grafo $\mathcal{DU}$, diferentes critérios para teste estrutural intramétodo podem ser utilizados para derivar requisitos de teste. Na definição de tais critérios, alguns conceitos adicionais são requeridos:

Nós predicativos: o conjunto de nós predicativos é o conjunto de todos os nós do grafo $\mathcal{DU}$ que contêm mais de uma aresta regular de saída.

Caminhos livres de exceção: o conjunto de caminhos livres de exceção é o conjunto π | $\forall$ $(n_i, n_j) \in \pi \Rightarrow (n_i, n_j)$ alcançável por um caminho que não contém nenhuma aresta de exceção.

Nós dependentes e independentes de exceção: o conjunto de nós dependentes de exceção é definido como N_{e_d} = {$n \in N$ | não existe um caminho livre de exceção π tal que $n \in \pi$ }. O conjunto de nós independentes de exceção é o conjunto definido como N_{e_i} = N - N_{e_d}.

Arestas dependentes e independentes de exceção:as arestas dependentes de exceção formam o conjunto $E_{ed} = \{e \in E \mid$ não existe um caminho livre de exceção π tal que $e \in \pi$. As arestas independentes de exceção formam o conjunto E_{e_i} = *E* - E_{e_d}.

Seja *T* um conjunto de casos de teste para um programa *P* (sendo que $\mathcal{DU}$ é o grafo de fluxo de controle/dados de *P*), e seja Π o conjunto de caminhos exercitados por *T*. Diz-se que um nó *i* está incluído em Π se Π contém um caminho (n_1,..., n_m) tal que i_1 = n_j para algum j, $1 \leq j \leq m$. Similarmente, uma aresta (i_1, i_2) é incluída em Π se Π contém um caminho $(n_1, \cdots, n_m)$ tal que i_1=n_j e i_2 = n_{j+1} para algum *j*, 1≤ j ≤ m-1.

Os critérios de fluxo de controle e de dados definidos pelo pesquisador foram:

- **Todos-nós:**
  - **Todos-nós-independentes-de-exceção (*todos-nós*$_{e_i}$):** Π satisfaz o critério todos-nós-independentes-de-exceção se cada nó $n_{e_i} \in N_{e_i}$ está incluído em Π. Em outras palavras, requer que cada nó de um grafo $\mathcal{DU}$, que é alcançável por pelo menos um caminho livre de exceção, seja exercitado ao menos uma vez por algum caso de teste.
  - **Todos-nós-dependentes-de-exceção (*todos-nós*$_{e_d}$):** Π satisfaz o critério todos-nós-dependentes-de-exceção se cada nó $n_{e_d} \in N_{e_d}$ está incluído em Π. Em outras palavras, requer que cada nó de um grafo $\mathcal{DU}$, que não é alcançável por pelo menos um caminho livre de exceção, seja exercitado ao menos uma vez por algum caso de teste.
- **Todas-arestas:**
  - **Todas-arestas-independentes-de-exceção (*Todas-aresta*s_{e_i}):** Π satisfaz o critério todas-arestas-independentes-de-exceção se cada aresta $e_{e_i} \in E_{e_i}$ está incluída em Π. Em outras palavras, requer que cada aresta de um grafo $\mathcal{DU}$, que é alcançável por pelo menos um caminho livre de exceção, seja exercitada ao menos uma vez por algum caso de teste.
  - **Todas-arestas-dependentes-de-exceção (*Todas-aresta*s_{e_d}):** Π satisfaz o critério todas-arestas-dependentes-de-exceção se cada aresta $e_{e_d} \in E_{e_d}$ está incluída em Π. Em outras palavras, requer que cada aresta de um grafo $\mathcal{DU}$, que não é alcançável por pelo menos um caminho livre de exceção, seja exercitada ao menos uma vez por algum caso de teste.
- **Todos-usos:**

- **Todos-usos-independentes-de-exceção (*Todos-uso*s_{e_i})**: requer que seja exercitado pelo menos um caminho livre de exceção e livre de definição para uma variável definida em um nó *n* para todo nó e toda aresta que tem um uso da mesma variável e que possa ser alcançada a partir de *n*.
- **Todos-usos-dependentes-de-exceção (Todos-usos_{e_d}):** requer que seja exercitado pelo menos um caminho que não seja livre de exceção, mas livre de definição para uma variável definida em um nó *n* para todo nó e toda aresta que tem um uso da mesma variável e que possa ser alcançada a partir de *n*.

Segundo Vincenzi (2004), o modelo de fluxo de dados proposto, ainda que desenvolvido para trabalhar com *bytecode*, também pode ser aplicado para o código fonte com resultados não significantemente diferentes. Assim, os requisitos de teste e sua cobertura por um dado conjunto de teste T não serão diferentes se a técnica for aplicada ao código fonte.

4.3 Teste de Programas Orientados a Aspectos

Programas Orientados a Aspectos têm características específicas que diferem de programas desenvolvidos utilizando programação orientada a objetos e programação procedimental, tornando o processo de teste mais difícil de ser efetuado. Algumas dessas características, conforme Alexander et al. (2004):

- Os aspectos dependem do contexto de outras classes;
- As dependências de controle e de dados não são facilmente compreensíveis a partir do código fonte dos aspectos e das classes;
- As implementações dos aspectos são fortemente acopladas às classes as quais eles foram combinados;
- Um defeito pode estar na implementação da classe ou do aspecto, ou pode ser um efeito colateral de uma ordem particular de combinação de múltiplos aspectos.

Segundo Alexander et al. (2004), o teste sistemático de sistemas OA deve ser baseado em modelos de defeitos que refletem a estrutura e o comportamento característicos dessa técnica; assim, os critérios e as estratégias para testar programas OA devem ser desenvolvidos em termos desse modelo. Assim, ele define um modelo de defeitos para programas orientados a aspectos com os seguintes tipos de defeitos:

1. **Restrição incorreta em padrões de conjuntos de junção:** as restrições dos padrões da assinatura dos conjuntos de junção determinam quais pontos de junção serão selecionados. Se o padrão for muito forte, alguns pontos de junção necessários não serão selecionados. Se o padrão for muito fraco, os pontos de junção, que deveriam ser ignorados, serão selecionados. Esses defeitos podem causar mais falhas nos aspectos que nas classes. Assim, um critério para revelar esse defeito é requerer o teste dos aspectos.

2. **Precedência incorreta de aspectos:** a ordem na qual os adendos de múltiplos aspectos são combinados em um interesse afeta o comportamento do sistema, especialmente quando há interações mútuas entre aspectos por meio de variáveis de estado na classe. Esses defeitos ocorrem quando múltiplos aspectos interagem e são afetados pela ordem de combinação; assim, um critério para revelar esses defeitos é testar todas as ordens de combinação.

3. **Defeito na preservação de pós-condições impostas:** os aspectos podem causar mudanças no fluxo de controle do código da classe. No entanto, é esperado que as pós-condições sejam satisfeitas, independentemente se há ou não aspectos interferindo no comportamento do método. Os defeitos levam à falha das classes e, assim, um critério razoável é retestar todos os métodos interceptados por aspectos, usando o conjunto de casos de testes original.

4. **Defeito na preservação de invariantes de estado:** o comportamento de um interesse é definido em termos de uma representação física de seu estado e dos métodos que atuam naquele estado. Dessa forma, para estabelecer suas pós-condições, os métodos devem garantir que os invariantes de estado sejam satisfeitas e que a combinação não acarrete violações. Esses defeitos também podem causar falhas nos métodos das classes, de modo que um critério apropriado para revela-los é retestar os métodos de interesse.

5. **Foco incorreto do fluxo de controle:** há casos em que a informação necessária para tomar determinadas decisões está disponível apenas em tempo de execução. Isso ocorre quando os pontos de execução devem ser selecionados dentro de um contexto de execução particular, por exemplo, dentro da estrutura de controle de um determinado objeto. Assim, pode ser que pontos de junção sejam negligenciados ou tomados por engano. Esses defeitos podem causar erroneamente a ativação de adendos; assim, um critério para revelar esses defeitos pode ser uma forma de cobertura de condição dos designadores de conjuntos de junção.

6. **Mudanças incorretas no controle de dependências:** mudanças errôneas nas dependências de fluxo de controle podem ocorrer a partir do mau uso de aspectos. Por exemplo,

o adendo `around` de AspectJ, que executa no lugar do ponto de junção, pode alterar o fluxo de controle e as dependências de dados de um método. Esses defeitos afetarão o comportamento da classe, assim como os tipos de defeito 3 e 4.

Ainda que muitas abordagens tenham sido propostas para softwares procedimentais e orientados a objetos, elas não podem ser aplicados para o software orientado a aspectos. Assim, tornam-se necessárias novas estratégias e ferramentas de teste que dêem suporte a essa técnica. A seção a seguir apresenta algumas abordagens.

4.3.1 Abordagem de Teste Estrutural Proposta por Zhao (2002, 2003)

O primeiro pesquisador a apresentar uma abordagem para teste estrutural de programas orientados a aspectos foi Zhao (2002, 2003). Em seu trabalho, que considera a classe e o aspecto como as menores unidades de um programa OA, é proposta uma abordagem para teste de unidade baseada em fluxo de dados. Segundo ele, para testar corretamente as classes e os aspectos deve-se, primeiramente, testar os aspectos junto com os métodos cujos comportamentos podem sem afetados por um adendo (perspectiva do aspecto) e, depois, testar uma classe junto com os adendos que podem afetar seu comportamento (perspectivas das classes). Sendo assim, ele define os seguintes conceitos:

- **Aspecto combinado (c-aspecto):** é um aspecto individual junto com alguns métodos pertencentes a uma ou mais classes, de modo que o comportamento dos métodos pode ser afetado pelos adendos.
- **Classe combinada (c-classe):** é uma classe individual junto com alguns adendos e pertencentes a um ou mais aspectos, tal que o adendo pode afetar o comportamento dos métodos da classe.
- **Método combinado (c-método):** é um método individual junto com um ou mais adendos que podem afetar o comportamento dos métodos.
- **Classe normal (n-classe):** é uma classe individual cujo comportamento nunca será afetado por nenhum aspecto.
- **Método normal (n-método):** é um método individual cujo comportamento nunca será afetado por nenhum adendo.

Para Zhao (2002, 2003), um módulo pode ser um adendo, método, construtor, método simples, construtor simples ou um método introduzido. Ele é uma parte da unidade em teste e não a unidade em si. Assim, baseado em Harrold e Rothermel (1994), o pesquisador considera três níveis diferentes de teste: teste **intramódulo** (executa o teste em um módulo individual em uma classe ou aspecto), teste **intermódulo** (executa o teste em um módulo público junto com os módulos que ele chama, direta ou indiretamente, de um aspecto ou classe) e teste **intra-aspecto/classe** (executa testes nas interações de múltiplos módulos públicos de um aspecto ou de uma classe, quando eles são executados em diferentes sequências).

Para cada um desses níveis de teste, (Zhao, 2002, 2003) define os seus pares def-uso correspondentes, que são os pares def-uso intramódulo, pares def-uso intermódulo e pares def-uso intra-aspecto/classe, respectivamente. A seleção dos testes para aspectos ou classes é baseada na computação desses pares def-uso. Para computá-los, o pesquisador utiliza os seguintes grafos:

- **Grafo de Fluxo de Controle:** o GFC é utilizado como uma base para obtenção da informação de fluxo de dados em um módulo individual (isto é, um c-método, um n-método ou um adendo) de um c-aspecto ou c-classe.

- **Grafo de Fluxo de Controle Interprocedimental (ICFG):** o ICFG é utilizado como uma base para obtenção da informação do fluxo de dados que envolvem mais que um módulo em um c-aspecto ou c-classe. Esse grafo é composto por um grafo de chamadas — um grafo direcionado em que os vértices representam módulos e os arcos, as chamadas possíveis entre esses módulos — e de um grupo de GFCs no qual cada um representa um módulo no c-aspecto ou c-classe.

- **Grafo de Fluxo de Controle com *Frame*:** o grafo de fluxo de controle com *frame* (FCFG), definido pelo pesquisador, é utilizado como base para derivar as informações de fluxo de dados em um c-aspecto ou c-classe. O FCFG consiste de uma coleção de CFGs, em que cada um está presente em um módulo no c-aspect ou c-classe e algumas arestas adicionais utilizadas para construir o *frame*. Em um FCFG, alguns vértices são utilizados para representar o *frame*, tais como o *frame* de entrada, *loop frame*, *frame* de chamada, *frame* de retorno e *frame* de saída. Na Figura 4.6 são mostrados os FCFGs do c-aspecto `PointShadowProtocol` e da c-classe `Point`, apresentadas na Figura 4.5.

Essa abordagem não utiliza o *bytecode*, pois na época em que essa pesquisa foi realizada, o compilador de AspectJ ainda estava em sua primeira versão (em que a combinação era feita no código-fonte)

```
/*ce0*/  public class Point {
/*s1*/     protected int x,y;
           public Shadow Shadow;
/*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;
           }
/*me11*/   public void setY(int _y){
/*s12*/      y = _y;
           }
/*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*/       this.x = x;
/*s24*/       this.y = y;
            }
          }
```

```
/*ase27*/  public 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(void Point.setX(int));
/*pe38*/     pointcut settingY(Point p):
               target(p) && call(void Point.setY(int));

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

Figura 4.5: Código exemplo para o teste de POA proposto por Zhao (2003).

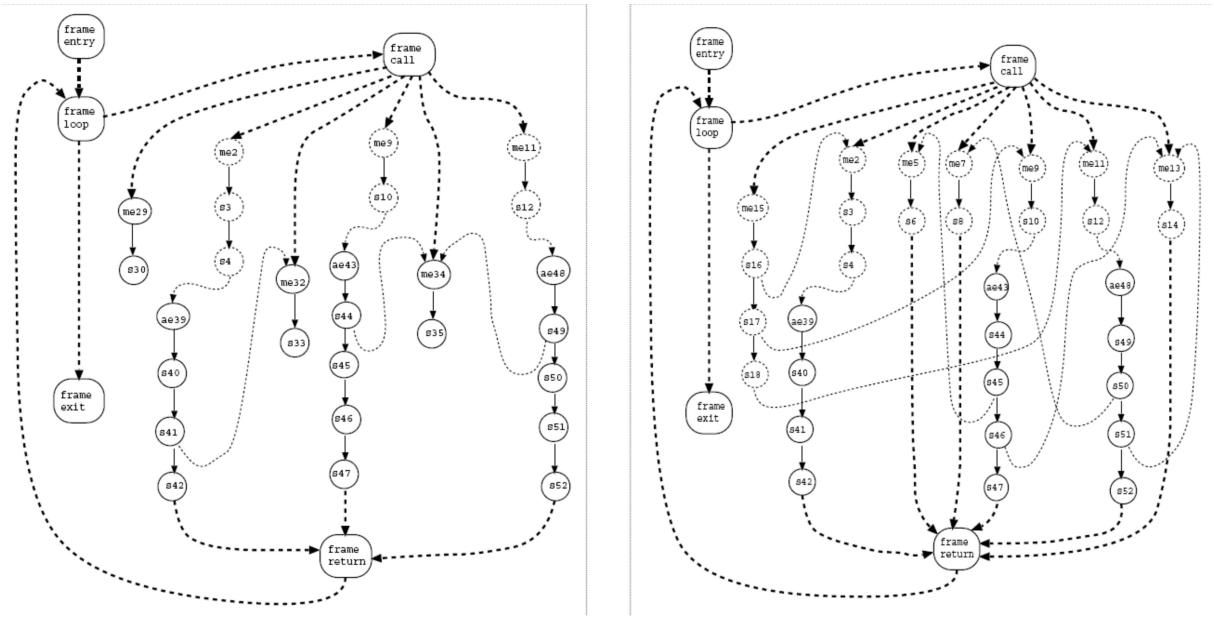

Figura 4.6: FCFG para o c-aspecto `PointShadowProtocol` e para a c-classe `Point`

4.3.2 Abordagem de Teste Estrutural Proposta por Lemos (2005)

Outra abordagem existente é a de Lemos (2005). O pesquisador propõe uma abordagem para teste estrutural de unidade de controle e fluxo de dados para programas orientados a aspectos. Ele considera o método como a menor unidade de teste e, então, divide a atividade de teste OA nas seguintes fases:

- O teste de unidade de programas OA como o teste de cada método (intramétodo) e cada adendo (intra-adendo);
- O teste de módulo como o teste de uma coleção de unidades dependentes, isto é, unidades que interagem por meio de chamadas ou interações com adendos, que pode ser dividida entre os seguintes tipos de testes: intermétodo, adendo-método, método-adendo, inter-adendo, intermétodo-adendo, intraclasse e interclasse.
- Teste de sistema como o teste de integração entre todos os módulos do sistema ou do subsistema.

O pesquisador propõe um grafo de fluxo de dados ($\mathcal{AODU}$ — Grafo Def-Uso Orientado a Aspectos) como um modelo do qual os requisitos de teste de controle e de fluxo de dados são derivados. O grafo é construído para cada método da construção do AspectJ (métodos e construtores regulares, adendos e métodos e construtores declarações inter tipos). Antes de construir o grafo $\mathcal{AODU}$, seguindo o trabalho de Vincenzi (2004), constrói-se o Grafo de Instruções. As informações de fluxo de dados sobre as definições e os usos das variáveis são derivadas a partir do *bytecode* e adicionadas para cada nó ou aresta. Pelo fato de o número de nós e arestas envolvidas no grafo $\mathcal{IG}$ poder ser muito grande para tratar, o grafo $\mathcal{AODU}$ é construído com base nele, mas utilizando o conceito de bloco de instruções. Um grafo $\mathcal{AODU}$ de uma dada uma unidade *u* é definido como um grafo dirigido $\mathcal{AODU}$ (*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-upla ordenada de instruções agrupadas no nó *n*;
- $E = E_r \bigcup E_e$ 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 que parte do último elemento de I_{n_i} ao primeiro elemento de I_{n_j} no $\mathcal{IG}$ (u)}

- $s \in N|IN(s) = 0$ é o nó de entrada de u;
- $T \subseteq N$ é o conjunto (possivelmente vazio) de nós de saída. Isto é T = {$n \in N$ | $OUT(n) = 0$}.
- $C \subseteq N$ 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.

Na Figura 4.7 é mostrado um exemplo de programa escrito em AspectJ. O grafo $\mathcal{AODU}$ do método `affectedMethod` com a presença do aspecto é apresentado na Figura 4.8.

```
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: Programa escrito em AspectJ utilizada por Lemos (2005).

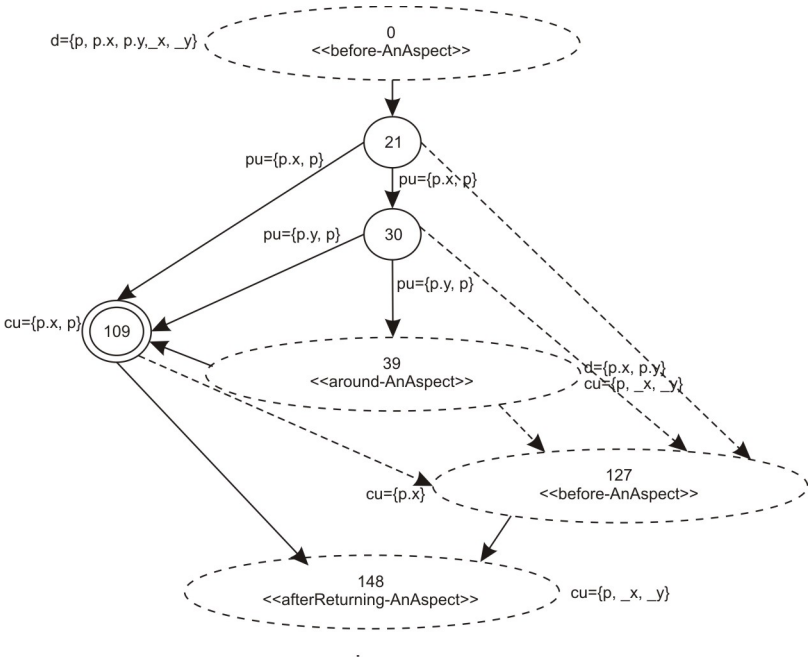

Figura 4.8: Grafo $\mathcal{AODU}$ referente ao código da Figura 4.7

Na representação gráfica do grafo $\mathcal{AODU}$ há os conjuntos cu, pu e d, que correspondem aos c-usos, p-usos e definições de variáveis; os nós regulares são representados por círculos desenhados com linha simples; nós de chamada são representados por círculos desenhados com linhas duplas; nós de saída são representados por círculos desenhados com linhas simples negritadas. Estes três tipos de nós contêm, em seus rótulos, a primeira instrução de *bytecode* do bloco. Nós transversais, representados por círculos desenhados com linhas duplas, contêm, em seus nós, além da primeira instrução do *bytecode* do bloco, o tipo de adendo que afeta aquele ponto (`before`, `after` ou `around`) e o aspecto a que o adendo pertence. Arestas regulares são representadas por linhas contínuas, mostrando o fluxo de controle normal; arestas de exceção são representadas por linhas tracejadas, mostrando o fluxo de controle do nó em que uma exceção é gerada até o primeiro nó correspondente ao tratador daquela exceção.

O pesquisador adaptou os critérios de fluxo de controle todos-nós, todas-arestas, todos-nós-independentes-de-exceção, todas-arestas-independentes-de-exceção e todas-arestas-dependentes-de-exceção propostas por Vincenzi (2004) para serem aplicados no contexto de programas orientados a aspectos. Considerando que nós transversais representam blocos de instruções em que são identificadas invocações implícitas dos adendos, foram definidos mais dois critérios:

- **Todos-nós-transversais (*Todos-nós*$_c$):** Π satisfaz o critério todos-nós-transversais se cada nó $n_i \in C$ é incluída em Π. Em outras palavras, esse critério requer que cada

nó transversal do grafo $\mathcal{AODU}$ seja exercitado pelo menos uma vez por algum caso de teste *T*.

- **Todas-arestas-transversais (Todas-arestas_c):** Π satisfaz o critério todas-arestas-transversais se cada aresta $e_c \in E_c$ é incluída em Π. Em outras palavras, esse critério requer que cada aresta do grafo $\mathcal{AODU}$ que tem um nó transversal como nó de início ou destino, seja executada pelo menos uma vez por algum caso de teste *T*.

Seguindo a mesma idéia, Lemos (2005) adaptou os critérios de fluxo de dados todos-usos, todos-usos-independentes-de-exceção e todos-usos-dependentes-de-exceção. Além disso, definiu um novo critério:

- **Todos-usos-transversais (*Todos-uso*s_c):** π satisfaz o critério todos-usos-transversais se, para todos os nós $i \in def(i)$, π inclui um caminho livre de definição em relação a *x* de *i* para cada elemento de dcu(*x*,*i*), que é um nó transversal e para cada elemento de dpu(*x*,*i*) no qual o nó de início da aresta é um nó transversal. Em outras palavras, esse critério requer que cada par def-c-uso $((i, j, x), j) \in dcu(x, i)$ e cada par def-p-uso $(i, (j, k), x), (j, k) \in dpu(x, i)$ tal que $j \in C$ seja executado pelo menos uma vez por algum caso de teste *T*.

Lemos (2005) também propôs uma abordagem para teste de integração de POA com enfoque no teste do tipo método-adendo. Ele estendeu as definições para o teste de unidade de sua abordagem e utilizou um modelo base para o teste de fluxo de dados (o grafo $\mathcal{MADU}$ (*Method-Advice Def-Use*)) de integração, composto pelo $\mathcal{AODU}$ do método em si, juntamente com os $\mathcal{AODU}$'s dos adendos que o afetam diretamente. Essa abordagem não foi implementada e serviu de inspiração para a proposta descrita a seguir.

4.3.3 Abordagem para Teste Estrutural de Integração Par-a-Par Proposta por Franchin (2007)

O objetivo do trabalho de Franchin (2007) foi estender a abordagem de teste de unidade de Lemos (2005, 2007) propondo, dessa forma, uma abordagem de teste estrutural de integração par-a-par para programas orientados a objetos e orientados a aspectos. O modelo estrutural proposto é chamado de $\mathcal{PWDU}$ e representa a estrutura de pares de unidades. Para programas OO há apenas um tipo de par de interação de unidades: método-método. Para programas OA, por outro lado, há quatro tipos de pares de interação entre unidades: método-método,

método-adendo (quando um método é afetado por um adendo), adendo-método (quando um adendo chama outro método) e adendo-adendo (quando um adendo é afetado por outro adendo).

Nessa abordagem também foi necessário revisar e adaptar os modelos e critérios de fluxo de controle e de fluxo de dados propostos anteriormente. A definição do grafo $\mathcal{AODU}$ foi alterada para $\mathcal{AODU}\ (u) = (N, E, s, T, I)$ para tratar, além das interações com adendos, as interações com métodos. Dessa forma, o componente C (ver Seção 4.3.2) foi substituído pelo componente I, definido como um conjunto de nós de interação, isto é, um conjunto que contém tanto os nós transversais — que representam um nó no qual ocorre uma interação com um adendo de um dado aspecto — quanto os nós de chamada — que representam um nó no qual ocorre uma interação com um método de uma dada classe ou aspecto.

Para representar adequadamente o fluxo de execução dos dados que ocorre dentro de um par de unidades, definiu-se o grafo $\mathcal{PWDU}$ (*PairWise Def-Use*), que estende o grafo $\mathcal{AODU}$ com o objetivo de criar um grafo integrando a unidade base — a unidade que está chamando um método ou sendo afetada por um adendo — com a unidade chamada — a unidade em que o fluxo de controle é passado. Para diferenciar nós e arestas das unidades, foram definidos os nós integrados que representam os nós da unidade chamada e dois tipos de arestas: as arestas integradas que conectam dois nós integrados, e as arestas de integração, que representam o fluxo de controle entre um nó da unidade base e um nó da unidade integrada, e vice-versa.

A definição formal do grafo $\mathcal{PWDU}$ é a seguinte: considere o grafo $\mathcal{AODU}$ das unidades u_1 e u_2 da forma:

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

$\mathcal{AODU}\ (u_2) = (N_2, E_2, s_2, T_2, I_2)$

O grafo $\mathcal{PWDU}$ do par u_1 e u_2 é definido como um grafo dirigido $\mathcal{PWDU}\ (u_1, u_2) = (N, E, s, T, I, i, R)$, tal que:

- $N = N_1 \bigcup N_2$ representa o conjunto completo de nós do grafo $\mathcal{PWDU}$, tal que:
  - N_1 é o conjunto de nós do $\mathcal{AODU}$ de u_1;
  - N_2 é o conjunto de nós do $\mathcal{AODU}$ de u_2, também denominado conjunto de nós integrados — N_i;
- $E = E_1 \bigcup E_2 \bigcup E_I - e_d$ é o conjunto completo de arestas do grafo $\mathcal{PWDU}$, tal que:
  - $E_1 \subseteq N_1 X N_1$ é o conjunto de arestas de u_1.
  - $E_2 \subseteq N_2 X N_2$ é o conjunto de arestas de u_2, também denominado conjunto de arestas integradas — E_i.

  - E_I é o conjunto de arestas de integração, criadas para integrar os dois grafos $\mathcal{AODU}$.
  - e_d é a aresta original que ligava o nó onde ocorreu a integração ao nó subsequente e que foi removida.

- $s \in N$ e $s = s_1$ é o nó de entrada do grafo $\mathcal{PWDU}$, tal que $s_1 \in N_1$ é o nó de entrada de u_1.

- $T \subseteq N$ e $T = T_1$ é o conjunto de nós de saída do grafo $\mathcal{PWDU}$, tal que T_1 é o conjunto de nós de saída de u_1.

- $I = I_1 \bigcup I_2$ é o conjunto completo de nós de interação (ou seja, nós transversais e nós de chamada) do grafo $\mathcal{PWDU}$, tal que:

  - $I_1 \subseteq N_1$ é o conjunto de interação de u_1;
  - $I_2 \subseteq N_2$ é o conjunto de interação de u_2;

- $i \in I_1$ é o nó onde ocorre a chamada a u_2;

- $R \subseteq N_1$ é o conjunto de nós de retorno da chamada a u_2;

Na Figura 4.9 é apresentado como exemplo o código fonte de uma aplicação de soma e subtração utilizado por Franchin (2007). Na Figura 4.10(a) é mostrado o grafo $\mathcal{AODU}$ do método `doCalculation`. Na Figura 4.10(b) é mostrado o grafo $\mathcal{AODU}$ do método `calculate`. Na Figura 4.11 mostra-se o grafo $\mathcal{PWDU}$ que integra os dois grafos da Figura 4.10 e acrescenta as informações de fluxo de dados.

Na representação gráfica do grafo $\mathcal{AODU}$ há os conjuntos cu, pu e d que correspondem aos c-usos, p-usos e definições de variáveis; os nós regulares são representados por círculos desenhados com linha simples; nós de chamada são representados por círculos desenhados com linhas duplas; nós de saída são representados por círculos desenhados com linhas simples negritadas. Esses três tipos de nós contêm, em seus rótulos, o número da primeira instrução de *bytecode* do bloco. Nós transversais, que são representados por círculos desenhados com linhas duplas, contêm, em seus nós, além da primeira instrução do *bytecode* do bloco, o tipo de adendo que afeta aquele ponto (`before`, `after` ou `around`) e o aspecto que o adendo pertence. Nós integrados são representados como nós regulares, nós de chamada, nós transversais ou nós de saída. Como diferencial, seus rótulos são iniciados com o símbolo "i.". Arestas regulares são representadas por linhas contínuas, mostrando o fluxo de controle normal; arestas de exceção são representadas por linhas tracejadas, mostrando o fluxo do controle do nó em que uma exceção é gerada, até o primeiro nó correspondente ao tratador daquela exceção; arestas

```
public aspect Logging{
  pointcut loggedOp(int a1, int a2):
    execution(* src.Calculus.calculate(..))
      && args(a1, a2);

  before(int a1, int a2) :loggedOp(a1, a2){
    System.out.println("Numbers: " +
      a1 + " and " + a2);
  }
}

public class Calculus{
  public int resSum;
  public int resSub;

  public void calculate(int p1, int p2){
    this.resSum = p1 + p2;
    if (p1 > p2)
      this.resSub = p1 - p2;
    else
      this.resSub = p2 - p1;
  }
}

public class Main{
  public static void doCalculation(
    int d1, int d2){
    int num1 = d1;
    int num2 = d2;
    Calculus calc = new Calculus();
    calc.calculate(num1, num2);
    System.out.println("Sum = " +
      calc.resSum);
    System.out.println("Subtraction = " +
      calc.resSub);
  }
}
```

Figura 4.9: Exemplo de código fonte de uma aplicação para soma e subtração utilizado por Franchin (2007).

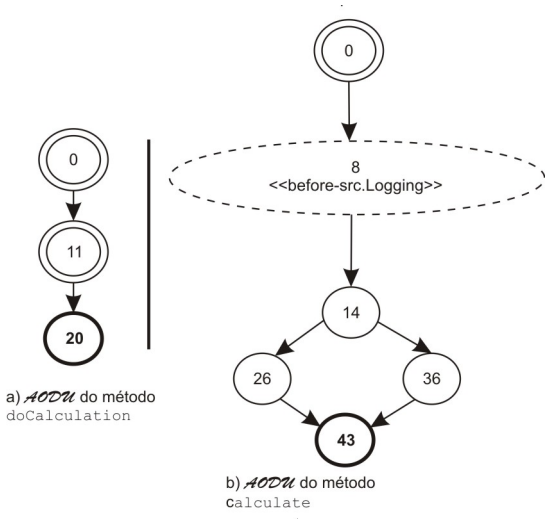

Figura 4.10: Grafos $\mathcal{AODU}$ dos métodos `doCalculation` e `calculate`

integradas são representadas como as regulares ou arestas de exceção; arestas de integração são representadas como as regulares.

A partir desse grafo foram definidos os seguintes critérios de teste de fluxo de controle:

- **Todos-nós-integrados-par-a-par (*Todos-PW-nó*s_i):** Π satisfaz o critério todos-nós-integrados-par-a-par se, cada nó integrado $n_i \in N_i$ de um grafo $\mathcal{PWDU}$ está incluído em Π. Em outras palavras, esse critério requer que cada nó integrado de um grafo $\mathcal{PWDU}$ seja exercitado ao menos uma vez por algum caso de teste T.

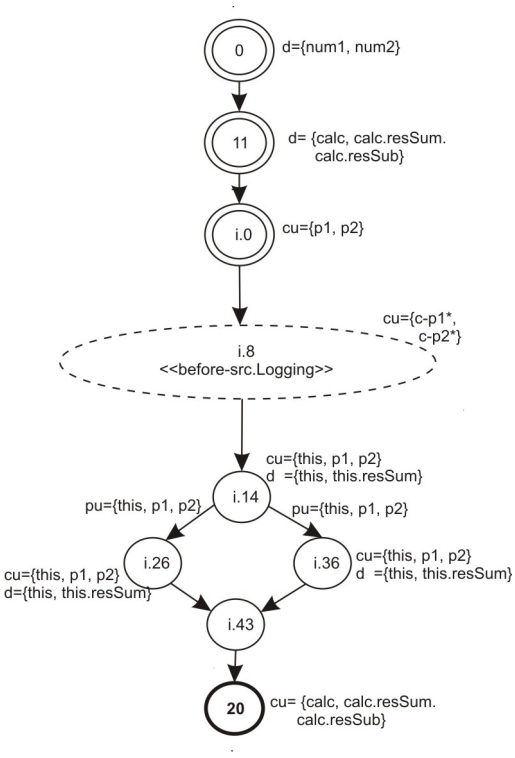

Figura 4.11: Grafo $\mathcal{PWDU}$ do exemplo da Figura 4.9

- **Todas-arestas-integradas-par-a-par (*Todas-PW-Aresta*s_i):** Π satisfaz o critério todas-arestas-integradas-par-a-par se, cada aresta integrada $e_i \in E_i$ de um grafo $\mathcal{PWDU}$ está incluído em Π. Em outras palavras, esse critério requer que cada aresta integrada de um grafo $\mathcal{PWDU}$ seja exercitado ao menos uma vez por algum caso de teste T.

Da mesma forma, foi definido o seguinte critério de fluxo de dados:

- **Todos-usos-integrados-par-a-par (*Todos-PW-uso*s_i):** Π satisfaz o critério todos-usos-integrados-par-a-par se 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 na unidade chamada.

  – a execução de um caminho livre de definição com relação a cada variável de comunicação a partir de cada definição relevante na unidade chamada até todo uso computacional e todo uso predicativo na unidade chamadora.

4.3.4 Abordagem para Teste Estrutural de Integração Baseada em Conjuntos de Junção

Considerando que um defeito pode estar relacionado com a execução de uma parte particular do adendo em um ponto de junção específico, Lemos (2009) propôs uma abordagem de teste de integração estrutural baseada no mecanismos de conjuntos de junção para AspectJ, para que se possa garantir a cobertura da estrutura do adendo em cada ponto de junção que ele pode atuar e, dessa forma, aumentando 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 $\mathcal{AODU}$'s das unidades afetadas pelo adendo e o $\mathcal{AODU}$ do adendo repetido em cada possível ponto de junção. A repetição do $\mathcal{AODU}$ do adendo deve-se ao fato de querer rastrear sua execução em cada ponto de junção. Os nós do $\mathcal{PCDU}$ são rotulados com duas cadeias de caracteres: um prefixo e um sufixo. O sufixo sempre corresponde ao deslocamento da primeira instrução *bytecode* do bloco correspondente, como nos $\mathcal{AODU}$'s. O prefixo é: (1) uma letra maiúscula, se o nó pertence a um $\mathcal{AODU}$ base; ou (2) o número do ponto de junção correspondente, se o nó pertence à estrutura do adendo. Uma letra é designada para cada unidade base em cada ponto de junção e um número é designado para cada ponto de junção.

Baseado no $\mathcal{PCDU}$, Lemos (2009) definiu dois critérios para fluxo de controle e um critério para fluxo de dados, descritos a seguir:

- **Todos-nós-baseados-em-conjunto-de-junção:** Requer que cada nó do $\mathcal{PCDU}$ que tem um número como prefixo em seu rótulo seja exercitado ao menos uma vez por algum caso de teste.
- **Todas-arestas-baseadas-em-conjunto-de-junção:** Requer que cada aresta do $\mathcal{PCDU}$ que tem um número como prefixo nos rótulos de seu nó de origem e de seu nó destino seja exercitado pelo menos uma vez por algum caso de teste.
- **Todos-usos-baseados-em-conjunto-de-junção:** Requer que cada par def-uso da unidade afetada ou do adendo seja exercitado pelo menos uma vez por algum caso de teste, sendo que a definição ocorre na unidade afetada e o uso no adendo ou a definição ocorre no adendo e o uso na unidade afetada.

Para ilustrar um exemplo do grafo $\mathcal{PCDU}$, considere o código-fonte da classe (`Song`) e do aspecto (`Billing`) apresentado na Figura 4.12. Na Figura 4.13 é apresentado o grafo $\mathcal{PCDU}$

, que representa o adendo `bill` e o correspondente ponto de junção do aspecto `Billing`, integrado com todos os pontos de junção afetados. Note que o adendo afeta quatro pontos de junção nos seguintes métodos da classe `Song`: `play`, `showLirics`, `showCompose` e `showRelated`.

A abordagem foi implementada na ferramenta JaBUTi/AJ, que passou a ser chamada de JaBUTi/PC-AJ.

```
public class Song implements Playable {
  private String name;
  private boolean bonus;
  private String composer;
  private ArrayList<Song> related = new ArrayList<Song>();

  public Song(String name, String composer, boolean bonus) {
    super();
    this.name = name;
    this.composer = composer;
    this.bonus = bonus;
  }

...

  public void play() {
    ...
  }

  public void showLyrics(){
    ...
  }

  public void showRelated() {
    if (!related.isEmpty()) {
      System.out.println("Related songs:");
      for(Iterator<Song> it=related.iterator(); it.hasNext();){
        System.out.println(((Song)it.next()).getName());
      }
    }
  }

  public void showComposer(){
    System.out.println("The composer of the song is "
        + getComposer());
  }
```

```
  public boolean equals(Object o){
    Song other = (Song) o;
    return this.name.equals(other.name);
  }

  public int hashCode() {
    return name.hashCode();
  }

  public boolean isBonus() {
    return bonus;
  }

  ...

}

public aspect Billing {
  public pointcut useTitle() :
    execution(* Song.play(..)) ||
    execution(* Song.show*(..));

  @AdviceName("bill")
  after(Song song) returning throws
    InsufficientBalanceException : useTitle() &&
      this(song) {
    if (song.isBonus())
      return;
    User user = (User)Session.instance().getValue("currentUser");
    int amount = song.getPrice();
    if(amount > user.getAccount().getBalance())
      throw new InsufficientBalanceException(
        "Insufficient available balance.");
    user.getAccount().debit(amount);
    System.out.println("Charge: " + user + " " + amount);
  }
}
```

Figura 4.12: Código fonte da classe `Song` e do aspecto `Billing`.

4.3.5 Outras Abordagens para Teste de POA

Xu et al. (2005) propuseram uma abordagem mista de teste, baseada em estados e estrutural para programas orientados a aspectos (porém com ênfase na técnica baseada em estados). A idéia principal é estender o modelo baseado em estados FREE (*Flattened Regular Expression*) proposto por Binder (1999), para um modelo de estados aspectual (*Aspectual State Model — ASM*). O modelo FREE representa os estados e comportamentos dinâmicos de objetos e oferece diretrizes para implementação. A partir desse modelo é derivada uma árvore de transições a partir da qual são criados casos de teste que definem sequências de chamadas a métodos. A segunda parte da abordagem mistura a técnica estrutural à técnica baseada em estados. A idéia

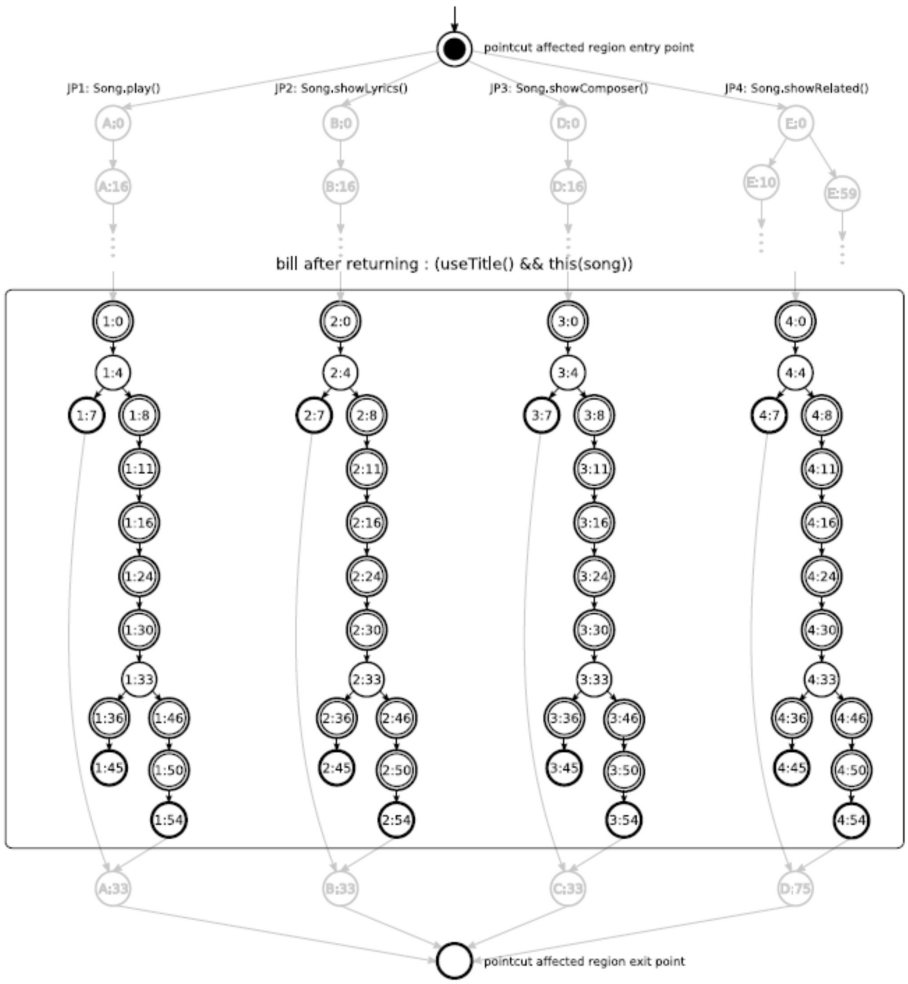

Figura 4.13: $\mathcal{PCDU}$ para o adendo `bill` e ponto de junção `useTitle`

é substituir as transições e interações de aspectos por grafos de fluxo de controle referentes aos métodos e adendos. O modelo é chamado de grafo de fluxo de aspectos (*Aspect Flow Graph* — AFG). Com esse modelo podem ser realizadas análises de cobertura a partir da execução dos casos de teste gerados com auxílio da árvore de transição, para auxiliar na geração de novos casos de teste e para se ter idéia da suficiência da atividade de teste. Recentemente, esses pesquisadores propuseram uma abordagem para gerar casos de teste para programas OA.

Massicotte et al. (2007) apresentam uma abordagem de teste baseado em diagramas de comunicação. O foco está no teste da integração de um ou mais aspectos com um grupo de objetos que colaboram entre si. A abordagem é iterativa, e consiste de duas fases principais: (1) análise estática, que é a geração de sequências de testes baseadas nas interações dinâmicas entre aspectos e classes, e (2) análise dinâmica, que é a verificação da execução das sequências

selecionadas. Esquemas XML são utilizados para descrever o diagrama de comunicação e os aspectos relacionados. A estratégia segue um processo interativo, iniciando com o teste dos cenários de colaboração sem considerar os aspectos. Na segunda etapa, um a um os aspectos são analisados e o diagrama de comunicação é modificado para representar as novas sequências resultantes da combinação do aspecto. Para a geração dos casos de teste, uma árvore de mensagens é derivada do diagrama de comunicação, representando as possíveis sequências de operações. A partir dessa árvore, os casos de teste são gerados de acordo com os critérios definidos nesses trabalhos.

Ferrari et al. (2008) apresentam uma abordagem de teste de mutação para programas OA. Nesse trabalho é definido um conjunto de operadores de mutação para programas baseados em AspectJ. Esse conjunto foi identificado a partir de uma extensa revisão sobre teste de programas OA.

Alguns pesquisadores propuseram abordagens que testam os aspectos de forma isolada da perspectiva de tentar executar e testar adendos e mecanismos de quantificação sem a necessidade de combinar o aspecto com o resto do sistema. Entre essas abordagens está a de Lopes e Ngo (2005).

4.4 Estratégias de Ordenação de Classes e Aspectos

Um fator importante no teste de programas OO e OA é definir a ordem em que as classes e aspectos são testados, uma vez que isso influencia a ordem em que são desenvolvidos, a ordem em que os erros são encontrados e o número de *stubs* e drivers implementados. O problema da ordenação surge quando o sistema a ser testado é constituído de classes e aspectos que possuem ciclos de dependência. No contexto de estratégias de ordenação, entende-se por dependência a necessidade que um aspecto ou classe tem da existência de outros aspectos e classes para que seja possível efetuar os testes (Ré, 2009).

O trabalho de Kung et al. (1993) foi um dos primeiros a apresentar uma solução para o problema de ciclos de dependência no teste de unidade e no teste de integração de programas OO. Nesse trabalho foi proposto o ORD, que é um dígrafo no qual os vértices representam classes e as arestas os relacionamentos entre essas classes. No ORD são representados as dependências de herança (“I”), agregação (“Ag”) e associação (“As”). Briand et al. (2003) também propõe uma estratégia para programas OO. A estratégia inicia com a aplicação do algoritmo de Tarjan (1972) em um grafo construído a partir de um ORD. Deve-se, recursivamente, remover arestas dos SCCs (componentes fortemente conexos) não triviais (quando os SCCs são compostos por mais de um vértice) para quebrar os ciclos e aplicar o algoritmo de Tarjan (1972). A aresta

que deve ser removida é aquela que apresenta maior peso, obtido por meio da multiplicação das arestas de entrada do vértice origem pelas arestas de saída do vértice destino de cada aresta de associação. Esses passos devem ser seguidos até não existirem mais SCCs não triviais. A Figura 4.14 apresenta o exemplo de um ORD.

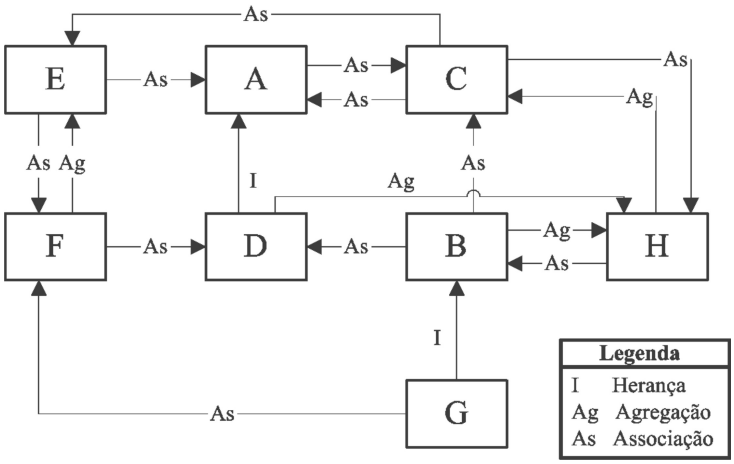

Figura 4.14: Exemplo do ORD estudado por Briand et al. (2003), retirado de Ré (2009)

Para determinar a ordem de programas orientados a aspectos, Ré definiu um modelo de tipos de dependência, que visa auxiliar o entendimento do relacionamento entre aspectos e classes e adaptou o ORD a partir desse modelo para representar adequadamente os novos tipos de dependência, chamando-o de AORD (Aspect and Object Relation Diagram). Os novos tipos de dependência encontrados em programas OA são declaração intertipos (“It”), entrecorte (“C”), de uso de conjunto de junção (“U”) e de relaxamento de exceção (“S”). Em seu trabalho, ele também apresenta duas estratégias que estendem a proposta de Briand et al. (2003) para ordenar aspectos e classes usando o AORD: a estratégia Incremental+ e a estratégia Conjunta. Na estratégia Incremental+ primeiramente as classes são testadas e integradas e posteriormente os aspectos são testados e integrados. Nessa estratégia são construídos dois AORDs, o primeiro apenas com as classes e o segundo apenas com os aspectos. Para construir o primeiro AORD devem ser usados apenas os mecanismos de conexão OO, enquanto que para construir o segundo são usados todos os mecanismos de conexão. O algoritmo de Briand é então aplicado no primeiro AORD e a ordem de teste e integração de classes é determinada. Posteriormente, o algoritmo de Briand é aplicado ao segundo AORD e a ordem de teste e integração entre aspectos é determinada. Nessa estratégia, as dependências entre classes e entre aspectos são consideradas separadamente. Isso faz com que as dependências que ocorrem no formato aspecto/classe e classe/aspecto sejam desconsideradas. O resultado é que, para cada dependência desconsiderada, um *stub* de aspecto deve ser implementado para quebrar o acoplamento, o que deve ser

computado no custo de uso da estratégia. As Figuras 4.15 e 4.16 apresentam um exemplo de AORD do mesmo programa. Na primeira, que é constituída apenas das classes, pode-se observar que existem classes isoladas que não se relacionam com as demais porque elas são utilizadas apenas pelos aspectos, a ordem de implementação e teste computada é mostrada na Tabela 4.2. Na Figura 4.16 o AORD é constituído somente de aspectos e a ordem de implementação e teste dos aspectos pode ser vista na Tabela 4.3.

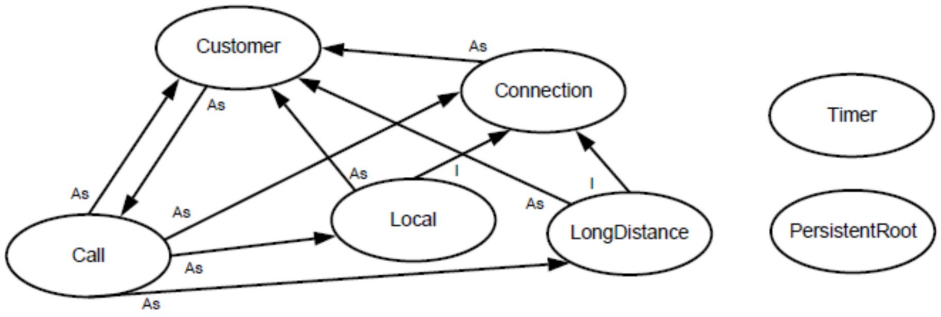

Figura 4.15: Exemplo do AORD somente com classes, retirado de Ré (2009)

Tabela 4.2: Ordem de implementação das classes para o AORD da Figura 4.15

Ordem	Módulo	Stub
1	Customer	Call
2	Connection	MyPersistentEntities
3	Local	
4	LongDistance	
5	Call	
6	PersistentRoot	
7	Timer	

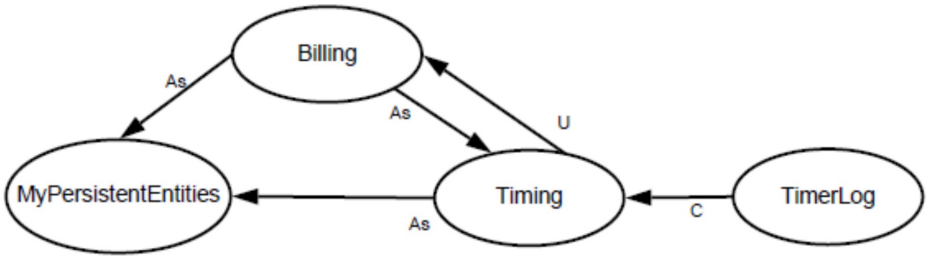

Figura 4.16: Exemplo do AORD somente com aspectos, retirado de Ré (2009)

A estratégia Conjunta não divide o AORD em dois subgrafos e, ao contrário da estratégia Incremental+, considera todas as dependências representadas. Isso faz com que classes e aspectos sejam testados e integrados conjuntamente, de acordo com a ordem determinada pela

Tabela 4.3: Ordem de implementação dos aspectos para o AORD da Figura 4.15

Ordem	Módulo	Stub
1	MyPersistentEntities	
2	Timing	Billing
3	Billing	
4	TimerLog	

estratégia. O AORD é construído em um único grafo e o algoritmo de Briand é aplicado, resultando na ordem de teste e integração de classes e aspectos. A Figura 4.17 mostra um exemplo do AORD com classes e aspectos e a Tabela 4.4 apresenta a ordem de implementação e teste calculada para esse AORD.

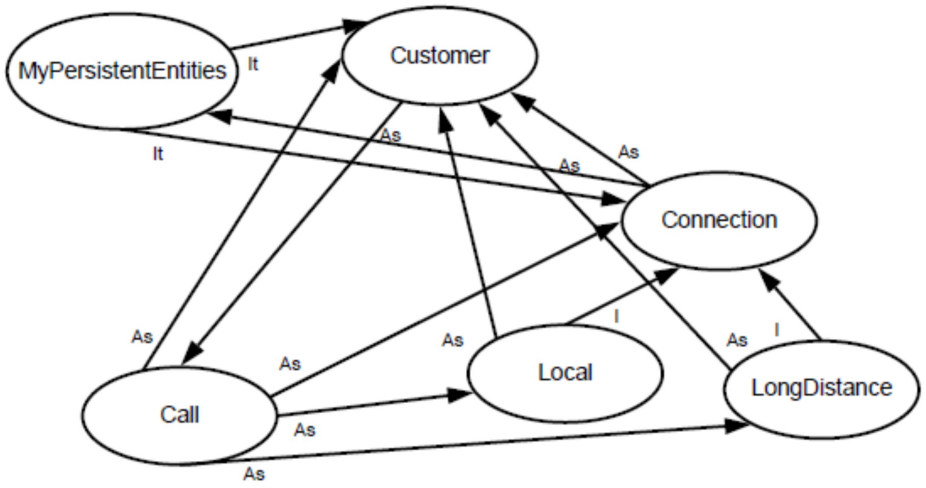

Figura 4.17: Exemplo do AORD com classes e aspectos, retirado de Ré (2009)

Tabela 4.4: Ordem de implementação dos aspectos para o AORD da Figura 4.15

Ordem	Módulo	Stub
1	PersistentRoot	
2	Customer	Call
3	Connection	MyPersistentEntities
4	MyPersistentEntities	
5	Local	
6	LongDistance	
7	Call	
8	Timer	
9	Timing	Billing
10	Billing	
11	TimerLog	

4.5 Ferramentas de Apoio a Teste de POO e POA

A atividade de teste propriamente dita é uma atividade muito propensa a erros, além de ser demorada se realizada manualmente. Dessa forma, as ferramentas de teste auxiliam na automatização dessa tarefa, resultando em maior eficiência e redução do esforço necessário para a realização do teste. Outras vantagens da utilização de ferramentas para esse fim é que as ferramentas permitem a aplicação prática dos critérios de teste; o apoio a estudos empíricos com a finalidade de comparar e avaliar os diversos critérios de testes existentes e a transferência das tecnologias de teste para a indústria.

Embora existam várias ferramentas para teste de programas OO, o número de ferramentas de teste de POA ainda é pequeno. Entre as ferramentas para o teste de programas OO podem-se citar: *C++ Test*, *JProbeSuite*, *JTest*, *Panorama C/C++*, *Panorama for Java*, *PureCoverage* e *JaBUTi*. Para programas OA foram encontradas as seguintes: *Aspectra*, *Wrasp*, *APTE*, *JamlUnit* e *JaBUTi/AJ*. A seguir é apresentada uma descrição sucinta de algumas dessas ferramentas, que foi parcialmente extraída dos trabalhos de Domingues (2002) e de Lemos (2005).

JProbeSuite(Quest Software, 2008) é um conjunto de três ferramentas: a *JProbe Profiler and Memory Debugger*, que ajuda a eliminar gargalos de execução causados por algoritmos ineficientes em código Java, e aponta as causas de perdas de memória nessas aplicações; a *JProbe Threadalyzer*, que monitora interações entre threads e avisa o testador quando essa interação representar perigo; e a *JProbe Coverage*, que localiza códigos não testados e mede quanto do código está sendo exercitado.

JTest(Parasoft Corparation, 2008) é a ferramenta que executa os seguintes tipos de teste para classes escritas em Java: análise estática; teste funcional; teste estrutural; e teste de regressão. No teste funcional ela gera automaticamente um conjunto essencial de casos de teste, projetado com o objetivo de alcançar a maior cobertura possível. Além disso, o testador pode melhorar esse conjunto de casos de teste, se desejar. Além de detetar erros, a ferramenta pode preveni-los, assegurando que eles não serão adicionados ao código quando for modificado de forma automatizada. A análise estática previne erros, padronizando o código-fonte. Qualquer violação no padrão é relatada como um defeito.

JUnit é um framework que oferece suporte à criação, execução e avaliação de casos de teste (Beck e Gamma, 2008). Com o JUnit é possível implementar algumas classes específicas que armazenam informações sobre os dados de entrada e a respectiva saída esperada para cada caso de teste. Após a execução de um caso de teste, a saída obtida é comparada com a saída esperada e as divergências, se houver, são relatadas. O JUnit não avalia a cobertura obtida pelos casos de teste segundo critérios de teste.

JaBUTi (Java Bytecode Understanding and TestIng) ferramenta de teste desenvolvida pelo Grupo de Pesquisa em Engenharia de Software do ICMC, em colaboração com outros grupos de pesquisa. Ela é um ambiente completo para o entendimento e teste de programas e componentes Java que fornece diferentes critérios de testes estruturais para a análise de cobertura, um conjunto de métricas estatísticas para avaliar a complexidade das classes que compõem o programa/componente e implementa ainda algumas heurísticas de particionamento de programas que visam auxiliar a localização de defeitos (Vincenzi, 2004).

JaBUTi/AJ (Java Bytecod Undesrtanding and TestIng for AspectJ) é uma extensão da ferramenta JaBUTi e apóia o teste de unidades de programas orientados a aspectos e também ao teste estrutural de integração par-a-par escritos em Java e AspectJ. Por ser uma extensão, todas as funcionalidades existentes na JaBUTi também estão presentes na JaBUTi/AJ (Franchin, 2007; Lemos et al., 2007; Lemos, 2005).

Aspectra ferramenta que opera em qualquer conjunto de casos de teste gerados para programas AspectJ, com o intuito de detetar os testes redundantes. Ela reduz o tamanho dos testes gerados para inspeção quando as especificações não estão escritas para programas AspectJ (Xie et al., 2005).

Wrasp framework proposto por Xie et al. (2005) para geração automática de testes para programas AspectJ. O Wrasp pode gerar casos de testes para testar a integração das seguintes unidades: métodos interceptados por um adendo, adendos e métodos inter-tipos. Ele também pode testar um adendo como uma unidade isolada. A principal contribuição do Wrasp é a síntese automática de classes empacotadoras (*wrapper*), que permite que ele gere testes para programas AspectJ, usando ferramentas existentes, que geram testes para programas Java. Ele é complementar ao Aspectra; o número de testes gerados pelos Wrasp pode ser minimizado com a inspeção da Aspectra.

JamlUnit é uma extensão do JUnit proposta como um framework para executar teste de unidade de aspectos escritos em JAML. O JAML (*Java Aspect Markup Language*) é um framework de linguagem extensível, desenvolvido por Lopes (2005). Especificamente, a ferramenta testa o comportamento aspectual, ou seja, o comportamento implementado pelos adendos. O JamlUnit utiliza objetos simulados (*mock objects*) que encapsulam a informação do contexto de execução (ponto de junção).

Ferramenta proposta por Massicotte et al. (2007) apóia a estratégia, discutida na seção anterior, que é baseada em diagramas de comunicação da UML. Ela dá suporte à geração de teste de sequência e ao processo de verificação.

4.6 Comparação entre as Abordagens

A Tabela 4.5 resume as abordagens apresentadas de teste estrutural de programas orientados a objetos e a aspectos levadas em consideração neste texto.

Nota-se que dentre as abordagens apresentadas neste texto apenas foram implementadas as do grupos do ICMC-Labes, resultando na família JaBUTi.

4.7 Considerações Finais

Neste capítulo foram apresentadas as principais estratégias de teste para programas orientados a objetos e programas orientados a aspectos, com enfoque ao teste estrutural. Também foram vistas algumas ferramentas que dão apoio ao teste de programas OO e OA e realizou-se uma comparação entre essas abordagens.

Considerando as estratégias de teste de unidade e de integração apresentadas na Seção 2.7, em se tratando de programas OO e OA, apenas o teste estrutural de unidade pode ser realizado completamente com as ferramentas de testes existentes. O teste estrutural de integração pode ser realizado parcialmente, uma vez que, atualmente, ele só é tratado entre duas unidades. Nota-se, então, que seria interessante que houvesse um ambiente completo para teste estrutural de programas OO e OA e que torne possível realizar o teste de mais do que duas unidades, como por exemplo, a integração de uma unidade com todas as que interagem com ela.

Tabela 4.5: Resumo das abordagens para teste estrutural de POO e POA

Autor(es)	Abordagem	Fase/Nível	Critérios propostos	Grafo	Ferramenta
Harrold e Rothermel (1994)	Teste estrutural de unidade/integração apenas para POO	Unidade: Intramétodo Intermétodo Intraclasse Integração: Interclasse	Não propõe nenhum critério	Intramétodo e Intermétodo: GFC Intraclasse: CCFG	Não
Vincenzi (2004)	Teste estrutural de unidade apenas para POO	Unidade: Intramétodo	Fluxo de Controle: todos-nos-independentes-de-exceção todos-nos-dependentes-de-exceção todas-arestas-independentes-de-exceção todas-arestas-dependentes-de-exceção Fluxo de Dados: todos-usos-independentes-de-exceção todos-usos-dependentes-de-exceção	$\mathcal{DU}$	JaBUTi
Zhao (2002, 2003)	Teste estrutural de unidade/integração	Unidade: Intramódulo Intermódulo Intra-aspecto/ classe Integração Interclasse	Não propõe nenhum critério	Intramódulo:CFG Intermódulo: ICFG Intra-aspecto/ classe: FCFG	Não
Lemos et al (2005, 2007)	Teste estrutural de unidade	Unidade: Intramétodo/ Intra-adendo Integração: Intermétodo adendo-método método-adendo adendo-adendo intermétodo -adendo intra-classe inter-classe	Fluxo de Controle: todos-nos-transversais todas-arestas-transversais Fluxo de Controle: todos-usos transversais	Intramétodo e intra-adendo: $\mathcal{AODU}$ método-adendo: $\mathcal{MADU}$	JaBUTi/AJ (unidade)
Franchin (2007)	Teste estrutural de integração par-a-par	O mesmo de Lemos(2007)	Fluxo de Controle: todos-nos-integrados todas-arestas-integradas Fluxo de Dados: todos-usos-integrados	$\mathcal{PWDU}$: método-método método-adendo adendo-método adendo-adendo	JaBUTi/ AJ/PW
Lemos (2009)	Teste estrutural de integração baseada em conjuntos de junção	Integração adendo-método método-adendo	Fluxo de Controle: todos-nos-CJ todas-arestas-CJ Fluxo de Controle todos-usos-CJ	$\mathcal{PCDU}$	JaBUTi/ PC-AJ

CAPÍTULO

5

Teste de Integração Contextual de Programas Orientados a Objetos e a Aspectos

5.1 Considerações Iniciais

Neste capítulo é apresentada uma abordagem para o teste estrutural de integração nível um para programas OO e OA escritos em Java e AspectJ. Por meio dessa abordagem é possível revelar defeitos que podem ocorrer nas interfaces entre uma determinada unidade (método ou adendo) e todas as outras que interagem diretamente com ela, bem como revelar defeitos que podem ocorrer em sequências de chamadas dessas unidades.

Este capitulo está organizado da seguinte forma: na Seção 5.2 é definido o teste estrutural de integração nível um; na Seção 5.3 é discutido o modelo de fluxo de dados que é utilizado para identificar o que caracteriza uma definição ou um uso de uma variável em uma sentença Java; na Seção 5.4 é definido o grafo de fluxo utilizado pela abordagem de teste proposta; na Seção 5.5 são apresentados os critérios de teste propostos; na Seção 5.6 são apresentados alguns casos especiais desse tipo de teste e, por fim, na Seção 5.7 são apresentadas as considerações finais do capítulo.

5.2 Teste Estrutural de Integração Nível Um

Programas OO ou OA são constituídos de módulos que definem unidades. Essas unidades podem se relacionar com o objetivo de produzir um determinado comportamento no programa. Em um programa OO, os módulos são as classes e as unidades são os métodos. Já em um programa OA, os módulos são as classes e aspectos e as unidades são métodos e adendos.

O teste estrutural de integração nível um de um programa OO ou OA tem como objetivo testar a interação entre uma determinada unidade com todas as outras que interagem diretamente com ela. Para programas OO, esse tipo de teste envolve testar a interação entre métodos; já para programas OA, o teste estrutural de integração nível um deve considerar as interações método-método, método-adendo, adendo-adendo e adendo-método.

Na Figura 5.1 é mostrado um exemplo simples de programa OA contendo duas classes C1 e C2 e um aspecto A1. A classe `C1` possui os métodos `m1()`, `m2()` e `m3()`. A classe `C2` tem um método `m4()`. No aspecto `A1` são definidos o conjunto de junção (*pointcut*) `pc1`, que captura as execuções do método `m2()` da classe `C1`, e o *pointcut* `pc2`, que captura as chamadas ao método `m3()` da classe `C1`. O adendo anterior `before` é disparado antes da execução dos pontos de junções selecionados pelo *pointcut* `pc1` e o adendo posterior `after` é disparado após a chamada dos pontos de junções selecionados pelo *pointcut* `pc2`. Na Tabela 5.1 são mostradas as interações entre as unidades desse exemplo.

```
public class C1 {
  public void m1(){
    //...
      m2();
    //...
      m3();
    //...
      C2 c2 = new C2();
      c2.m4();
    //...
  }
  public void m2(){
    // o adendo before() do aspecto A1
    //é disparado nesse ponto, antes
    //da execução de m2().
  }
  public static void m3(){
    //...
  }
}

public class C2 {
  public void m4(){
    //...
  }
}
```

```
public aspect A1 {
  pointcut pc1(): execution (* C1.m2());
  pointcut pc2(): call(* C1.m3());

  before(): pc1(){
    //...
      C1 c1 = new C1();
      c1.m1();
    //...
      c1.m3();
    // o adendo after() definido neste
    //aspecto é disparado nesse ponto,
    //após a chamada ao método m3() de C1.
  }

  after(): pc2(){
    //...
    C2 c2 = new C2();
    c2.m4();
    //...
  }
}
```

Figura 5.1: Exemplo simples de um programa OA para demonstrar a interação nível um entre as unidades.

Tabela 5.1: Interações entre as unidades do exemplo da Figura 5.1

Unidade chamadora	Unidades chamadas/que afetam
(C1) m1	(C1) m2; (C1) m3(); (C2) m4; (A1) `after`
(C1) m2	(A1) `before`
(A1) `before`	(C1) m1, (C1) m3, (A1) `after`
(A1) `after`	(C2) m4

Na Figura 5.2 é representada uma sequência de chamadas com profundidade nível um para o método m1. Ele chama diretamente os métodos m1, m2, m3 e m4 e é afetado pelo adendo `after`.

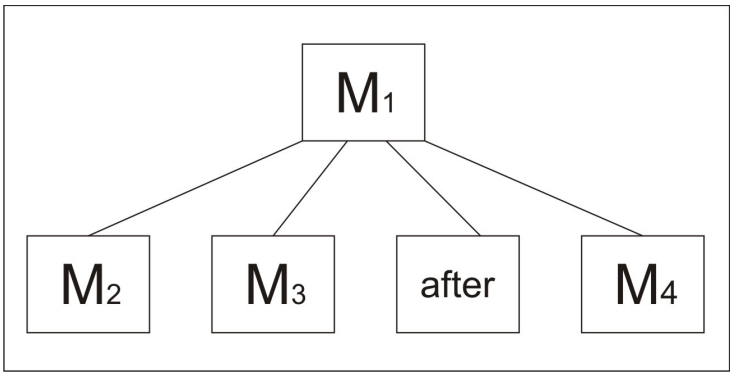

Figura 5.2: Sequência de chamadas para o método `m1` do exemplo

5.3 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 segue o modelo de fluxo de dados proposto por Vincenzi (2004) e adaptado por Franchin (2007). O modelo de dados é proposto utilizando *bytecode* Java, e o AspectJ não introduz nenhum tipo de dado novo (os aspectos são considerados como uma classe *singleton* e os adendos são tratados como métodos). Os seguintes tipos de dados são considerados: variáveis locais, elementos de array, atributos estáticos , atributos de instância e parâmetros formais.

Para mostrar alguns exemplos de decisões tomadas, considere as seguintes definições:

- *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 atributi 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;

Foram estabelecidas as seguintes regras (conservadoras) para classificar as definições e usos:

1. Não é considerado o uso de um literal *c*.
2. A definição ou uso de uma variável primitiva *p* é considerada como 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 está 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 somente a definição de *r*. A definição de uma variável de referência *r* envolvendo uma referência a um objeto é considerada como a definição de *r* e, no caso de uma 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 unicamente o uso de *r*.
4. Variáveis agregadas são consideradas como uma única posição de memória. Assim, a definição de uma variável agregada *a[]*, que é um elemento de uma array referenciado pela variável de referência *a*, é considerada como 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 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 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 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 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 rerência *r* do tipo da classe *Cl*, no nível de *bytecode*, tal variável de referência é automaticamente convertida no nome da classe, não 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 1 a 6. Na invocação de um método estático *ms*, tal como *Cl.ms*($t_1; t_2; \cdots ; t_n$), considera-se que ocorrem usos 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*.

5.4 Definição do Grafo de Fluxo de Controle/Dados

O grafo $\mathcal{INTP}$ 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 $\mathcal{AODU}$, definido por Lemos (2005) e adaptado por Franchin (2007) (ver Seções 4.3.2 e 4.3.3), da unidade u_0 (chamadora). O mesmo vale para os nós de saída, ou seja, os nós de saída do

grafo $\mathcal{INTER}$ 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 $\mathcal{INTER}$ 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 $\mathcal{INTER}$ e que $d \in \mathbb{N}$. O grafo $\mathcal{INTER}$ das unidades $u_0, u_1, \cdots, u_n$ é definido como um grafo dirigido $\mathcal{INTER}\ (u_0, u_1, \cdots, u_n) = (N, E, s, T, I, I_s, R)$, tal que:

- Seja $\overline{N_k} = N_{k_1} \bigcup N_{k_2} \bigcup \cdots \bigcup N_{k_i}$, onde $i \in \mathbb{N}^+$ indica o número de vezes que u_k é chamado repetidamente em u_0.
  - $N = N_0 \bigcup \overline{N_0} \bigcup \overline{N_1} \bigcup \cdots \bigcup \overline{N_n}$ representa o conjunto completo de nós do grafo $\mathcal{INTER}$:
    - $N_I = \overline{N_0} \bigcup \overline{N_1} \bigcup \cdots \bigcup \overline{N_n}$ 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 $\overline{E_k} = E_{k_1} \bigcup E_{k_2} \bigcup \cdots \bigcup E_{k_i}$, onde $i \in \mathbb{N}^+$ indica o número de vezes que u_k é chamado repetidamente em u_0
  - $E = E'_0 \bigcup \overline{E_0} \bigcup \overline{E_1} \bigcup \cdots \bigcup \overline{E_n} \bigcup \overline{E_{I_0}} \bigcup \overline{E_{I_1}} \bigcup \cdots \bigcup \overline{E_{I_n}}$ é o conjunto completo de arestas do grafo $\mathcal{INTER}$, 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.
  - $\overline{E_{I_k}} = E_{I_{k_1}} \bigcup E_{I_{k_2}} \cdots \bigcup E_{I_{k_i}}$, 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 $\mathcal{AODU}$, 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 $x \in T_{k_i}$;
  - $E_I = \overline{E_{I_0}} \bigcup \overline{E_{I_1}} \bigcup \cdots \bigcup \overline{E_{I_n}}$ é o conjunto de arestas integradas;

- $s \in N$ e $s = s_0$ é o nó de entrada do grafo $\mathcal{INIP}$, 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 $\mathcal{INIP}$, 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}
- $I = I_0 \bigcup \overline{I_0} \bigcup \overline{I_1} \bigcup \cdots \bigcup \overline{I_n}$ é o conjunto completo de nós de interação (ou seja, nós transversais e nós de chamada) do grafo $\mathcal{INIP}$, tal que:
  - $\overline{I_k} = I_{k_1} \bigcup I_{k_2} \bigcup \cdots \bigcup I_{k_i}$
  - $I_{k_i} \subseteq N_{k_i}$ 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 $R = \{y \in N_0 | \exists (x,y) \in E_d \wedge x \in I_s\}$.

Para o caso em que uma unidade sob teste é afetada por um adendo `around` que chama o método `proceed` há uma exceção na definição do grafo. Nesse caso, o grafo $\mathcal{AODU}$ do método entrecortado é integrado dentro da chamada do próprio adendo (no nó correspondente ao método `proceed`). Assim, a aresta integrada que liga o nó correspondente a chamada do método `proceed` ao próximo nó é deletada e são adicionadas duas novas arestas: a que liga

o nó da chamada ao `proceed` ao nó de entrada do método entrecortado e a que liga o nó de saída do método entrecortado ao próximo nó do grafo do adendo.

A representação gráfica do grafo $\mathcal{INIP}$ é parecida com a do grafo $\mathcal{PWDU}$ e é definida da seguinte forma:

- Um nó regular é representado por um círculo desenhado com 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 a primeira instrução de *bytecode* do bloco;
- Um nó transversal é representado por uma elipse desenhada com linha tracejada. Seu rótulo informa, além da primeira instrução de *bytecode* do bloco, qual tipo de adendo afeta aquele ponto (`before`, `after` ou `around`) 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 a primeira instrução de *bytecode* do bloco;
- Um nó integrado é representado como um nó regular, um nó de chamada, um nó transversal ou um nó de saída[1]). Como diferencial, seu rótulo tem um prefixo iniciado por um número e seguido por ":";
- 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;

Os conjuntos de definições e usos dos nós do grafo $\mathcal{INIP}$ são derivados a partir das instruções presentes no bloco de instruções de cada nó. Com o grafo $\mathcal{INIP}$ criado a partir da integração entre o grafo $\mathcal{AODU}$ da unidade chamadora com os grafos $\mathcal{AODU}$'s das unidades chamadas já é possível derivar os requisitos de teste para os critérios de fluxo de controle e fluxo de dados, conforme é discutido na Seção 5.5.

[1] Um nó de saída é transformado em um nó regular de integração

5.4.1 Exemplo 1

Considere-se o código-fonte apresentado na Figura 5.3. Nesse código é apresentado a implementação de uma tabela de símbolos adaptada do exemplo apresentado por Harrold e Rothermel (1994). O código-fonte possui a classe `SymbolTable`, em que são implementados os principais método que uma tabela de símbolos deve possuir e o aspecto `AspectSymbolTable`, que foi implementado apenas para exemplicar o entrecorte de adendos na classe `SymbolTable`.

```
/*01*/ public class SymbolTable {

/*02*/   private int numentries;
/*03*/   private int index;
/*04*/   private final int tablemax;
/*05*/   private String[][] tabSimbolos;
/*06*/   String Info;

/*07*/   public SymbolTable(int maxSymbols){
/*08*/     tablemax = maxSymbols;
/*09*/     numentries=0;
/*10*/     tabSimbolos = new String[tablemax][2];
/*11*/   }

/*12*/   public boolean addToTable(String symbol,
                        String symbolInfo){
/*13*/     if (numentries<tablemax){
/*14*/       if (lookup(symbol)!=-1){
/*15*/         return false;
/*16*/       }
/*17*/       addSymbol(symbol);
/*18*/       addInfo(symbolInfo, index);
/*19*/       numentries++;
/*20*/       return true;        //OK
/*21*/     }
/*22*/     return false;         //NOT OK
/*23*/   }

/*24*/   public int getFromTable(String symbol){
/*25*/     int index = lookup(symbol);
/*26*/     if (index == -1){
/*27*/       Info = null;
/*28*/       return -1; //NOT OK
/*29*/     }

/*30*/     Info = getInfo(index);
/*31*/     return index;         //OK
/*32*/     }

/*33*/   private int lookup(String key){
/*34*/     int saveindex;
/*35*/     index = saveindex = hash(key);
/*36*/     while (!getSymbol(index).equals(key)){
/*37*/       index++;

/*38*/       if (index==tablemax)       /*wrap around*/
/*39*/         index=0;
/*40*/       if (getSymbol(index).equals("NOT FOUND") ||
/*41*/            index==saveindex)
/*42*/         return -1;        //NOT FOUND
/*43*/     }
/*44*/     return index;         //FOUND
/*45*/   }
```

```
/*46*/   private int hash(String symbol){
/*47*/     int key;
/*48*/     key = Math.abs(symbol.hashCode())%tablemax;
/*49*/     return key;
/*50*/   }

/*51*/   private void addSymbol(String symbol){
/*52*/     index = hash(symbol);
/*53*/     while (!getSymbol(index).equals("NOT FOUND")){
/*54*/       if (index==tablemax-1) index = 0;
/*55*/       else index++;
/*56*/     }
/*57*/     tabSimbolos[index][0] = symbol;
/*58*/   }

/*59*/   private void addInfo(String symbolInfo, int index){
/*60*/     tabSimbolos[index][1] = symbolInfo;
/*61*/   }

/*62*/   private String getSymbol(int index){
/*63*/     if (tabSimbolos[index][0]!=null){
/*64*/       return tabSimbolos[index][0];       //FOUND
/*65*/     }
/*66*/     else
/*67*/       return "NOT FOUND";
/*68*/   }

/*69*/   private String getInfo(int index){
/*70*/     return tabSimbolos[index][1];
/*71*/   }
/*72*/ }


/*72*/ public aspect AspectSymbolTable {

/*73*/   pointcut chamaAddSymbol(): call(* addSymbol(*));
/*74*/   pointcut chamaGetFromTable(): call(* getFromTable(*));
/*75*/   pointcut execGetSymbol(int index):
           execution (* getSymbol(int)) && args(index);
/*76*/   pointcut execLookup(String key):
           execution(int lookup(String))  && args(key);

/*77*/   before(int index): execGetSymbol(index){
/*78*/     System.out.println("Procurando símbolo na posição: "
             + index);
/*79*/   }

/*80*/   after(String key) returning: execLookup(key){
/*81*/   System.out.println("Símbolo procurado:" + key);
/*82*/ }
```

Figura 5.3: Código fonte do algoritmo para tabela de símbolos.

Como exemplo, considere-se o grafo integrado de profundidade 1 do método `addToTable`. Como nesse exemplo não há recursão nem repetição de chamada de uma mesma unidade, $\overline{N_k} = N_k$. Assim, suas unidades são: u_0 = `addToTable`, u_1 = `lookup`, u_2 = `addSymbol` e u_3 = `addInfo`, em que `addToTable` é a unidade chamadora e as demais são as unidades

chamadas. As Figuras 5.4, 5.5, 5.6 e 5.7 apresentam, respectivamente, os grafos $\mathcal{AODU}$ de cada umas dessas unidades gerados pela ferramenta JaBUTi e também seus *bytecodes*. O grafo $\mathcal{INIP}$ (u_0, u_1, u_2, u_3) = (N, E, s, T, I, I_s, R) tal que:

```
boolean addToTable(Ljava/lang/String;Ljava/lang/String;)Z
0:  aload_0
1:  getfieldSymbolTable.numentries I (22)
4:  aload_0
5:  getfieldSymbolTable.tablemax I (20)
8:  if_icmpge#48
11: aload_0
12: aload_1
13: invokespecialSymbolTable.lookup (Ljava/lang/String;)I (34)
16: iconst_m1
17: if_icmpeq#22
20: iconst_0
21: ireturn
22: aload_0
23: aload_1
24: invokespecialSymbolTable.addSymbol (Ljava/lang/String;)V (38)
27: aload_0
28: aload_2
29: aload_0
30: getfieldSymbolTable.index I (42)
33: invokespecialSymbolTable.addInfo (Ljava/lang/String;I)V (44)
36: aload_0
37: dup
38: getfieldSymbolTable.numentries I (22)
41: iconst_1
42: iadd
43: putfieldSymbolTable.numentries I (22)
46: iconst_1
47: ireturn
48: iconst_0
49: ireturn
```

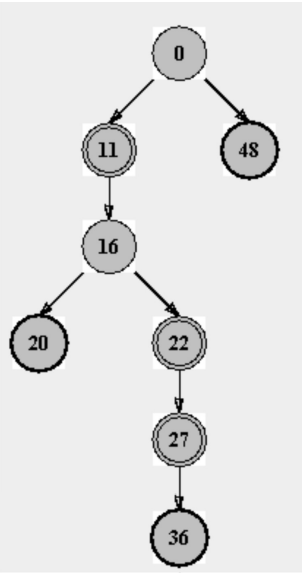

Figura 5.4: *Bytecode* e grafo $\mathcal{AODU}$ do método `addToTable`

N_0 = {0, 11, 48, 16, 20, 22, 27, 36};

E_0 = {(0, 11), (0, 48), (11, 16), (16, 20), (16, 22), (22, 27), (27, 36) }

```
int lookup(Ljava/lang/String;)I
 0:  aload_1
 1:  astore_3
 2:  aload_0
 3:  aload_0
 4:  aload_1
 5:  invokespecialSymbolTable.hash
                (Ljava/lang/String;)I (58)
 8:  dup
 9:  istore_2
 10: putfieldSymbolTable.index I (42)
 13: goto#72
 16: aload_0
 17: dup
 18: getfieldSymbolTable.index I (42)
 21: iconst_1
 22: iadd
 23: putfieldSymbolTable.index I (42)
 26: aload_0
 27: getfieldSymbolTable.index I (42)
 30: aload_0
 31: getfieldSymbolTable.tablemax I (20)
 34: if_icmpne#42
 37: aload_0
 38: iconst_0
 39: putfieldSymbolTable.index I (42)
 42: aload_0
 43: aload_0
 44: getfieldSymbolTable.index I (42)
 47: invokespecialSymbolTable.getSymbol (I)
                Ljava/lang/String; (61)
 50: ldc"NOT FOUND" (64)
 52: invokevirtualjava.lang.String.equals
                (Ljava/lang/Object;)Z (66)
 55: ifne#66
 58: aload_0
 59: getfieldSymbolTable.index I (42)
 62: iload_2
 63: if_icmpne#72
 66: iconst_m1
 67: istore%4
 69: goto#93
 72: aload_0
 73: aload_0
 74: getfieldSymbolTable.index I (42)
 77: invokespecialSymbolTable.getSymbol
                (I)Ljava/lang/String; (61)
 80: aload_1
 81: invokevirtualjava.lang.String.equals
                (Ljava/lang/Object;)Z (66)
 84: ifeq#16
 87: aload_0
 88: getfieldSymbolTable.index I (42)
 91: istore%4
 93: invokestaticAspectSymbolTable.aspectOf ()
                LAspectSymbolTable; (91)
 96: aload_3
 97: invokevirtualAspectSymbolTable.
        ajc$afterReturning$AspectSymbolTable$
        2$2a30a747 (Ljava/lang/String;)V (94)
100: iload%4
102: ireturn
```

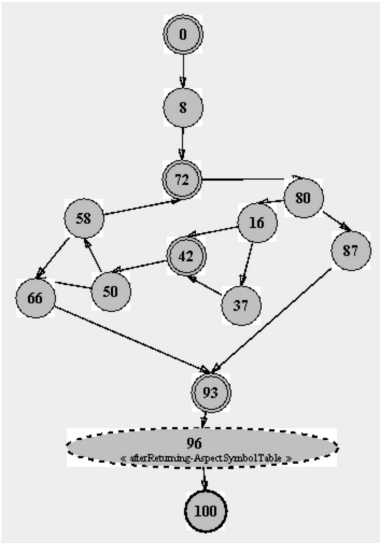

Figura 5.5: *Bytecode* e grafo $\mathcal{AODU}$ do método `lookup`

N_1 = {0, 8, 72, 80, 16, 42, 37, 87, 50, 58, 66, 93, 96, 100}

E_1 = {(0, 8), (8, 72), (72, 80), (80, 16), (80, 87), (16, 42), (16, 37), (37, 42), (42, 50), (50, 58), (50, 66), (58, 66), (66, 93), (87, 93), (93, 96), (96, 100)}

```
void addSymbol(Ljava/lang/String;)V
0:   aload_0
1:   aload_0
2:   aload_1
3:   invokespecialSymbolTable.hash (Ljava/lang/String;)I (58)
6:   putfieldSymbolTable.index I (42)
9:   goto#43
12:  aload_0
13:  getfieldSymbolTable.index I (42)
16:  aload_0
17:  getfieldSymbolTable.tablemax I (20)
20:  iconst_1
21:  isub
22:  if_icmpne#33
25:  aload_0
26:  iconst_0
27:  putfieldSymbolTable.index I (42)
30:  goto#43
33:  aload_0
34:  dup
35:  getfieldSymbolTable.index I (42)
38:  iconst_1
39:  iadd
40:  putfieldSymbolTable.index I (42)
43:  aload_0
44:  aload_0
45:  getfieldSymbolTable.index I (42)
48:  invokespecialSymbolTable.getSymbol (I)Ljava/lang/String; (61)
51:  ldc"NOT FOUND" (64)
53:  invokevirtualjava.lang.String.equals (Ljava/lang/Object;)Z (66)
56:  ifeq#12
59:  aload_0
60:  getfieldSymbolTable.tabSimbolos [[Ljava/lang/String; (25)
63:  aload_0
64:  getfieldSymbolTable.index I (42)
67:  aaload
68:  iconst_0
69:  aload_1
70:  aastore
71:  return
```

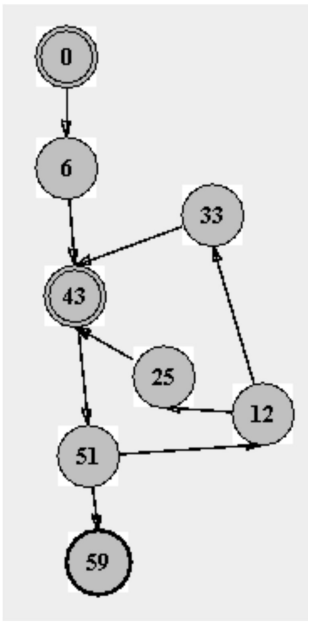

Figura 5.6: *Bytecode* e grafo $\mathcal{AODU}$ do método `addSymbol`

N_2 = {0, 6, 43, 51, 33, 25, 12, 59}

E_2 = {(0, 6), (6, 43), (43, 51), (51, 12), (51, 59), (12, 25), (12, 33), (33, 43), (25, 43)}

```
void addInfo(Ljava/lang/String;I)V
0:   aload_0
1:   getfieldSymbolTable.tabSimbolos [[Ljava/lang/String; (25)
4:   iload_2
5:   aaload
6:   iconst_1
7:   aload_1
8:   aastore
9:   return
```

Figura 5.7: *Bytecode* e grafo $\mathcal{AODU}$ do método `addInfo`

N_3 = {0}

E_3 = $\emptyset$

- $N = N_0 \bigcup N_1 \bigcup N_2 \bigcup N_3$:
  - N_0 = {0, 11, 48, 16, 20, 22, 27, 36};

  - $N_I = N_1 \bigcup N_2 \bigcup N_3$ = {1:0, 1:8, 1:72, 1:80, 1:16, 1:42, 1:37, 1:87, 1:50, 1:58, 1:66, 1:93, 1:96, 1:100} $\bigcup$ {2:0, 2:6, 2:43, 2:51, 2:33, 2:25, 2:12, 2:59} $\bigcup$ {3.0}

- $E = E'_0 \bigcup E_1 \bigcup E_2 \bigcup E_3 \bigcup E_{I_1} \bigcup E_{I_2} \bigcup E_{I_3}$
  - E_0 = {(0, 11), (0, 48), (11, 16), (16, 20), (16, 22), (22, 27), (27, 36) }
  - E_d = {(11, 16), (22, 27), (27, 36) }
  - $E'_0 = E_0 - E_d$ = {(0, 11), (0, 48), (16, 20), (16, 22)}
  - E_1 = {(1:0, 1:8), (1:8, 1:72), (1:72, 1:80), (1:80, 1:16), (1:80, 1:87), (1:16, 1:42), (1:16, 1:37), (1:37, 1:42), (1:42, 1:50), (1:50, 1:58), (1:50, 1:66), (1:58, 1:66), (1:66, 1:93), (1:87, 1:93), (1:93, 1:96), (1:96, 1:100) }
  - E_2 = {(2:0, 2:6), (2:6, 2:43), (2:43, 2:51), (2:51, 2:12), (2:51, 2:59), (2:12, 2:25), (2:12, 2:33), (2:33, 2:43), (2:25, 2:43)}
  - $E_3 = \emptyset$
  - E_{I_1} = {(11, 1:0), (1:100, 16)}
  - E_{I_2} = {(22, 2:0), (2:59, 27)}
  - E_{I_3} = {(27, 3:0), (3:0, 36)}

- s = 1
  - s_0 = 0;
  - s_1 = 1:0;
  - s_2 = 2:0;
  - s_3 = 3:0;

- T_0 = {20, 36, 48};

- $I = I_0 \bigcup I_1 \bigcup I_2 \bigcup I_3$:
  - I_0 = {11, 22, 27};
  - I_1 = {1:0, 1:72, 1:42, 1:93, 1:96};
  - I_2 = {2:0, 2:43};
  - $I_3 = \emptyset$;

- I_s = {11, 22, 27}

- R = {16, 27, 36}

A Figura 5.8 apresenta o grafo $\mathcal{INIP}$ do exemplo.

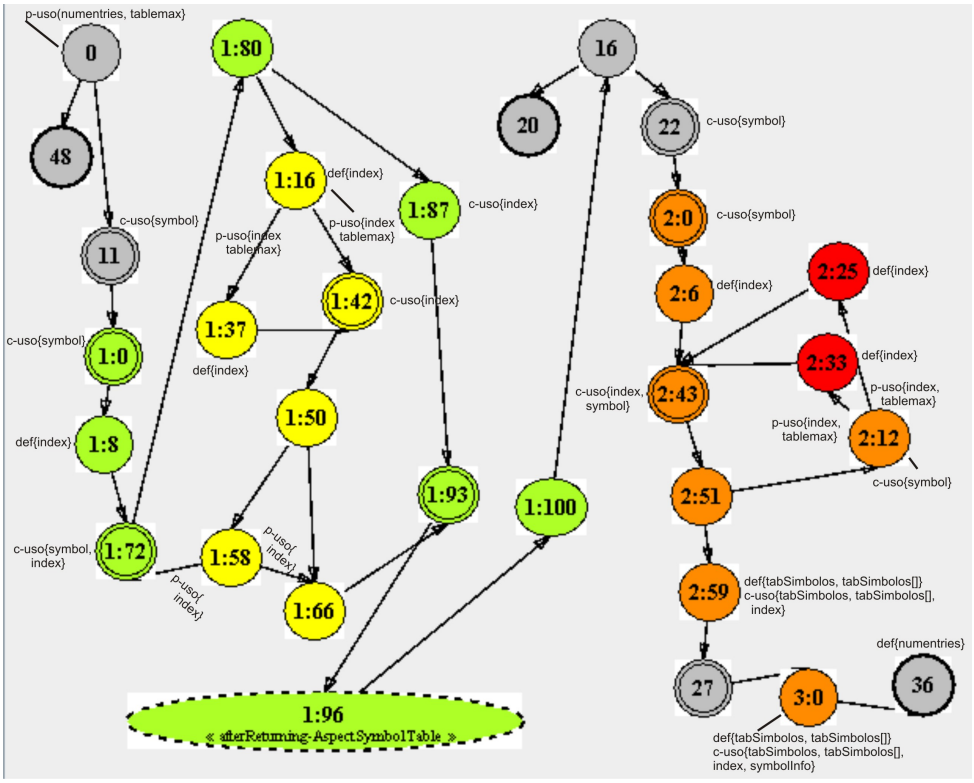

Figura 5.8: Grafo integrado de profundidade 1 do método `addToTable`

5.4.2 Exemplo 2

As Figuras 5.9 e 5.10 apresentam, respectivamente, o código fonte e o grafo $\mathcal{INIP}$ do segundo exemplo, que ilustra chamadas repetidas e recursivas. Para simplificar o exemplo, quando houver a definição ou uso de elementos de um array será representado como a definição ou uso apenas do array.

Sejam u_0 = `metodo1`, u_1 = `before` u_2 = `metodo2` e u_3 = `metodo3`. São definidos os seguintes conjuntos:

Como há repetições, o conjunto de chamadas é representado por $u_0u_{0_1}u_{1_1}u_{1_2}u_{2_1}u_{2_2}u_{3_1}$.

N_0 = {0, 84, 95, 106, 180, 112, 127, 138, 149, 164, 177, 191, 197, 219}

N_{0_1} = {6:0, 6:84, 6:95, 6:106, 6:180, 6:191, 6:197, 6:214, 6:219, 6:112, 6:127, 6:138, 6:149, 6:164, 6:177}

N_{1_1} = {1:0, 1:12, 1:29, 1:48, 1:60, 1:77, 1:96, 1:108, 1:125, 1:144}

N_{1_2} = {3:0, 3:12, 3:29, 3:48, 3:60, 3:77, 3:96, 3:108, 3:125, 3:144}

N_{2_1} = {2:0}

N_{2_2} = {4:0}

```
public class Principal {
  public static int var1;
  public static int var2;
  public static int var3;

  public static void metodo1(Integer[] y, Integer[] z){
    Integer[] x = new Integer[1];
    x[0] = 2;
    Integer[] w = new Integer[1];
    w[0] = 5;
    var1 = x[0]+y[0];
    var2 = var1 + z[0];
    var3 = z[0]+y[0];

    metodo2(x, y, z, w);
    for (int i=0;i<w[0];i++){
      metodo2(x,z, y, w);
          if (x[0] > z[0]) x[0]++;
    }
    metodo3(x, y, z);
    x[0] = 2;
    if (x[0] > 2) metodo1(y, z);
  }

  public static void metodo2(Integer[] a, Integer[] b,
        Integer[] c, Integer[] d) {

    a[0] = b[0] + c[0] + d[0];
    c[0] = c[0] + a[0];
  }
```

```
  public static void metodo3(Integer[] i, Integer[] j,
        Integer[] k) {

    if (i[0] > j[0]) j[0] = k[0];
    if (k[0] > i[0]) i[0] = j[0];
  }
}

public aspect Aspecto {
  pointcut pcMetodo(Integer[] l, Integer[] m, Integer[] n,
        Integer[] o):
    call(* *.metodo2(..)) && args(l, m, n, o);

  before (Integer l[], Integer[] m, Integer[] n,
        Integer[] o): pcMetodo(l, m, n, o){

    if (Principal.var1 > l[0]) l[0] = l[0]+2;
    else l[0] = m[0]+n[0];

    if (Principal.var2 > m[0]) m[0] = m[0]+2;
    else m[0] = l[0]+n[0];

    if (Principal.var3 > n[0]) n[0] = n[0]+4;
    else n[0] = m[0]+l[0];
  }
}
```

Figura 5.9: Código fonte do segundo exemplo.

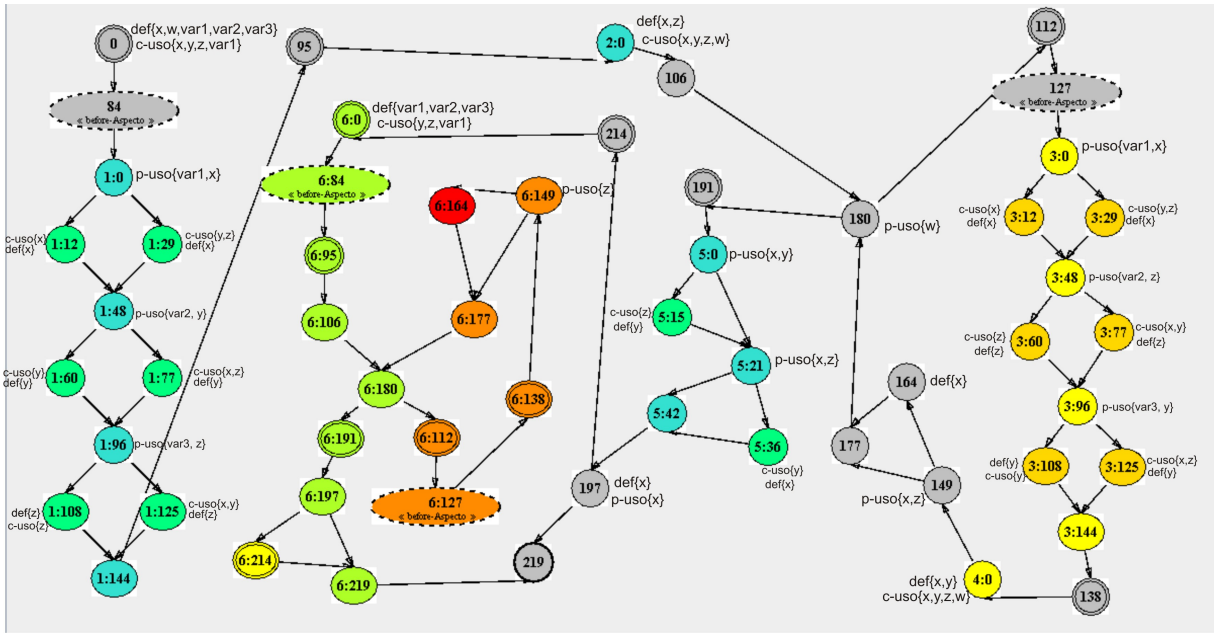

Figura 5.10: Grafo $\mathcal{INIP}$ do método `metodo1` do segundo exemplo

N_{3_1} = {5:0, 5:15, 5:21, 5:42, 5:36}

$\overline{N_0} = N_{0_1}$

$\overline{N_1} = N_{1_1} \bigcup N_{1_2}$

$\overline{N_2} = N_{2_1} \bigcup N_{2_2}$

$\overline{N_3} = N_{3_1}$

$N = N_0 \bigcup \overline{N_0} \bigcup \overline{N_1} \bigcup \overline{N_2} \bigcup \overline{N_3}$

$N_I = \overline{N_0} \bigcup \overline{N_1} \bigcup \overline{N_2} \bigcup \overline{N_3}$

E_{0_1} = {(6:0, 6:84), (6:84, 6:95), (6:95, 6:106), (6:106, 6:180), (6:180, 6:191), (6:191, 6:197), (6:197, 6:214), (6:197, 6:219), (6:214, 6:219), (6:180, 6:112),(6:112, 6:127), (6:127, 6:138), (6:138, 6:149), (6:149, 6:164), (6:164, 6:177), (6:177, 6:180)}

E_{1_1} = {(1:0, 1:12), (1:0, 1:29), (1:12, 1:48), (1:29, 1:48), (1:48, 1:60), (1:48, 1:77), (1:60, 1:96), (1:77, 1:96), (1:96, 1:108), (1:96, 1:125), (1:108, 1:144), (1:108, 1:125)}

E_{1_2} = {(3:0, 3:12), (3:0, 3:29), (3:12, 3:48), (3:29, 3:48), (3:48, 3:60), (3:48, 3:77), (3:60, 3:96), (3:77, 3:96), (3:96, 3:108), (3:96, 3:125), (3:108, 3:144), (3:108, 3:125)}

$E_{2_1} = \emptyset$

$E_{2_2} = \emptyset$

E_{3_1} = {(5:0, 5:15), (5:0, 5:21), (5:15, 5:21), (5:21, 5:42), (5:21, 5:36), (5:36, 5:42)}

$\overline{E_0} = E_{0_1}$

$\overline{E_1} = E_{1_1} \bigcup E_{1_2}$

$\overline{E_2} = E_{2_1} \bigcup E_{2_2}$

$\overline{E_3} = E_{3_1}$

$\overline{E_{I_0}} = E_{I_{0_1}}$

$\overline{E_{I_1}} = E_{I_{1_1}} \bigcup E_{I_{1_2}}$

$\overline{E_{I_2}} = E_{I_{2_1}} \bigcup E_{I_{2_2}}$

$\overline{E_{I_3}} = E_{I_{3_1}}$

E'_0 = {(0, 84), (84, 95), (95, 106), (106, 180), (180, 112), (112, 127), (127, 138), (138, 149), (149, 177), (149, 164), (164, 177), (177, 180), (180, 191), (191, 197), (197, 214), (214, 219), (197, 219)}

E_d = {(84, 95), (95, 106), (127, 138), (138, 149), (191, 197), (214, 219)}

$E_0 = E'_0 \bigcup E_d$

$E_I = \overline{E_{I_0}} \bigcup \overline{E_{I_1}} \bigcup \overline{E_{I_2}} \bigcup \overline{E_{I_3}} \bigcup \overline{E_{I_4}}$

s_0 = {0}

s_{0_1} = {6:0}; s_{1_1} = {1:0}; s_{1_2} = {3:0}; s_{2_1} = {2:0}; s_{2_2} = {4:0}; s_{3_1} = {5:0};

T = {19}= T_0

$I = I_0 \bigcup \overline{I_0} \bigcup \overline{I_1} \bigcup \overline{I_2} \bigcup \overline{I_3}$

$= I_0 \bigcup I_{0_1} \bigcup I_{1_1} \bigcup I_{1_2} \bigcup I_{2_1} \bigcup I_{2_2} \bigcup I_{3_1}$

I_s = {84, 95, 127, 138, 191, 214}

R = {95, 106, 138, 149, 197, 219}

5.5 Critérios de Fluxo de Controle e de Fluxo de Dados para Teste Estrutural de Integração Nível Um

Nesta seção são apresentadas as definições dos critérios de teste estrutural de integração nível um, os quais podem ser utilizados para derivar os requisitos de teste de integração nível um de programas OO e OA. Após a definição dos critérios, um exemplo é apresentado.

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 $\mathcal{IN}1\mathcal{P}$ é o grafo de fluxo de controle/dados integrado de nível um de profundidade 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 $x = y_l$ para algum $l, 1 \leq l \leq m$. Similarmente, uma aresta (x_1, x_2) é incluída em Π se Π contém um caminho $(y_1, \cdots, y_m)$ tal que $x_1 = y_l$ e $x_2 = y_{l+1}$ para algum $l, 1 \leq l \leq m - 1$. Assim, definem-se:

- **Todos-nós-integrados-N1:** Π satisfaz o critério todos-nós-integrados-N1 se todo nó $x \in N_I$ está incluído em Π. Em outras palavras, esse critério requer que cada nó integrado de um grafo $\mathcal{IN}1\mathcal{P}$ seja exercitado ao menos uma vez por algum caso de teste em T.
- **Todas-arestas-integradas-N1:** Π satisfaz o critério todas-arestas-integradas-N1 se cada aresta integrada $e \in E_I$ está incluída em Π. Em outras palavras, esse critério requer que cada aresta integrada de um grafo $\mathcal{IN}1\mathcal{P}$ seja exercitado ao menos uma vez por algum caso de teste em T.

5.5.2 Critérios de Fluxo de Dados

Para o teste de fluxo de dados, decidiu-se adaptar o critério tradicionald *todos-usos*, que requer que todas as associações entre uma definição de variável e seus subsequentes usos sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição. Esse critério foi revisado no contexto do teste estrutural de integração nível um de programas OO e OA levando em consideração a interface entre as unidades. Para definir o fluxo de dados entre uma unidade base com todas as outras que interagem diretamente com ela, tomou-se como base a abordagem de teste de interfaces proposta por Linnenkugel e Müllerburg (1990) e retomada por Franchin (2007) no teste par-a-par de programas Java OO e OA.

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 Müllerburg, 1990). Note que ainda está se considerando o adendo 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*);

O teste de fluxo de dados de integração nível um deve considerar somente os caminhos no $\mathcal{INIP}$ (ou relações definição-uso) que influenciam diretamente a comunicação entre as unidades afetadas e as execuções do adendo relacionadas:

- 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 ou OA consiste de unidades u_k. Para cada unidade u_{k_i} são definidos os seguintes conjuntos (note que o mesmo exemplo da seção anterior é usado para ilustrar as definições desta seção e que para simplificar a notação nos casos em que não há recursão e nem repetição de chamada de uma mesma unidade, o índice i será omitido):

- FP-IN(u_{k_i}) é o conjunto de parâmetros formais de u_{k_i} usados como entrada. Para o método `addToTable`, FP-IN(`addToTable`) = $\{symbol, symbolInfo\}$

- FP-OUT(u_{k_i}) é o conjunto de parâmetros formais de u_{k_i} usados como saída. Para o método `addToTable`, FP-OUT(`addToTable`) = $\emptyset$

- SF-IN(u_{k_i}) é o conjunto de atributos estáticos usados em u_{k_i}. Para o método `addToTable`, SF-IN(`addToTable`) = $\{numentries, tablemax, index\}$

- SF-OUT(u_{k_i}) é o conjunto de atributos estáticos definidos em u_{k_i}. Para o método `addToTable`, SF-OUT(`addToTable`) = $\{numentries\}$

Seja u_0 uma unidade chamadora e u_{k_i} uma unidade chamada. A chamada de u_{k_i} em u_0 é representada por u_{k_i0}. Para essa chamada são definidos os seguintes conjuntos:

- AP-IN(u_{k_i0}) é o conjunto de parâmetros reais usados como entrada na chamada u_{k_i0}. Considere u_0 =`addToTable` e u_1 = `lookup`, AP-IN(u_{10}) = $\{symbol\}$.

- AP-OUT(u_{k_i0}) é o conjunto de parâmetros reais usados como saída da chamada u_{k_i0}. Considere u_0 =`addToTable` e u_1 = `lookup`, AP-OUT(u_{10}) = $\emptyset$

Para descrever as relações entre parâmetros reais e seus correspondentes parâmetros formais e entre os atributos estáticos usados pelas unidades são definidos dois mapeamentos I_{k_i0} e O_{k_i0}. 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 array). Outra observação diz respeito aos atributos estáticos. Eles possuem a mesma identificação, tanto na unidade chamadora quanto na unidade chamada.

O mapeamento I_{k_i0} relaciona cada parâmetro real de entrada da chamada u_{k_i0} com o correspondente parâmetro formal de entrada da unidade chamada u_{k_i} e cada atributo estático usado como entrada, com ele mesmo.

- I_{k_i0}: AP-IN(u_{k_i0})$\bigcup$ SF-IN(u_{k_i}) $\rightarrow$ FP-IN(u_{k_i}) $\bigcup$ SF-IN(u_{k_i}), com

  - AP-IN(u_{k_i0}) $\rightarrow$ FP-IN(u_{k_i}) e SF-IN(u_{k_i}) $\rightarrow$ SF-IN(u_{k_i})

Seja o método `addToTable` a unidade chamadora (u_0) e o método `lookup` uma unidade chamada (u_1). Então:

- I_{10}: AP-IN(u_{10})$\bigcup$ SF-IN(u_j) $\rightarrow$ FP-IN(u_1) $\bigcup$ SF-IN(u_1), com

  - AP-IN(u_{10}) $\rightarrow$ FP-IN(u_1) e SF-IN(u_1) $\rightarrow$ SF-IN(u_1)

- AP-IN(u_{10}) = {symbol}
- SF-IN(u_1) = {index, tablemax, numentries}
- FP-IN(u_1) = {key}
- I_{10} : $\{< symbol, key >, < index, index >, < tablemax, tablemax >,$ $< numentries, numentries >\}$

O mapeamento O_{k_i0} relaciona cada parâmetro real de saída da chamada u_{k_i0} com o correspondente parâmetro formal de saída da unidade chamada u_{k_i} e cada atributo estático usado como saída, com ele mesmo.

- O_{k_i0}: AP-OUT(u_{k_i0})$\bigcup$ SF-OUT(u_{k_i}) $\rightarrow$ FP-OUT(u_{k_i}) $\bigcup$ SF-OUT(u_{k_i}), com
  - AP-OUT(u_{k_i0}) $\rightarrow$ FP-OUT(u_{k_i}) e SF-OUT(u_{k_i}) $\rightarrow$ SF-OUT(u_{k_i})

Assim, utilizando o mesmo exemplo acima, tem-se:

- O_{10}: AP-OUT(u_{10})$\bigcup$ SF-OUT(u_j) $\rightarrow$ FP-OUT(u_1) $\bigcup$ SF-OUT(u_1), com
  - AP-OUT(u_{10}) $\rightarrow$ FP-OUT(u_1) e SF-OUT(u_1) $\rightarrow$ SF-OUT(u_1)
- AP-OUT(u_{10}) = $\emptyset$
- SF-OUT(u_1) = $\emptyset$
- FP-OUT(u_1) = $\emptyset$

Com base nessas definições e no grafo $\mathcal{INIP}$ 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-usos* em *x*; e *p-uso(x,y)* é o conjunto de variáveis para as quais existem *p-usos* na aresta *(x,y)*. Assim, para cada unidade chamada u_{k_i} e z uma variável foram definidos os seguintes conjuntos:

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

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

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

Para a chamada u_{k_i0} foram definidos os seguintes conjuntos:

- DEF-CALLER(u_{k_i0}, 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 $u_{k_i}\}$, sendo que $z \in$ AP-IN(u_{k_i0}) ou $z \in$ SF-IN(u_{k_i}).

- C-USE-CALLER(u_{k_i0}, 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_{k_i} até *x*$\}$, sendo que $z \in$ AP-OUT(u_{k_i0}) ou $z \in$ SF-OUT(u_{k_i}).

- P-USE-CALLER(u_{k_i0}, z) = $\{(x, y) \in u_0 | z \in$ p-uso(x) e existe um caminho livre de definição com relação a *z* a partir dos nós de retorno de u_{k_i} até *(x,y)*$\}$, sendo que $z \in$ AP-OUT(u_{k_i0}) ou $z \in$ SF-OUT(u_{k_i}).

A partir das definições anteriores é definido o critério **todos-usos-integrados-N1** utilizado para derivar requisitos de teste estrutural de integração com profundidade um baseados nas interfaces entre as unidades.

- **todos-usos-integrados-N1**: Π satisfaz o critério todos-usos-integrados-N1 se:

  1. para cada $z \in$ (AP-IN$(u_{k_i0}) \bigcup$ SF-IN$(u_{k_i}))$, Π inclui um caminho livre de definição com relação a z a partir de cada nó $x \in$ DEF-CALLER(u_{k_i0}, z) até cada nó $y \in$ C-USE-CALLED$(u_{k_i}, I_{k_i0}(z))$ e até cada aresta $(x, y) \in$ P-USE-CALLED$(u_{k_i}, I_{k_i0}(z))$. 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$(u_{k_i0}) \bigcup$ SF-OUT$(u_{k_i}))$, Π inclui um caminho livre de definição com relação a z a partir de cada nó $x \in$ DEF-CALLED$(u_{k_i}, O_{k_i0}(z))$ até cada nó $y \in$ C-USE-CALLER(u_0, z) e até cada aresta $(x, y) \in$

P-USE-CALLER(u_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 nas unidades chamadas até todo uso computacional e todo uso predicativo na unidade chamadora.

3. para cada $z \in$ (AP-OUT$(u_{k_i0}) \bigcup$ SF-OUT$(u_{k_i})) \bigcap$(AP-IN$(u_{m_i0}) \bigcup$ SF-IN$(u_{m_i}))$, Π inclui um caminho livre de definição C com relação a z a partir de cada nó $x \in$ DEF-CALLED$(u_{k_i}, O_{k_i0}(z))$ até cada nó $y \in$ C-USE-CALLED$(u_{m_i}, I_{m_i0}(z))$ e até cada aresta $(x, y) \in$ P-USE-CALLED$(u_{m_i}, I_{m_i0}(z))$, com $1 \leq m \leq n$ e $\exists w \in u_j | w \in C$ e $j \neq m$. 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, incluindo a própria.

Há uma exceção nas cláusulas (2) e (3) da definição do critérios todos-usos-integrados-N1, com relação a definições dos parâmetros formais dentro das unidades chamadas e seus posteriores usos, após o retorno da chamada, na unidade chamadora. Em Java, a passagem de parâmetros é feita por valor, ou seja, se o parâmetro real for do tipo primitivo ou for um objeto da classe `String` ou de uma *wrapper class* (ver seção 3.2.2), o parâmetro formal correspondente recebe e armazena o dado do parâmetro real. Caso o parâmetro real seja do tipo de referência, o parâmetro formal correspondente recebe e armazena o endereço do objeto em memória referenciado pelo parâmetro real. Pode-se dizer que o parâmetro formal é uma cópia do parâmetro real. Assim, qualquer alteração no valor da cópia de um parâmetro real não vai afetá-lo, seja o parâmetro real do tipo primitivo ou de referência. Desse modo, caso haja um uso posterior do parâmetro real depois da chamada, não será caracterizado um par definição-uso.

O mesmo não ocorre se um parâmetro real for do tipo de referência e sua cópia definir/alterar, por meio do endereço de referência, os dados do objeto referenciado pelo parâmetro real. Nessa situação, a definição terá efeito no parâmetro real, pois o objeto que ele referencia foi modificado. Desse modo, caso haja um eventual uso do parâmetro real depois da chamada, um par definição-uso será caracterizado.

5.5.3 Exemplos

1) Sejam $u_0 = addToTable; u_1 = lookup; u_2 = addSymbol; u_3 = addInfo$

$FP - IN(u_0)$= {symbol, symbolInfo}
$FP - IN(u_1)$= {key}
$FP - IN(u_2)$= {symbol}
$FP - IN(u_3)$= {symbolInfo, index}
$FP - OUT(u_0) = \emptyset$
$FP - OUT(u_1) = \emptyset$
$FP - OUT(u_2) = \emptyset$
$FP - OUT(u_3) = \emptyset$
$SF - IN(u_0)$ ={tablemax, numentries, index}
$SF - IN(u_1)$ ={index, tablemax}
$SF - IN(u_2)$ ={index, tabSimbolos, tablemax}
$SF - IN(u_3)$ ={tabSimbolos}
$SF - OUT(u_0)$ = {numentries}
$SF - OUT(u_1)$ = {index}
$SF - OUT(u_2)$ = {index, tabSimbolos}
$SF - OUT(u_3)$ = {tabSimbolos}
$AP - IN(u_{10})$ = {symbol}
$AP - IN(u_{20})$ = {symbol}
$AP - IN(u_{30})$ = {symbolInfo, index}
$AP - OUT(u_{10}) = \emptyset$
$AP - OUT(u_{20}) = \emptyset$
$AP - OUT(u_{30}) = \emptyset$

I_{10} ={<symbol, key>, <index, index>, <tablemax, tablemax>}
I_{20} ={<symbol, symbol>, <index, index>, <tabSimbolos, tabSimbolos>}
I_{30} ={<symbolInfo, symbolInfo>, <index, index>, <tabSimbolos, tabSimbolos>}
O_{10} ={<index, index>}
O_{20} ={<index, index>, <tabSimbolos, tabSimbolos>}
O_{30} ={<tabSimbolos, tabSimbolos>}

DEF-CALLED($u_1, index$) = {1:8, 1:37, 1:16}
DEF-CALLED($u_2, index$) = {2:6, 2:25, 2:33}
DEF-CALLED($u_2, tabSimbolos$) = {2:59}
DEF-CALLED($u_3, tabSimbolos$) = {3:0}
C-USE-CALLED($u_1, symbol$) = {1:0, 1:72}
C-USE-CALLED($u_1, index$) = {1:72, 1:87, 1:42}
C-USE-CALLED($u_1, tablemax$) = $\emptyset$
C-USE-CALLED($u_2, symbol$) = {2:43, 2:12}
C-USE-CALLED($u_2, index$) = {2:43, 2:59}
C-USE-CALLED($u_2, tabSimbolos$) = {2:59}
C-USE-CALLED($u_2, tablemax$) = $\emptyset$
C-USE-CALLED($u_3, symbolInfo$) = {3:0}
C-USE-CALLED($u_3, index$) = {3:0}
C-USE-CALLED($u_3, tabSimbolos$) = {3:0}
P-USE-CALLED($u_1, symbol$) = $\emptyset$
P-USE-CALLED($u_1, index$) = {(1:16, 1:37), (1:16, 1:42), (1:58, 1:72), (1:58, 1:66)}
P-USE-CALLED($u_1, tablemax$) = {(1:16, 1:37), (1:16, 1:42)}
P-USE-CALLED($u_2, symbol$) = $\emptyset$
P-USE-CALLED($u_2, index$) = {(2:12, 2:25), (2:12, 2:33)}
P-USE-CALLED($u_2, tabSimbolos$) = $\emptyset$
P-USE-CALLED($u_2, tablemax$) = {(2:12, 2:25), (2:12, 2:33)}
P-USE-CALLED($u_3, symbolInfo$) = $\emptyset$
P-USE-CALLED($u_3, tabSimbolos$) = $\emptyset$
P-USE-CALLED($u_3, index$) = $\emptyset$

DEF-CALLER($u_{10}, symbol$) = $\emptyset$
DEF-CALLER($u_{10}, index$) = $\emptyset$
DEF-CALLER($u_{10}, tablemax$) = $\emptyset$
DEF-CALLER($u_{20}, symbol$) = $\emptyset$
DEF-CALLER($u_{20}, index$) = $\emptyset$
DEF-CALLER($u_{20}, tabSimbolos$) = $\emptyset$
DEF-CALLER($u_{20}, tablemax$) = $\emptyset$
DEF-CALLER($u_{30}, symbolInfo$) = $\emptyset$
DEF-CALLER($u_{30}, tabSimbolos$) = $\emptyset$
DEF-CALLER($u_{30}, index$) = $\emptyset$
C-USE-CALLER($u_{10}, index$) = $\emptyset$
C-USE-CALLER($u_{20}, index$) = {27}
C-USE-CALLER($u_{20}, tabSimbolos$) = $\emptyset$
C-USE-CALLER($u_{30}, tabSimbolos$) = $\emptyset$
P-USE-CALLER($u_{10}, index$) = $\emptyset$
P-USE-CALLER($u_{20}, index$) = $\emptyset$
P-USE-CALLER($u_{20}, tabSimbolos$) = $\emptyset$
P-USE-CALLER($u_{30}, tabSimbolos$) = $\emptyset$

2) Sejam u_o = `metodo1`, u_1 = adendo `before` u_2 = *metodo2* e u_3 = `metodo3`. São definidos os seguintes conjuntos:

FP-IN(u_{0_1})= {y, z}
FP-IN(u_{1_1}) = {l, m, n}
FP-IN(u_{1_2}) = {l, m, n}
FP-IN(u_{2_1}) = {a, b, c, d}
FP-IN(u_{2_2}) = {a, b, c, d}
FP-IN(u_{3_1}) = {i, j, k}
SF-IN(u_{0_1}) = {var1}
SF-IN(u_{1_1}) = {var1, var2, var3}
SF-IN(u_{1_2}) = {var1, var2, var3}
SF-IN(u_{2_1}) = ∅
SF-IN(u_{2_2}) = ∅
SF-IN(u_{3_1}) = ∅

FP-OUT(u_{0_1}) = ∅
FP-OUT(u_{1_1}) = {l, m, n}
FP-OUT(u_{1_2}) = {l, m, n}
FP-OUT(u_{2_1}) = {a, c}
FP-OUT(u_{2_2}) = {a, c}
FP-OUT(u_{3_1}) = {i, j}
SF-OUT(u_{0_1}) = {var1, var2, var3}
SF-OUT(u_{1_1}) = ∅
SF-OUT(u_{1_2}) = ∅
SF-OUT(u_{2_1}) = ∅
SF-OUT(u_{2_2}) = ∅
SF-OUT(u_{3_1}) = ∅

AP-IN(u_{0_1}) ={y, z}
AP-IN(u_{1_1}) = {x, y, z}
AP-IN(u_{1_2}) = {x, z, y}
AP-IN(u_{2_1}) = {x, y, z, w}
AP-IN(u_{2_2}) = {x, z, y, w}
AP-IN(u_{3_1}) = {x, y, z}
AP-OUT(u_{0_1}) = ∅
AP-OUT(u_{1_1}) = {x, y, z}
AP-OUT(u_{1_2}) = {x, z, y}
AP-OUT(u_{2_1}) = {x, z}
AP-OUT(u_{2_2}) = {x, y}
AP-OUT(u_{3_1}) = {x, y}

I_{0_10} = {<y, y>, <z, z>, <var1, var1>}
I_{1_10} ={<x, l>, <y, m>, <z, n>,
<var1, var1>, <var2, var2>, <var3, var3>}
I_{1_20} ={<x, l>, <z, m>, <y, n>,
<var1, var1>, <var2, var2>, <var3, var3>}
I_{2_10} ={<x, a>, <y, b>, <z, c>, <w, d>}
I_{2_20} ={<x, a>, <z, b>, <y, c>, <w, d>}

I_{3_10} ={<x, i>, <y, j>, <z, k>}
O_{0_10} = {<var1, var1>, <var2, var2>, <var3, var3>}
O_{1_10} ={<x, l>, <y, m>, <z, n>}
O_{1_20} ={<x, l>, <z, m>, <y, n>}
O_{2_10} ={<x, a>, <z, c>}
O_{2_20} ={<x, a>, <y, c>}
O_{3_10} ={<x, i>, <y, j>}

DEF-CALLED(u_{0_1}, $var1$)= {6:0}
DEF-CALLED(u_{0_1}, $var2$)= {6:0}
DEF-CALLED(u_{0_1}, $var3$)= {6:0}
DEF-CALLED(u_{1_1}, l)= {1:12, 1:29}
DEF-CALLED(u_{1_1}, m)= {1:60, 1:77}
DEF-CALLED(u_{1_1}, n)= {1:108, 1:125}
DEF-CALLED(u_{1_2}, l)= {3:12, 3:29}
DEF-CALLED(u_{1_2}, m)= {3:60, 3:77}

DEF-CALLED(u_{1_2}, n)= {3:108, 3:125}
DEF-CALLED(u_{2_1}, a)= {2:0}
DEF-CALLED(u_{2_1}, c)= {2:0}
DEF-CALLED(u_{2_2}, a)= {4:0}
DEF-CALLED(u_{2_2}, c)= {4:0}
DEF-CALLED(u_{3_1}, i)= {5:36}
DEF-CALLED(u_{3_1}, j)= {5:15}

C-USE-CALLED(u_{0_1}, y)= {6:0}
C-USE-CALLED(u_{0_1}, z)= {6:0}
C-USE-CALLED(u_{0_1}, $var1$)= {6:0}
C-USE-CALLED(u_{1_1}, l)= ∅
C-USE-CALLED(u_{1_1}, m)= {1:29, 1:60}
C-USE-CALLED(u_{1_1}, n)= {1:29, 1:77}
C-USE-CALLED(u_{1_1}, $var1$)= ∅
C-USE-CALLED(u_{1_1}, $var2$)= ∅
C-USE-CALLED(u_{1_1}, $var3$)= ∅
C-USE-CALLED(u_{1_2}, l)= ∅
C-USE-CALLED(u_{1_2}, m)= {3:29, 3:60}
C-USE-CALLED(u_{1_2}, n)= {3:29, 3:77}
C-USE-CALLED(u_{1_2}, $var1$)= ∅

C-USE-CALLED(u_{1_2}, $var2$)= ∅
C-USE-CALLED(u_{1_2}, $var3$)= ∅
C-USE-CALLED(u_{2_1}, a)= ∅
C-USE-CALLED(u_{2_1}, b)= {2:0}
C-USE-CALLED(u_{2_1}, c)= ∅
C-USE-CALLED(u_{2_1}, d)= {2:0}
C-USE-CALLED(u_{2_2}, a)= ∅
C-USE-CALLED(u_{2_2}, b)= {4:0}
C-USE-CALLED(u_{2_2}, c)= ∅
C-USE-CALLED(u_{2_2}, d)= {4:0}
C-USE-CALLED(u_{3_1}, i)= ∅
C-USE-CALLED(u_{3_1}, j)= {5:36}
C-USE-CALLED(u_{3_1}, k)= {5:15}

P-USE-CALLED(u_{0_1}, y)= ∅
P-USE-CALLED(u_{0_1}, z)= {(6:149, 6:164), (6:149, 6:177)}
P-USE-CALLED(u_{0_1}, $var1$)= ∅
P-USE-CALLED(u_{1_1}, l)= {(1:0, 1:12), (1:0, 1:29)}
P-USE-CALLED(u_{1_1}, m)= {(1:48, 1:60), (1:48, 1:77)}
P-USE-CALLED(u_{1_1}, n)= {(1:96, 1:108), (1:96, 1:125)}
P-USE-CALLED(u_{1_1}, $var1$)= {(1:0, 1:12), (1:0, 1:29)}
P-USE-CALLED(u_{1_1}, $var2$)= {(1:48, 1:60), (1:48, 1:77)}
P-USE-CALLED(u_{1_1}, $var3$)= {(1:96, 1:108), (1:96, 1:125)}
P-USE-CALLED(u_{1_2}, l)= {(3:0, 3:12), (3:0, 3:29)}
P-USE-CALLED(u_{1_2}, m)= {(3:48, 3:60), (3:48, 3:77)}
P-USE-CALLED(u_{1_2}, n)= {(3:96, 3:108), (3:96, 3:125)}
P-USE-CALLED(u_{1_2}, $var1$)= {(3:0, k:12), (2:0, 2:29)}
P-USE-CALLED(u_{1_2}, $var2$)= {(3:48, 3:60), (3:48, 3:77)}
P-USE-CALLED(u_{1_2}, $var3$)= {(3:96, 3:108), (3:96, 3:125)}
P-USE-CALLED(u_{2_1}, a)= ∅
P-USE-CALLED(u_{2_1}, b)= ∅
P-USE-CALLED(u_{2_1}, c)= ∅
P-USE-CALLED(u_{2_1}, d)= ∅
P-USE-CALLED(u_{2_2}, a)= ∅
P-USE-CALLED(u_{2_2}, b)= ∅
P-USE-CALLED(u_{2_2}, c)= ∅
P-USE-CALLED(u_{2_2}, d)= ∅
P-USE-CALLED(u_{3_1}, i)= {(5:0, 5:15), (5:0, 5:21), (5:21, 5:42), (5:21, 5:36)}
P-USE-CALLED(u_{3_1}, j)= {(5:0, 5:15), (5:0, 5:21)}
P-USE-CALLED(u_{3_1}, k)= {(5:21, 5:42), (5:21, 5:36)}

DEF-CALLER($u_{0_{1_0}}$, y)= ∅
DEF-CALLER($u_{0_{1_0}}$, z)= ∅
DEF-CALLER($u_{0_{1_0}}$, $var1$)= {0}
DEF-CALLER($u_{1_{1_0}}$, x)= {0}
DEF-CALLER($u_{1_{1_0}}$, y)= ∅
DEF-CALLER($u_{1_{1_0}}$, z)= ∅
DEF-CALLER($u_{1_{1_0}}$, $var1$)= {0}
DEF-CALLER($u_{1_{1_0}}$, $var2$)= {0}
DEF-CALLER($u_{1_{1_0}}$, $var3$)= {0}
DEF-CALLER($u_{1_{2_0}}$, x)= ∅
DEF-CALLER($u_{1_{2_0}}$, y)= ∅
DEF-CALLER($u_{1_{2_0}}$, z)= ∅
DEF-CALLER($u_{1_{2_0}}$, $var1$)= {0}
DEF-CALLER($u_{1_{2_0}}$, $var2$)= {0}
DEF-CALLER($u_{1_{2_0}}$, $var3$)= {0}
DEF-CALLER($u_{2_{1_0}}$, x)= ∅
DEF-CALLER($u_{2_{1_0}}$, y)= ∅
DEF-CALLER($u_{2_{1_0}}$, z)= ∅
DEF-CALLER($u_{2_{1_0}}$, w)= {0}
DEF-CALLER($u_{2_{2_0}}$, x)= ∅
DEF-CALLER($u_{2_{2_0}}$, y)= ∅
DEF-CALLER($u_{2_{2_0}}$, z)= ∅
DEF-CALLER($u_{2_{2_0}}$, w)= {0}
DEF-CALLER($u_{3_{1_0}}$, x)= {164}
DEF-CALLER($u_{3_{1_0}}$, y)= ∅
DEF-CALLER($u_{3_{1_0}}$, z)= ∅

C-USE-CALLER($u_{0_{1_0}}$, $var1$)= ∅
C-USE-CALLER($u_{0_{1_0}}$, $var2$)= ∅
C-USE-CALLER($u_{0_{1_0}}$, $var3$)= ∅
C-USE-CALLER($u_{1_{1_0}}$, x)= ∅
C-USE-CALLER($u_{1_{1_0}}$, y)= ∅
C-USE-CALLER($u_{1_{1_0}}$, z)= ∅
C-USE-CALLER($u_{1_{2_0}}$, x)= ∅
C-USE-CALLER($u_{1_{2_0}}$, y)= ∅
C-USE-CALLER($u_{1_{2_0}}$, z)= ∅
C-USE-CALLER($u_{2_{1_0}}$, x)= ∅
C-USE-CALLER($u_{2_{1_0}}$, z)= ∅
C-USE-CALLER($u_{2_{2_0}}$, x)= ∅
C-USE-CALLER($u_{2_{2_0}}$, y)= ∅
C-USE-CALLER($u_{3_{1_0}}$, x)= ∅
C-USE-CALLER($u_{3_{1_0}}$, y)= ∅

P-USE-CALLER($u_{0_{1_0}}$, $var1$)= ∅
P-USE-CALLER($u_{0_{1_0}}$, $var2$)= ∅
P-USE-CALLER($u_{0_{1_0}}$, $var3$)= ∅
P-USE-CALLER($u_{1_{1_0}}$, x)= ∅
P-USE-CALLER($u_{1_{1_0}}$, y)= ∅
P-USE-CALLER($u_{1_{1_0}}$, z)= ∅
P-USE-CALLER($u_{1_{2_0}}$, x)= ∅
P-USE-CALLER($u_{1_{2_0}}$, y)= ∅
P-USE-CALLER($u_{1_{2_0}}$, z)= {(149, 177), (149, 164)}
P-USE-CALLER($u_{2_{1_0}}$, x)= ∅
P-USE-CALLER($u_{2_{1_0}}$, z)= ∅
P-USE-CALLER($u_{2_{2_0}}$, x)= {(149, 177), (149, 164)}
P-USE-CALLER($u_{2_{2_0}}$, y)= ∅
P-USE-CALLER($u_{3_{1_0}}$, x)= ∅
P-USE-CALLER($u_{3_{1_0}}$, y)= ∅

Requisitos de Teste

Na Tabela 5.2 são mostrados os conjuntos de requisitos de teste derivados pelos critérios todos-nós-integrados-N1, todas-arestas-integradas-N1 e todos-usos-integrados-N1 para o exemplo. O grafo Def-Uso é apresentado na Figura 5.8. Dos conjuntos de requisitos de teste apresentados na Tabela 5.2 nota-se que: o conjunto R_n contém todos os nós integrados do grafo $\mathcal{IN1P}$; o conjunto R_e contém todas as arestas integradas; o conjunto R_u contém todos os pares de definições-usos das variáveis de comunicação. As notações *(x, i, j)* e *(x, i, (j, k))* utilizadas para representar os requisitos de R_u indicam que a variável x é definida no nó i e existe um uso computacional de x no nó j ou um uso predicativo de x na aresta *(j, k)*, respectivamente, bem como pelo menos um caminho livre de definição do nó i ao nó j ou à aresta *(j, k)*. Nesta notação, para identificar x é utilizado o nome da variável de comunicação conforme declarada na unidade chamadora.

Tabela 5.2: Requisitos de teste do exemplo 1

Critérios	**Conjuntos de Requisitos de Teste**
Todos-nós-integrados-N1	R_n = { 1:0, 1:8, 1:72, 1:80, 1:16, 1:37, 1:42, 1:50, 1:58, 1:66, 1:87, 1:93, 1:100, 1:96, 2:0, 2:6, 2:43, 2:51, 2:25, 2:33, 2:12, 2:59, 3:0}
Todas-arestas-integradas-N1	R_n = {(1:0, 1:8), (1:8, 1:72), (1:72, 1:80), (1:80, 1:16), (1:80, 1:87), (1:16, 1:37), (1:16, 1:42), (1:37, 1:42), (1:42, 1:50), (1:50, 1:58), (1:50, 1:66), (1:58, 1:66), (1:58, 1:72), (1:66, 1:93), (1:87, 1:93), (1:93, 1:96), (1:96, 1:100), (2:0, 2:6), (2:6, 2:43), (2:43, 2:51), (2:51, 2:59), (2:51, 2:12), (2:12, 2:33), (2:12, 2:25), (2:33, 2:43), (2:25, 2:43) }
Todos-usos-integrados-N1	2)u_2 R_u = {(index, 2:6, 27), (index, 2:25, 27), (index, 2:33, 27)} 3)u_2 e u_3 R_u = { (tabSimbolos, 2:59, 3:0), (tabSimbolos[], 2:59, 3:0) (index, 2:6, 3:0), (index, 2:25, 3:0), (index, 2:33, 3:0)}

5.6 Casos Especiais

5.6.1 Polimorfismo

No tipo de teste estrutural de integração pode ocorrer casos de polimorfismo da linguagem Java. Em alguns casos não é possível determinar em tempo de compilação qual dos métodos polimórficos será executado. As opções encontradas para esse caso foram:

1. Montar o grafo $\mathcal{IN1P}$ para cada possibilidade;

2. Mostrar ao usuário uma tela com todos os métodos possíveis e ele poderá selecionar um desses métodos para ser integrado;

Tabela 5.3: Requisitos de teste do exemplo 2

Critérios	**Conjuntos de Requisitos de Teste**
Todos-nos-integrados-N1	R_n = {1:0, 1:12, 1:29, 1:48, 1:60, 1:77, 1:96, 1:108, 1:125, 1:144, 2:0, 3:0, 3:12, 3:29, 3:48, 3:60, 3:77, 3:96, 3:108, 3:125, 3:144, 4:0, 5:0, 5:15, 5:21, 5:42, 5:36, 6:0, 6:84, 6:95, 6:106, 6:180, 6:191, 6:197, 6:214, 6:219, 6:112, 6:164, 6:149, 6:138, 6:127, 6:177}
Todas-arestas-integradas-N1	R_e = {(1:0, i:12), (1:0, 1:29), (1:12, 1:48), (1:29, 1:48), (1:48, 1:60), (1:48, 1:77), (1:60, 1:96),(1:77, 1:96), (1:96, 1:108), (1:96, 1:125), (1:108, 1:144), (1:125, 1:144), (3:0, 3:12), (3:0, 3:29),(3:12, 3:48), (3:29, 3:48), (3:48, 3:60), (3:48, 3:77), (3:60, 3:96), (3:77, 3:96), (3:96, 3:108),(3:72, 3:125), (3:108, 3:144), (3:125, 3:144), (5:0, 5:15), (5:0, 5:21), (5:15, 5:21), (5:21, 5:42), (5:21, 5:36), (5:36, 5:42), (6:0, 6:84), (6:84, 6:95), (6:95, 6:106) (6:106, 6:180), (6:180, 6:191), (6:191, 6:197), (6:197, 6:214), (6:197, 6:219), (6:214, 6:219), (6:180, 6:112), (6:112, 6:127), (6:127, 6:138), (6:138, 6:149), (6:149, 6:164), (6:149, 6:177), (6:164, 6:177), (6:177, 6:180)}
Todos-usos-integrados-N1	1)R_u = {(x, 0, (1:0, 1:12)), (x, 0, (1:0, 1:29)), (x,0, 1:12), (var1, 0, (1:0, 1:12)), (var1, 0, (1:0, 1:29)), (var2, 0, (1:48, 1:60)), (var2, 0, (1:48, 1:177)), (var3, 0, (1:96, 1:108)), (var3, 0, (1:96, 1:108)), (var3, 0, (1:96, 1:125)), (var1, 0, (3:0, 3:12)), (var1, 0, (3:0, 3:29)), (var2, 0, (3:48, 3:60)), (var2, 0, (3:48, 3:77)), (var3, 0, (3:96, 3:108)), (var3, 0, (3:96, 3:125)) } 2)u_{1_2} R_u = {(z, 3:60, (149, 177)), (z, 3:60, (149, 164)), (z, 3:77, (149, 177)), (z, 3:77, (149, 164)) } 2)u_{2_1} R_u = {(x, 2:0, (197, 214)), (x, 2:0, 219)} 2)u_{2_2} R_u = {(x, 4:0, (149, 177)), (x, 4:0, (149, 164)), (x, 4:0, (197, 214)), (x, 4:0, (197, 219))} 2)u_{3_1} R_u = {(x, 5:36, (197, 214)), (x, 5:36, (197, 219))} 3)u_{1_1} e u_{0_1} R_u = {(y, 1:60, 6:0), (y, 1:77, 6:0)} 3)u_{1_1} e u_{1_2} R_u = {(y, 1:60, 3:29), (y, 1:60, 3:77), (y, 1:60, (3:96, 3:125)), (y, 1:60, (3:96, 3:108)), (y, 1:77, 3:29), (y, 1:77, 3:77), (y, 1:77, (3:96, 3:108)), (y, 1:77, (3:96, 3:108))} 3)u_{1_1} e u_{2_1} R_u = {(y, 1:60, 2:0), (y, 1:77, 2:0)} 3)u_{1_1} e u_{3_1} R_u = {(y, 1:60, (5:0, 5:15)), (y, 1:60, (5:0, 5:21)), (y, 1:60, 5:36), (y, 1:77, (5:0, 5:15)), (y, 1:77, (5:0, 5:21)), (y, 1:60, 5:36) } 3)u_{1_2} e u_{0_1} R_u = {(z, 3:60, 6:0), (z, 3:60, (6:149, 6:164)), (z, 3:60, (6:149, 6:177)), (z, 3:77, 6:0), (z, 3:77, (6:149, 6:164)), (z, 3:77, (6:149, 6:177))} 3)u_{1_2} e u_{1_2} R_u = {(z, 3:60, 3:29), (z, 3:60, (3:48, 3:60)), (z, 3:60, (3:48, 3:77)), (z, 3:77, 3:29), (z, 3:77, (3:48, 3:60)), (z, 3:77, (3:48, 3:77))} 3)u_{1_2} e u_{2_2} R_u = {(z, 3:60, 4:0), (z, 3:77, 4:0))} 3)u_{1_2} e u_{3_1} R_u = {(z, 3:60, 5:15), (z, 3:60, (5:21, 5:42)), (z, 3:60, (5:21, 5:36)), (z, 3:77, 5:15), (z, 3:77, (5:21, 5:42)), (z, 3:77, (5:21, 5:36))} 3)u_{2_1} e u_{0_1} R_u = {(z, 2:0, 6:0), (z, 2:0, (6:149, 6:164)), (z, 2:0, (6:149, 6:177))} 3)u_{2_1} e u_{1_2} R_u = {(x, 2:0, (3:0, 3:12)), (x, 2:0, (3:0, 3:29)), (z, 2:0, 3:29), (z, 2:0, (3:48, 3:60)), (z, 2:0, (3:48, 3:77)), (z, 2:0, 3:60))} 3)u_{2_1} e u_{3_1} R_u = {(x, 2:0, (5:0, 5:15)), (x, 2:0, (5:0, 5:21)), (x, 2:0, (5:21, 5:42)), (x, 2:0, (5:21, 5:36)), (z, 2:0, 5:15), (z, 2:0, (5:21, 5:42)), (z, 2:0, (5:21, 5:36))} 3)u_{2_2} e u_{0_1} R_u = {(y, 4:0, 6:0)} 3)u_{2_2} e u_{1_2} R_u = {(x, 4:0, (3:0, 3:12)), (x, 4:0, (3:0, 3:29)), (y, 4:0, 3:29), (y, 4:0, 3:77), (y, 4:0, (3:96, 3:108)), (y, 4:0, (3:96, 3:125))} 3)u_{2_2} e u_{3_1} R_u = {(x, 4:0, (5:0, 5:15)), (x, 4:0, (5:0, 5:21)), (x, 4:0, (5:21, 5:42)), (x, 4:0, (5:21, 5:36)), (y, 4:0, (5:0, 5:15)), (y, 4:0, (5:0, 5:21)), (y, 4:0, 5:36)} 3)u_{3_1} e u_{0_1} R_u = {(y, 5:15, 6:0), (x, 5:36, 6:0)}

3. Mostrar ao usuário uma tela com todos os métodos possíveis e ele poderá selecionar um ou mais métodos para ser integrado. Dessa forma, o nó correspondente ao método chamado terá como filhos os nós de entrada de cada um dos métodos selecionados.

Na Figura 5.11 é apresentado um diagrama de exemplo de polimorfismo. A classe `Animal` tem como subclasses as classe `Felino`, `Canino` e `Hipopotamo`. As classes `Felino` e `Canino` sobrepõem os métodos circular de `Animal`. `Hipopotamo` sobrepõe os métodos `fazerBarulho` e `comer`. A classe `Felino` tem, por sua vez, as subclasses `Leao`, `Tigre` e `Gato`, que sobrepõem os métodos `fazerBarulho` e `comer`. Da mesma forma, a classe `Canino` tem as subclasses `Lobo` e `Cachorro` que sobrepõem os mesmos métodos. Na Figura 5.12 é apresentado um código fonte que faz uso dessas classes. Nesse código é criado um vetor de `Animal` e, em cada elemento desse vetor, é inserido um objeto de uma subclasse de `Animal`. Esse vetor é percorrido para chamar os métodos `comer` e `circular` para cada elemento.

Considerando a unidade `main` a ser testada, se a opção 1 fosse considerada, seriam gerados 21 grafos $\mathcal{IN1P}$ distintos, uma vez que são sete possibilidades para o método comer (da classe `Cachorro`, `Lobo`, `Felino`, `Gato`, `Leao`, `Hipopotamo` e `Animal`) e três para o método `circular` (da classe `Canino`, `Felino` e `Animal`). É fácil notar que a desvantagem dessa opção é que, dependendo do número de possibilidades, geraria-se vários grafos, o que poderia tornar a atividade de teste impraticável nesse caso. Levando em conta a opção 2, seria gerado apenas um grafo e o usuário teria que pedir a geração de todos os casos adicionais que quisesse testar. No entanto, também podem existir vários casos adicionais que deveriam ser testados. A opção escolhida foi a 3, embora a implementação seja mais difícil se comparada às outras duas. Nessa opção, conforme pode ser visto na Figura 5.13, apenas um grafo é montado com todas as possibilidades. O usuário ainda poderá desmarcar os métodos que ele achar desnecessários. Por exemplo, o método comer da classe `Animal` nunca será executado no código apresentado na Figura 5.12. O usuário poderia então desmarcá-lo e ele não seria integrado. Outra vantagem dessa opção é que pode ocorrer que uma variável de referência seja definida em um método polimórfico e usada em outro, gerando então um requisito de teste para o critério todos-usos-integrados-N1. Considerando as duas primeiras opções, esse requisito não seria gerado.

5.6.2 Uso de `around` com o Comando `proceed`

Um adendo do tipo `around` substitui a execução do método entrecortado pela execução do próprio adendo. Esse método só será executado caso exista uma chamada a ele via o comando

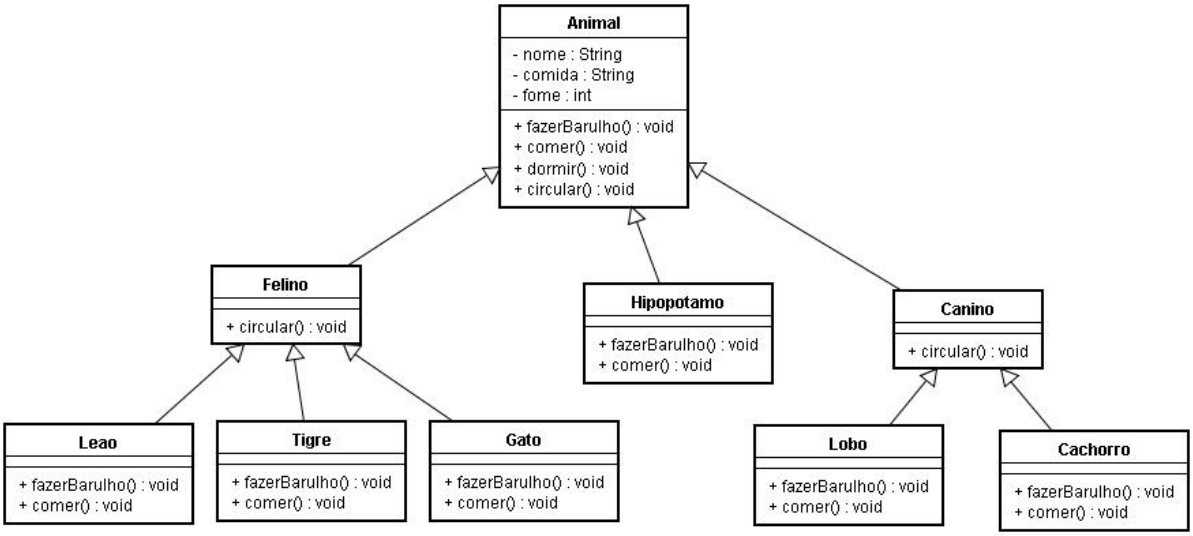

Figura 5.11: Diagrama do exemplo do polimorfismo

```
public class Main {
  public static void main(String args[]){
    Animal[] animais = new Animal[5];

    animais[0] = new Cachorro();
    animais[1] = new Gato();
    animais[2] = new Lobo();
    animais[3] = new Hipopotamo();
    animais[4] = new Leao();

    for (int i=0; i< animais.length; i++){
      animais[i].comer();
      animais[i].circular();
    }
  }
}
```

Figura 5.12: Código fonte do algoritmo de exemplo do polimorfismo.

"`proceed`"no corpo do adendo do tipo `around`. Conforme explicado na Seção 3.3.2, é possível observar, por meio da análise do *bytecode*, que não se trata de uma chamada no nível um e a priori determinar que esse caso não deveria ser integrado. Por outro lado, essa solução estaria semanticamente incorreta visto que o método entrecortado está no nível um. Dessa forma, há duas possibilidades a serem levadas em conta: (a) considerar essa chamada como de nível 2 e, desse modo, não construir o grafo desse método; (b) construir o grafo do método chamado sempre que houver uma chamada ao método `proceed`. Para explicar melhor as possibilidades, considere-se o código-fonte da Figura 5.14. Se a opção (a) fosse considerada, o grafo $\mathcal{INIP}$ seria criado conforme mostrado na Figura 5.15, em que os nós correspondentes ao método `proceed` (2:50 e 2:79) são representados como nós de chamada e o grafo $\mathcal{AODU}$ do método entrecortado não foi integrado. A vantagem dessa opção é a facilidade para implementação. No

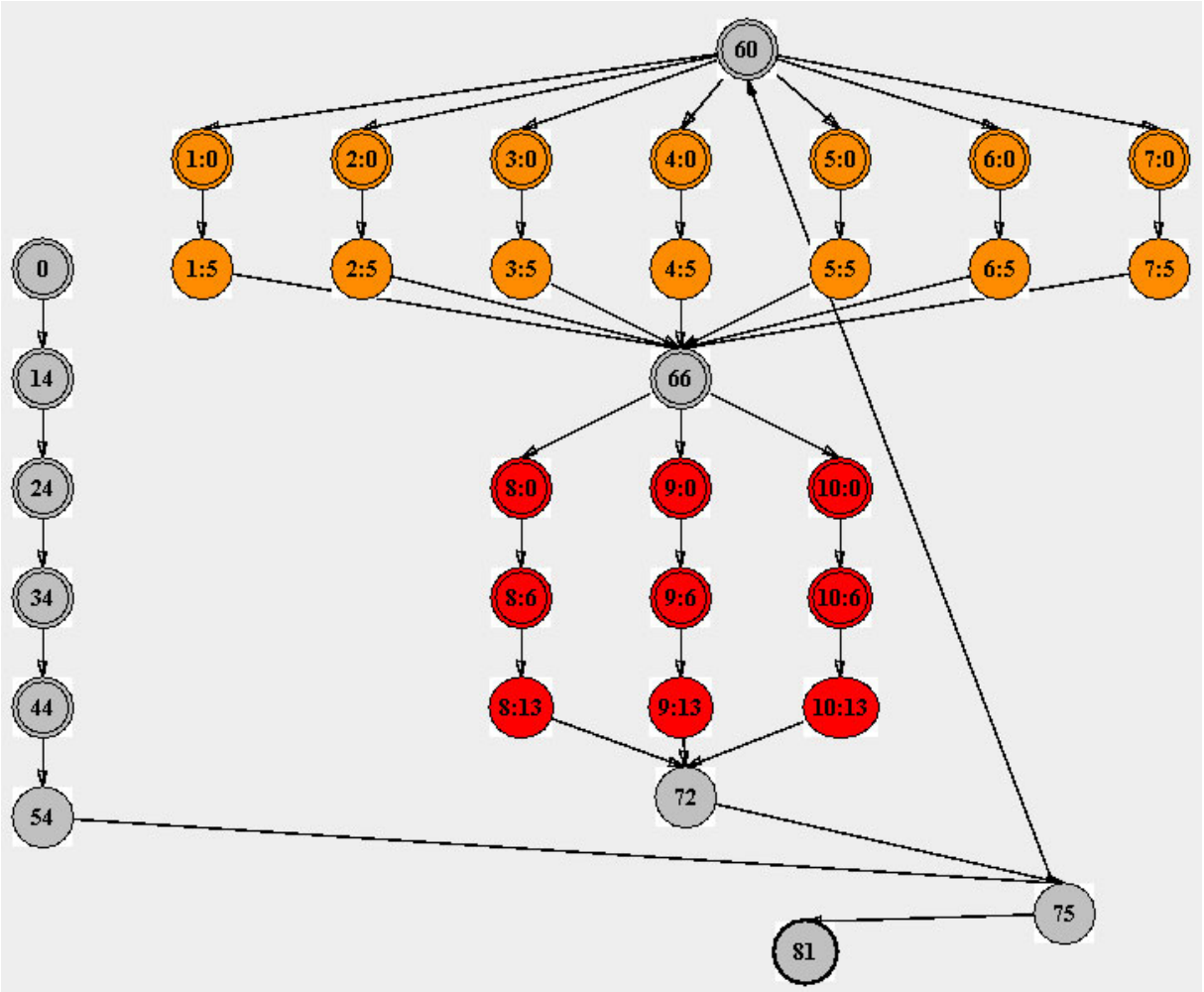

Figura 5.13: Grafo $\mathcal{INIP}$ para o método `main` do exemplo

entanto, decidiu-se por adotar a opção (b), integrando o grafo do método entrecortado logo após a chamada ao método `proceed`, como pode ser visto na Figura 5.16, em que os nós (1:50 e 1:79) são os nós correspondentes às chamadas ao método `proceed` e os blocos em destaque são os grafos $\mathcal{AODU}$ do método entrecortado(`ordenarArray`). A razão da escolha é que se considerou essa opção semanticamente correta e mais útil para o testador.

5.7 Considerações Finais

Neste capítulo foi apresentada uma abordagem de teste estrutural de integração nível um para programas OO e OA. Foi definido o grafo $\mathcal{INIP}$ para representar o fluxo de execução entre as unidades. Ele é uma abstração formada pela integração dos grafos $\mathcal{AODU}$ da unidade chamadora com as unidades chamadas. Foram propostos três critérios específicos para derivar

```
public class Principal {
  public static boolean estaOrdenado;
  public static boolean ativarOrdenacao = true;

  public static double calcularMediaPriUltElem(int[] pA){
    int i = 0;
    int n = pA.length - 1;
    ordenarArray(pA, i , n);
    double d = (double) (pA[i]+pA[n])/2;
    return d;
  }

  public static void ordenarArray(int[] pA, int pI, int pN){
    if (ativarOrdenacao){
      for (int j = pI; j <= pN -1; j++){
        for (int k = j+1; k<=pN; k++){
          if (pA[j]> pA[k]) {
            int aux = pA[j];
            pA[j] = pA[k];
            pA[k] = aux;
          }
        }
      }
    }
  }
}
```

```
public aspect Verificacao {
  pointcut pcOrdenar (int[] aA, int aI, int aN):
    call(* *.ordenarArray(..)) && args(aA, aI, aN);

  void around (int[] aA, int aI, int aN):
    pcOrdenar(aA, aI, aN){

    boolean arrayOrdenado = true;
    for (int i = 0; i <aA.length -1 &&
       arrayOrdenado; i++){
      if (aA[i] > aA[i+1])
        arrayOrdenado = false;
    }
    proceed(aA, aI, aN);
    Principal.estaOrdenado = arrayOrdenado;
    if (!arrayOrdenado)
      Principal.ativarOrdenacao = true;
    else
      Principal.ativarOrdenacao = false;
    proceed(aA, aI, aN);
  }
}
```

Figura 5.14: Exemplo de uso do `around` com o comando `proceed`.

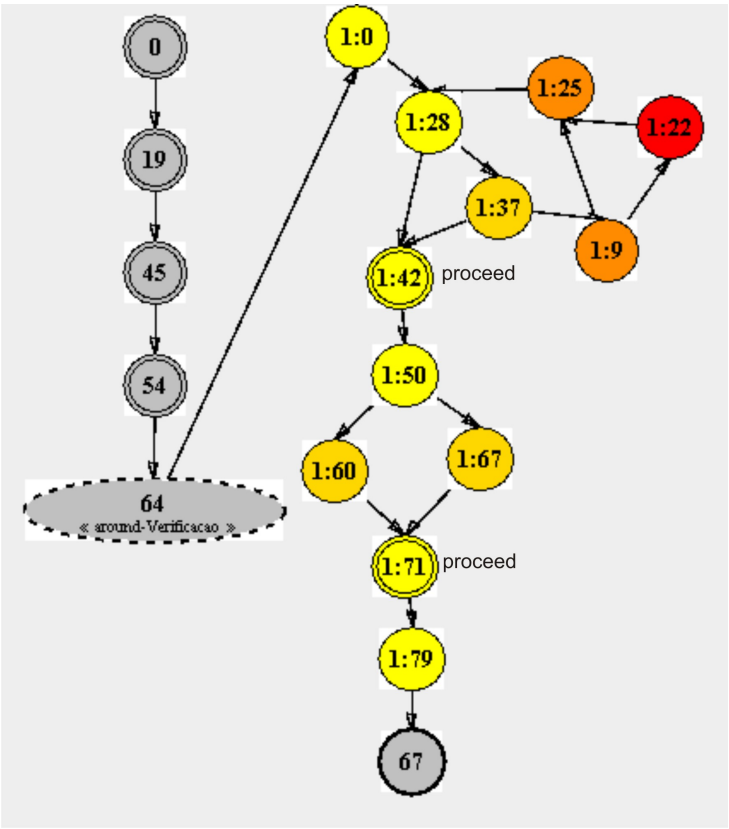

Figura 5.15: Grafo $\mathcal{INIP}$ sem integrar o método entrecortado pelo adendo `around` com o comando `proceed`

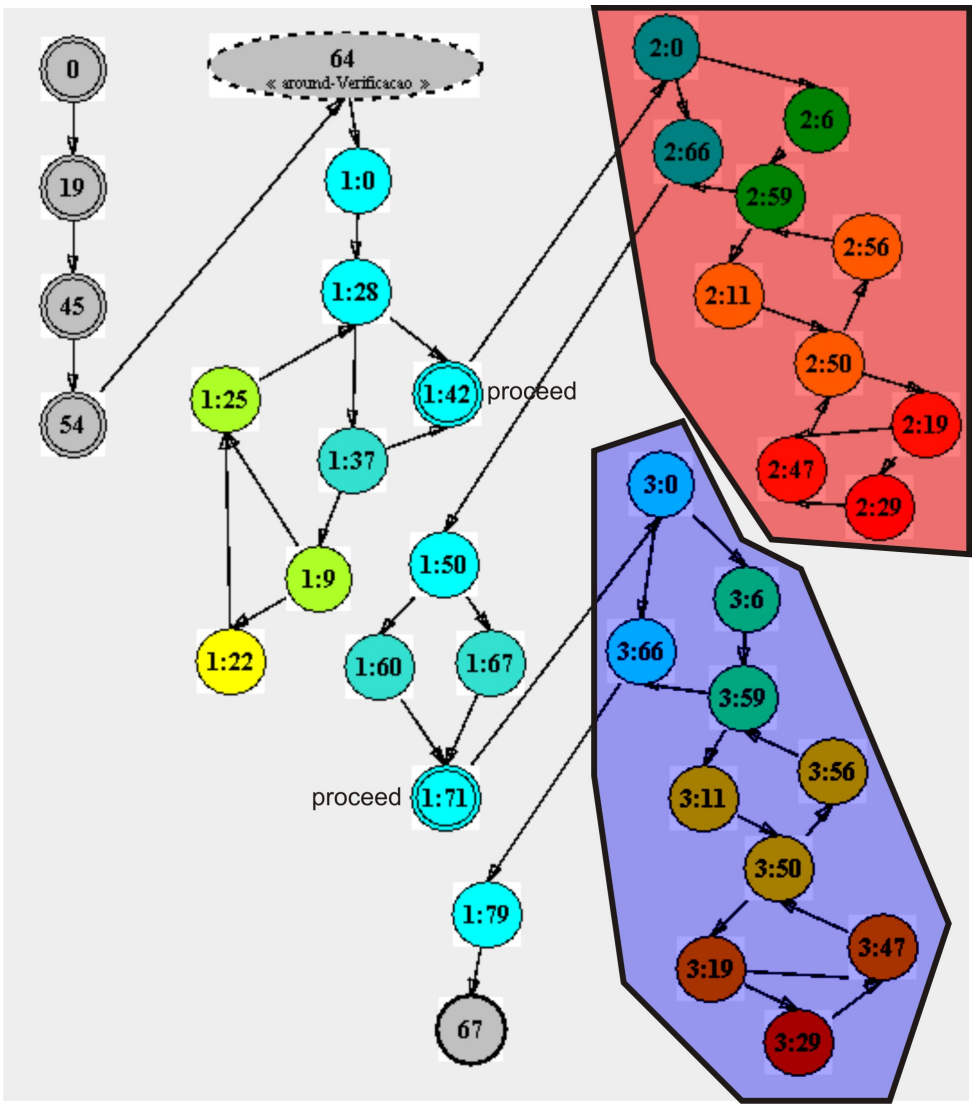

Figura 5.16: Grafo $\mathcal{INIP}$ integrando o método entrecortado pelo adendo `around` com o comando `proceed`

requisitos de teste para as unidades. Dentre eles, dois critérios são baseados em fluxo de controle: todos-nós-integrados-N1 e todas-arestas-integradas-N1; e um critério é baseado em fluxo de dados: todos-usos-integrados-N1. Por fim, foram discutidos os casos especiais desta abordagem.

Após a definição da abordagem de teste, é interessante implementá-la em uma ferramenta de teste. A implementação da abordagem proposta é apresentada no próximo capítulo.

CAPÍTULO

6

Implementação do Teste de Integração Contextual na Ferramenta JaBUTi/AJ

6.1 Considerações Iniciais

A atividade de teste pode ser muito custosa e sujeita a erros se realizada manualmente. Assim, as ferramentas de teste podem auxiliar a automatizar e tornar mais eficiente essa tarefa. O grupo de testes do ICMC/USP tem trabalhado com o teste estrutural de programas OO e OA e, nesse sentido, foram feitas propostas para o teste de unidades (métodos e adendos) (Lemos, 2005) e para o teste de integração par-a-par entre métodos e adendos (Franchin, 2007) e desenvolvida a ferramenta JaBUTi/AJ para apoiar essas propostas. Nesse contexto, é importante implementar a abordagem proposta no Capítulo 5 nessa ferramenta, evoluindo, então, para um ambiente mais completo de teste estrutural de programas OO e OA.

Este capitulo está organizado da seguinte forma: na Seção 6.2 são discutidos alguns aspectos da implementação da extensão da ferramenta realizada como parte deste trabalho; na Seção 6.3 é discutida a otimização realizada na ferramenta; na Seção 6.4 é apresentado um exemplo de uso da ferramenta; na Seção 6.5 é discutida uma possível estratégia de uso da ferramenta, com a extensão realizada e, por fim, na Seção 6.6 são apresentadas as considerações finais do capítulo.

6.2 Extensão da Ferramenta JaBUTi/AJ

A ferramenta JaBUTi/AJ foi estendida para dar suporte à abordagem proposta nesta dissertação. Primeiramente foi desenvolvido o ambiente para o teste de integração nível um. Esse ambiente utiliza os mesmos módulos selecionados para serem testados no ambiente de teste de unidade — o ambiente inicial da JaBUTi/AJ — e, a partir desses módulos, são identificadas todas as unidades que fazem ao menos uma chamada ou são afetadas por ao menos um adendo. Notar que se a unidade fizer apenas uma chamada e não é afetada por nenhum adendo ou é afetada por apenas um adendo mas não faz nenhuma chamada, o teste de integração nível um, nesse caso, é idêntico ao teste de integração par-a-par.

O ambiente para o teste de integração nível um possui seus próprios dados de teste. Dessa forma, todas as atividades realizadas nesse ambiente não interferirão nas demais. Por exemplo, a execução de um caso de teste importado no ambiente de teste de integração nível um não afetará as informações dos demais ambientes de teste e vice-versa.

O ambiente de teste de integração nível um apóia as seguintes atividades: checagem dos requisitos derivados para cada critério, importação dos casos de teste do JUnit, análise da cobertura obtida pelos casos de teste importados e visualização do grafo integrado para cada unidade.

As classes que implementam a ferramenta estão agrupadas em 13 pacotes, sete dos quais (`criteria`, `gvf`, `graph`, `lookup`, `metrics`, `project` e `verifier`) são responsáveis pela realização da análise estática, implementação dos critérios de teste e avaliação da cobertura. Os demais, como por exemplo o pacote `gui`, estão relacionados com a implementação da interface gráfica e com a coleta e armazenamento das informações de execução (Vincenzi, 2004). Para a extensão da ferramenta JaBUTi/AJ foram criadas as classes `AllN1Edges`, `AllN1Nodes` e `AllN1Uses` do pacote `criteria`, as classes `Unit1NData`, `Units1N`, do pacote `project` e a classe `DialogIntegrationPolymorphismSel`, do pacote `gui`. Além disso, as seguintes classes foram modificadas: `ClassFile`, `ClassIntegrationData`, `ClassMethod`, `DefUseIntegrationManager`, `IntegrationParameters`, `TestCase`, `TestingType` e `TestSet`, do pacote `project`; `CFG` e `CFGNode`, do pacote `graph`; `InstructionNode`, do pacote `verifier` e `DialogIntegrationSelection`, `JabutiGui`, `TableSorterPanel` e `WeightColor`, do pacote `gui`.

A Figura 6.1 apresenta uma visão simplificada da arquitetura da ferramenta JaBUTi/AJ e a interação entre o testador e a ferramenta, ilustrando a sequência de execução do ponto de vista do teste de integração nível um. Essa sequência possui os passos descritos em seguida.

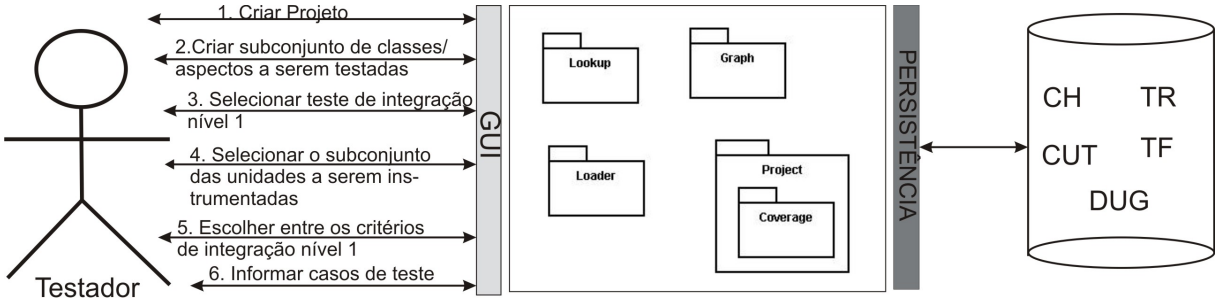

Figura 6.1: Sequência de execução da JaBUTi/AJ

Primeiramente, o testador cria um projeto de teste (um arquivo .jbt) e informa o nome e a localização do arquivo .class (CF) correspondente a classe/aspecto que ele deseja que seja classe/aspecto base da aplicação. Tendo esse arquivo como entrada, o módulo `Lookup` é utilizado e produz como saída toda a hierarquia de classes (CH) necessária para executar a classe/aspecto base, incluindo tanto classes do sistema quanto classes/aspectos definidos pelo usuário. O testador poderá selecionar um subconjunto das classes/aspectos definidos pelo usuário (programador), caracterizando as classes/aspectos a serem testados (CUTs). Tanto CH quanto CUTs são então armazenadas em uma base de dados do projeto. Em seguida, a ferramenta utiliza o módulo `Graph` para a construção dos grafos def-uso (DUG) de cada método/adendo (por padrão, inicialmente são gerados todos os grafos de unidade).

O próximo passo é selecionar a opção de teste de integração nível um. A ferramenta então utiliza o módulo `Project` para descobrir todas as unidades que fazem ao menos uma chamada a um método (ou são afetadas por ao menos um adendo) e as exibe ao testador que, por sua vez, poderá selecionar um subconjunto dessas unidades (U1N) para serem instrumentadas. Novamente, o módulo `Graph` é utilizado, mas dessa vez para gerar os grafos $\mathcal{INIP}$ dos U1N. Assim, os requisitos de teste para os critérios propostos são calculados.

O testador informa os casos de teste e o módulo `Loader`, que é o carregador de classes (`Class Loader`) da JaBUTi/AJ, instrumenta as CUTs, carrega as classes/aspectos instrumentados para serem executados e armazena as informações de execução em um arquivo de trace (TF) (um arquivo com extensão .trc). Cada execução do carregador de classes/aspectos da JaBUTi/AJ corresponde a um novo caso de teste adicionado.

O módulo `Coverage` utiliza as informações produzidas pelos outros módulos, incluindo o conjunto de requisitos de teste (TR) e o arquivo de trace (TF), para identificar o conjunto de requisitos cobertos a partir da execução dos casos de teste armazenados em TF. Ele também é responsável pela geração dos diferentes relatórios de teste, os quais são utilizados pelo testador para avaliar a qualidade do conjunto de teste e decidir quando parar os testes.

A versão intermediária desta ferramenta, incluindo os trabalhos de Lemos (2005) (teste de unidade), de Franchin (2007) (teste de integração par-a-par), de Lemos (2009) (teste baseado em pontos de junção) e este trabalho, de teste de integração nível um passará a ser chamada de JaBUTi/PC-IN1-AJ.

6.3 Otimização do Grafo $\mathcal{IN}1\mathcal{P}$

Para os casos em que não há desvio condicional em uma unidade chamada, ou seja, trata-se de uma unidade linear, o usuário tem a opção de, por meio de parametrização da JaBUTi/AJ, representar o grafo $\mathcal{AODU}$ dessa unidade por apenas um nó, uma vez que para cobrir os demais nós/arestas de unidades desse tipo bastaria cobrir apenas esse nó, que corresponde ao grafo $\mathcal{AODU}$ otimizado, e contém todas as informações de definição e uso de variáveis do grafo $\mathcal{AODU}$ original. Essa otimização deixa o grafo menos complexo e evita a criação de requisitos de teste desnecessários, o que facilita para o usuário.

Considere-se como exemplo o grafo apresentado na Figura 5.13. Todas as unidades integradas desse grafo tratam-se de unidades lineares, que com a otimização, passa a ser representado como na Figura 6.2

6.4 Exemplo de Uso da Ferramenta JaBUTi/AJ

Para ilustrar o uso da ferramenta, considere-se o exemplo 1 do Capítulo 5, que ilustra a integração de método com método (intraclasse) e de método com adendo (interclasse). Seguindo os passos apresentados na seção anterior, a primeira etapa é a criação do projeto de teste. A Figura 6.3 apresenta a tela da ferramenta para esse passo. A classe `SymbolTable` foi selecionada como classe base.

Após selecionar o ambiente para o teste de integração nível um, é exibida uma tela contendo todas as unidades que chamam ao menos um método ou são afetadas por ao menos um aspecto, como pode ser visto na Figura 6.4. Para o exemplo, foi selecionado para teste apenas o método `addToTable`. O grafo $\mathcal{IN}1\mathcal{P}$ desse método é apresentado na Figura 6.5. Os nós com um prefixo (iniciados com um número e seguidos por ":") são os grafos $\mathcal{AODU}$ de cada uma das unidades chamadas que foram integrados ao $\mathcal{AODU}$ do método `addToTable` (a unidade chamadora). Cada prefixo está relacionado a uma unidade chamada: os nós com prefixo 1 correspondem ao método `lookup`, com prefixo 2 correspondem ao método `addSymbol` e o nó com prefixo 3 corresponde ao método `addInfo`. Esse grafo está exibindo informações quanto ao critério todos-nós-integrados-N1. Os requisitos de teste calculados para o critérios

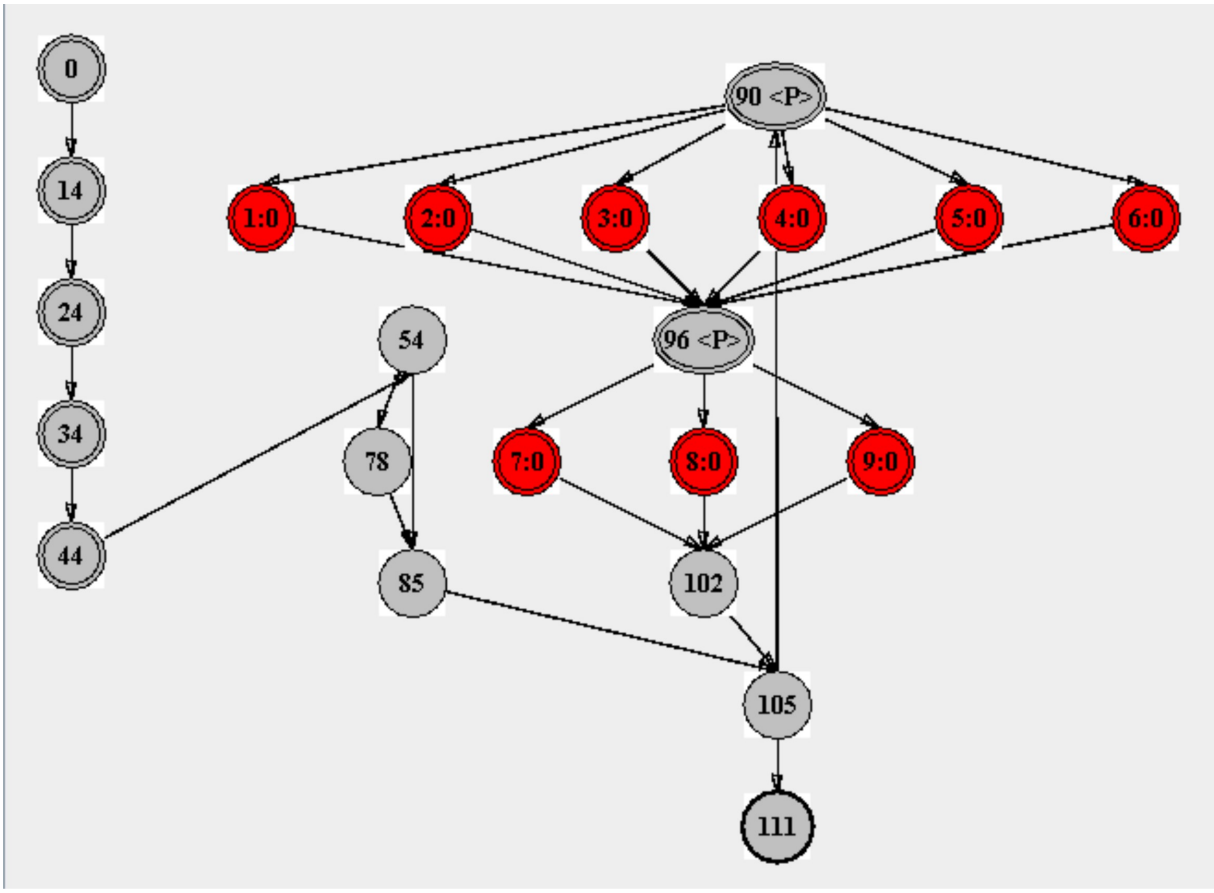

Figura 6.2: Grafo $\mathcal{INIP}$ para o método `main` do exemplo

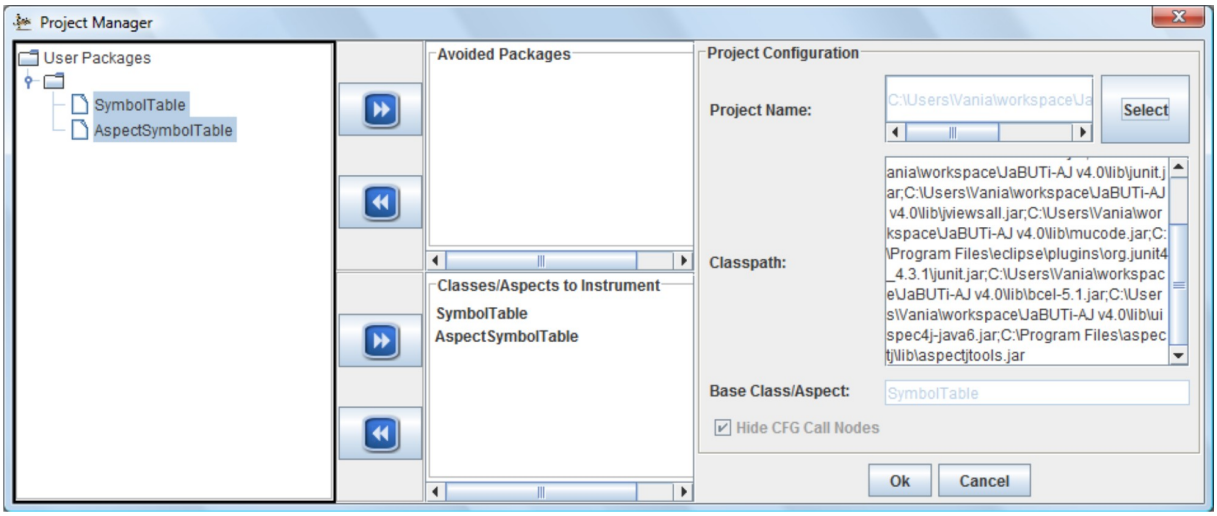

Figura 6.3: Tela do projeto de teste da ferramenta JaBUTi/AJ

todos-nós-integrados-N1, todas-arestas-integradas-N1 e todos-usos-integrados-N1 são apresentados, respectivamente, nas Figuras 6.6, 6.7 e 6.8.

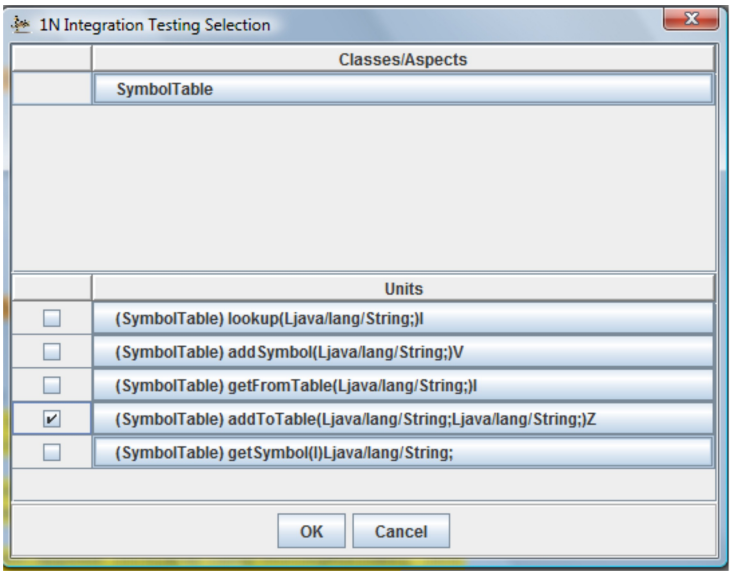

Figura 6.4: Tela de seleção das unidades a serem instrumentadas

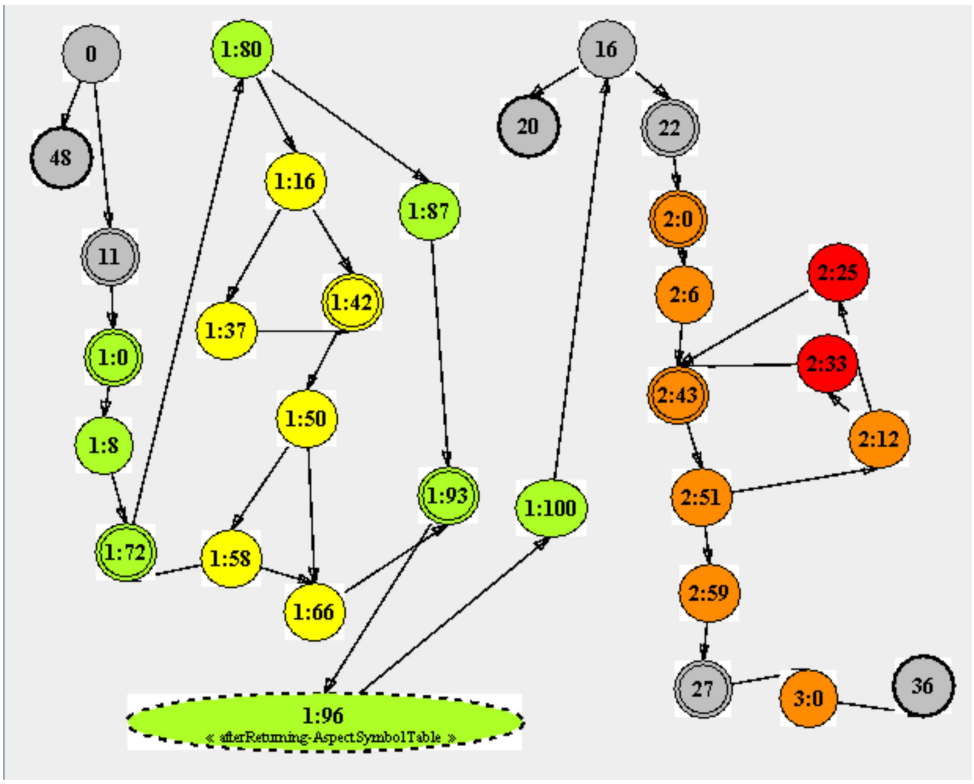

Figura 6.5: Grafo $\mathcal{INIP}$ do método `addToTable`

O próximo passo, então, é importar o conjunto de casos de teste. A execução de um caso de teste percorre um determinado caminho no fluxo do programa. A partir do caminho percorrido, a ferramenta verifica quais os nós, as arestas e os pares def-uso que foram exercitados. A classe

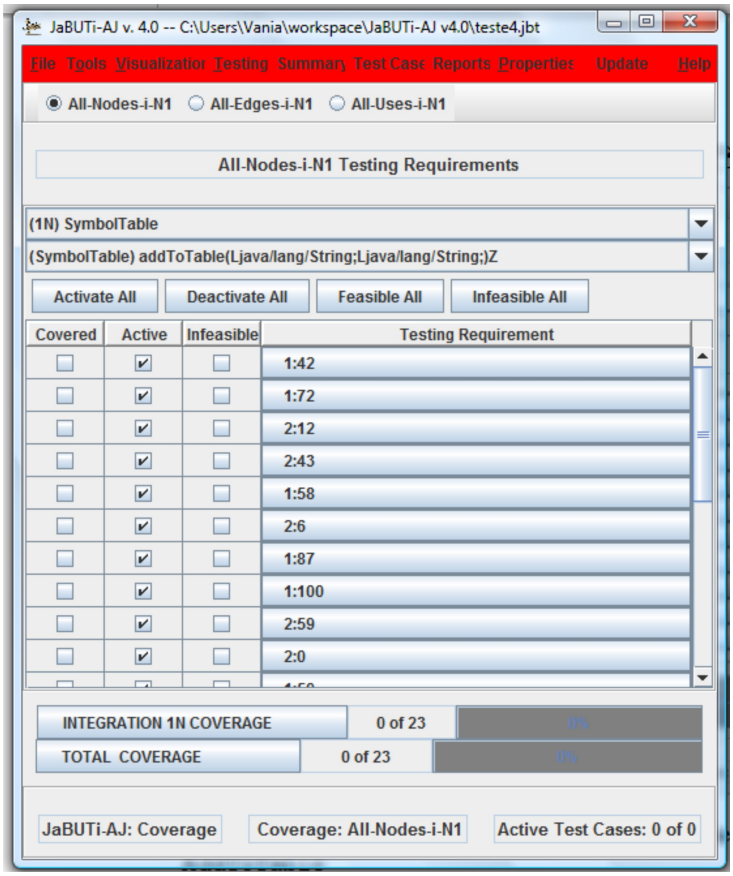

Figura 6.6: Requisitos de teste para o critério todos-nos-integrados-N1 tendo como base o método `addToTable`

`SymbolTableTestCase` escrita em JUnit e apresentada na Figura 6.9 foi testada com dois casos de teste que simulam a inserção de elementos na tabela de símbolos. Na Figura 6.10 é exibida a tela de importação dos casos de teste na ferramenta.

O caso de teste `testSymbolTable1` insere um elemento não repetido enquanto que o caso de teste `testSymbolTable2` insere um elemento repetido. O caso de teste `testSymbolTable1` é executado primeiro e, em seguida, o caso de teste `testSymbolTable2` é executado com a finalidade de cobrir os requisitos que ainda não foram cobertos. Esses casos de teste percorrem os caminhos apresentados na Tabela 6.1 do grafo $\mathcal{INIP}$. Com a execução desses casos de teste foi possível obter 86% de cobertura para o critério todos-nós-integrados-N1, cobrindo 20 de 23 requisitos de teste, como pode ser visto na Figura 6.11, 76% de cobertura para o critério todas-arestas-integradas-N1, cobrindo 20 de 26 requisitos, como pode ser visto na Figura 6.12 e 50% de cobertura para o critério todos-usos-integrados-N1, cobrindo 6 de 8 requisitos, como pode ser visto na Figura 6.13.

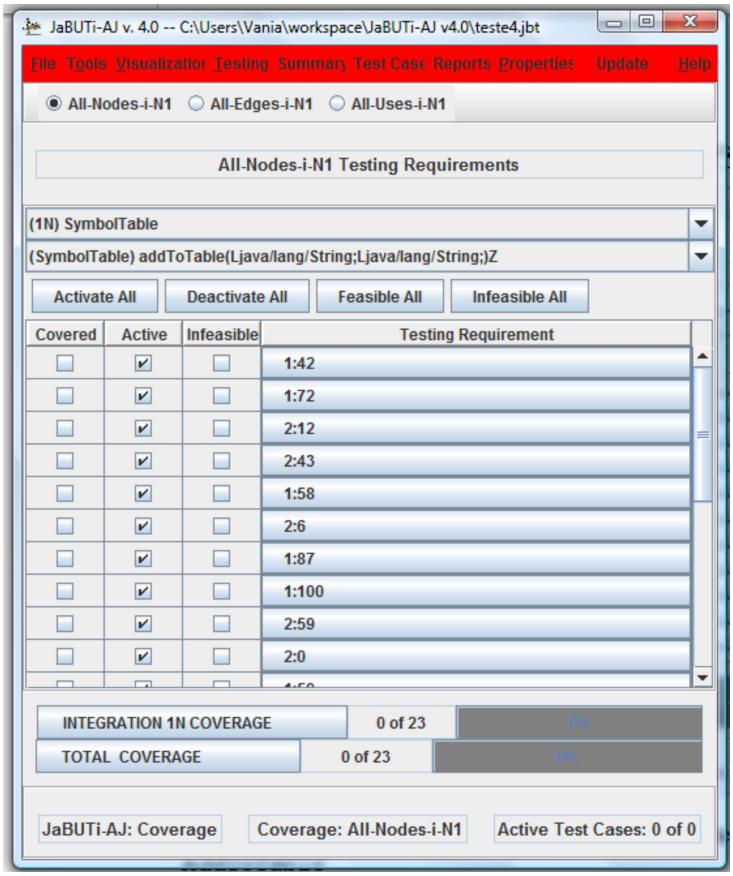

Figura 6.7: Requisitos de teste para o critério todas-arestas-integradas-N1 tendo como base o método `addToTable`

Tabela 6.1: Caminhos percorridos pelos casos de teste

Caso de Teste	Caminhos Percorridos
testSymbolTable1	{(0, 11, 1:0, 1:8, 1:72, 1:80, 1:16, 1:42, 1:50, 1:66, 1:93, 1:96, 1:100, 16, 22, 2:0, 2:6, 2:43, 2:51, 2:59, 27, 3:0, 36), (0, 11, 1:0, 1:8, 1:72, 1:80, 1:16, 1:37, 1:42, 1:50, 1:66, 1:93, 1:96, 1:100, 16, 22, 2:0, 2:6, 2:43, 2:51, 2:59, 27, 3:0, 36), (0, 11, 1:0, 1:8, 1:72, 1:80, 1:16, 1:42, 1:50, 1:66, 1:93, 1:96, 1:100, 16, 22, 2:0, 2:6, 2:43, 2:51, 2:59, 27, 3:0, 36), (0, 11, 1:0, 1:8, 1:72, 1:80, 1:16, 1:42, 1:50, 1:58, 1:72, 1:80, 1:16, 1:42, 1:50, 1:58, 1:72, 1:80, 1:16, 1:37, 1:42, 1:50, 1:66, 1:93, 1:96, 1:100, 16, 22, 2:0, 2:6, 2:43, 2:51, 2:59, 27)}
testSymbolTable1	{(0, 11, 1:0, 1:8, 1:72, 1:80, 1:16, 1:42, 1:50, 1:66, 1:93, 1:96, 1:100, 16, 22, 2:0, 2:6, 2:43, 2:51, 2:59, 27, 3:0, 36),(0, 11, 1:0, 1:8, 1:72, 1:80, 1:16, 1:37, 1:42, 1:50, 1:66, 1:93, 1:96, 1:100, 16, 22, 2:0, 2:6, 2:43, 2:51, 2:59, 27, 3:0, 36)}

Os grafos são utilizados para analisar a cobertura quanto aos critérios. Para o critério todos-nós-integrados-N1, por exemplo, os nós integrados cobertos aparecem no grafo com a cor branca. A partir da análise do grafo, do *bytecode* e do código-fonte é possível gerar novos casos de teste, com foco nos requisitos não cobertos, até se obter a cobertura desejada. Por exemplo, os nós não cobertos correspondem as linhas 54 e 55 do código-fonte, ou seja, para

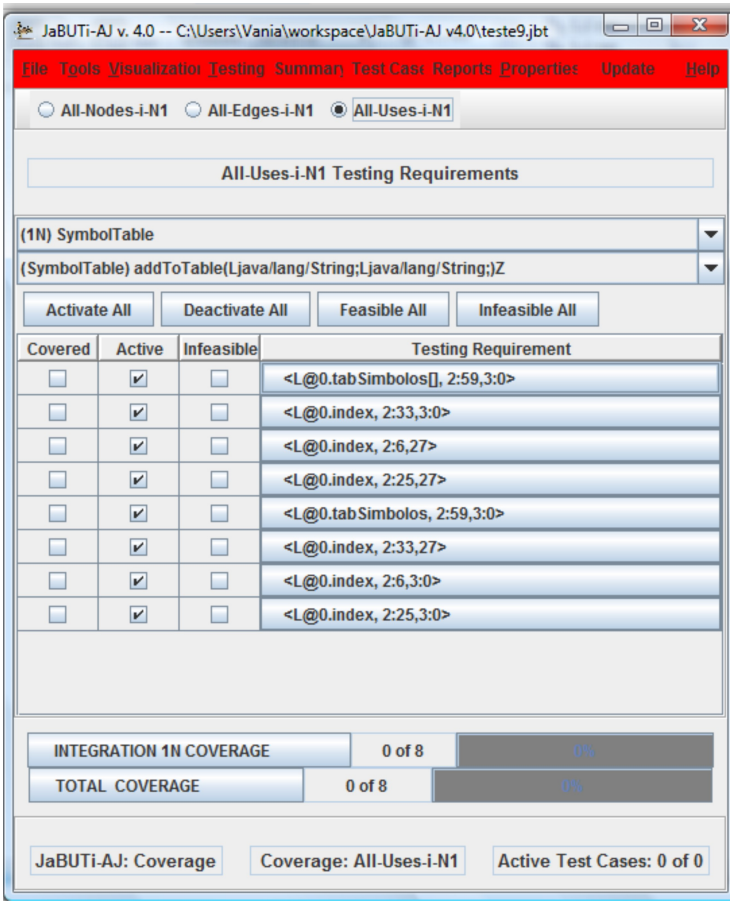

Figura 6.8: Requisitos de teste para o critério todos-usos-integrados-N1 tendo como base o método `addToTable`

cobrir esses nós é necessário um caso de teste em que a função `hash` retorne uma posição já ocupada por outro elemento. Adicionando um caso de teste com essa condição, obtém-se 100% de cobertura para o critério todos-nós-integrados-N1, 96% de cobertura para o critério todas-arestas-integradas-N1, cobrindo 25 de 26 requisitos e 75% de cobertura para o critério todos-usos-integrados-N1, cobrindo 6 de 8 requisitos. A aresta não coberta foi a (1:58, 1:66), que corresponde, no código-fonte, ao trecho `index == saveindex`. A análise do código mostra que esse trecho nunca será executado quando o método `lookup` for chamado por `addToTable`, pois `saveindex` só será igual a `index` quando `index` for igual a 0, que, por sua vez, só ocorrerá quando `index` for igual a `tablemax`. Ou seja, essa situação só irá ocorrer quando o vetor de símbolos estiver completo. Dessa forma, é possível marcar na ferramenta que esse requisito é *infeasible* (não executável) e, assim, obter 100% de cobertura para o critério todas-arestas-integradas-N1.

Com isso conseguiu-se obter cobertura total (100%) para os critérios todos-nós-integrados-N1 e todas-arestas-integradas-N1. O testador pode, então, continuar gerando mais casos de testes

```
public class SymbolTableTestCase extends TestCase implements Test{
  public SymbolTableTestCase() {
    super();
  }

  public SymbolTableTestCase(String name) {
    super(name);
  }

  public void testSymbolTable1(){
    SymbolTable st = new SymbolTable(3);
    st.addToTable("Simbolo1", "Info Simbolo 1");
    st.addToTable("Simbolo2", "Info Simbolo 2");
    assertEquals(true, st.addToTable("teste", "Info teste"));
  }

  public void testSymbolTable2(){
    SymbolTable st = new SymbolTable(3);
    st.addToTable("Simbolo1", "Info Simbolo 1");
    st.addToTable("Simbolo2", "Info Simbolo 2");
    assertEquals(false, st.addToTable("Simbolo1", "Info Simbolo 1"));
  }
}
```

Figura 6.9: Conjunto de casos de teste para o exemplo.

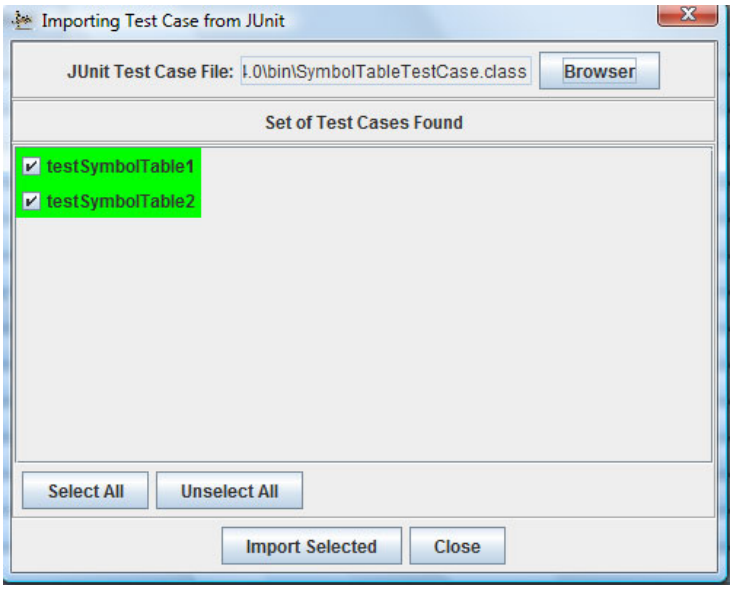

Figura 6.10: Tela de importação de casos de teste da ferramenta

para cobrir os requisitos todos-usos-integrados-N1 ou parar de gerar casos de testes e concluir que o teste de profundidade um, empregando a abordagem de teste estrutural de integração nível um aqui proposta, foi realizado com sucesso. A partir daí, o testador pode utilizar outras técnicas de teste (por exemplo, testes funcionais) com a finalidade de descobrir defeitos que ainda possam estar presentes no programa.

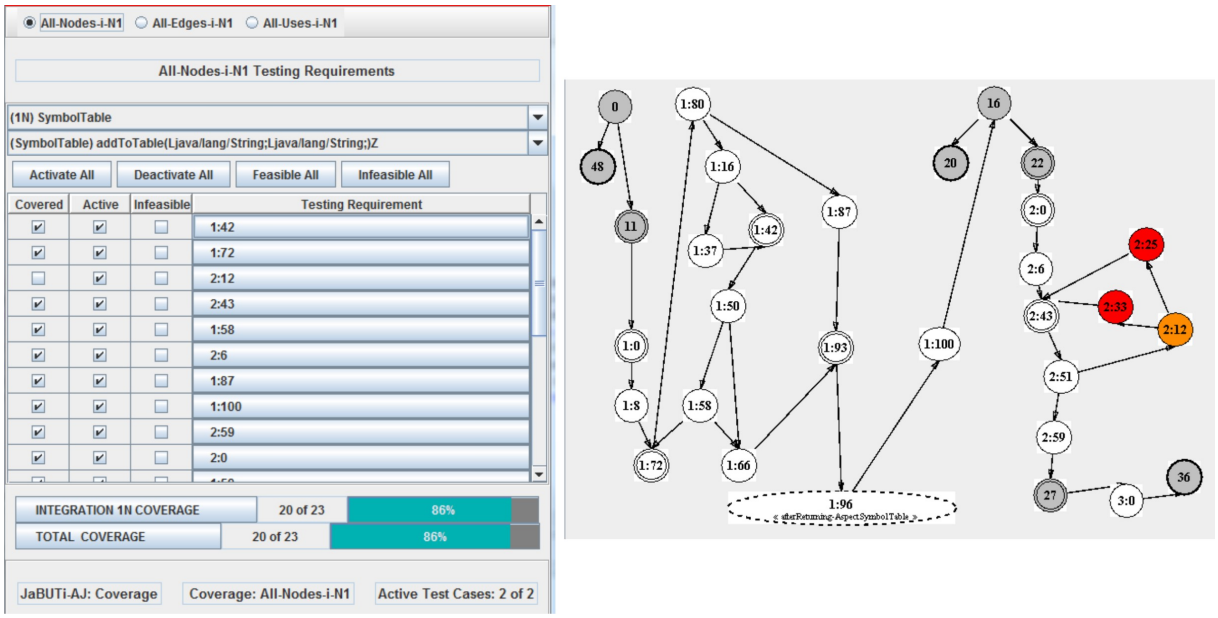

Figura 6.11: Cobertura para o critério todos-nos-integrados-N1 tendo como base o método `addToTable`

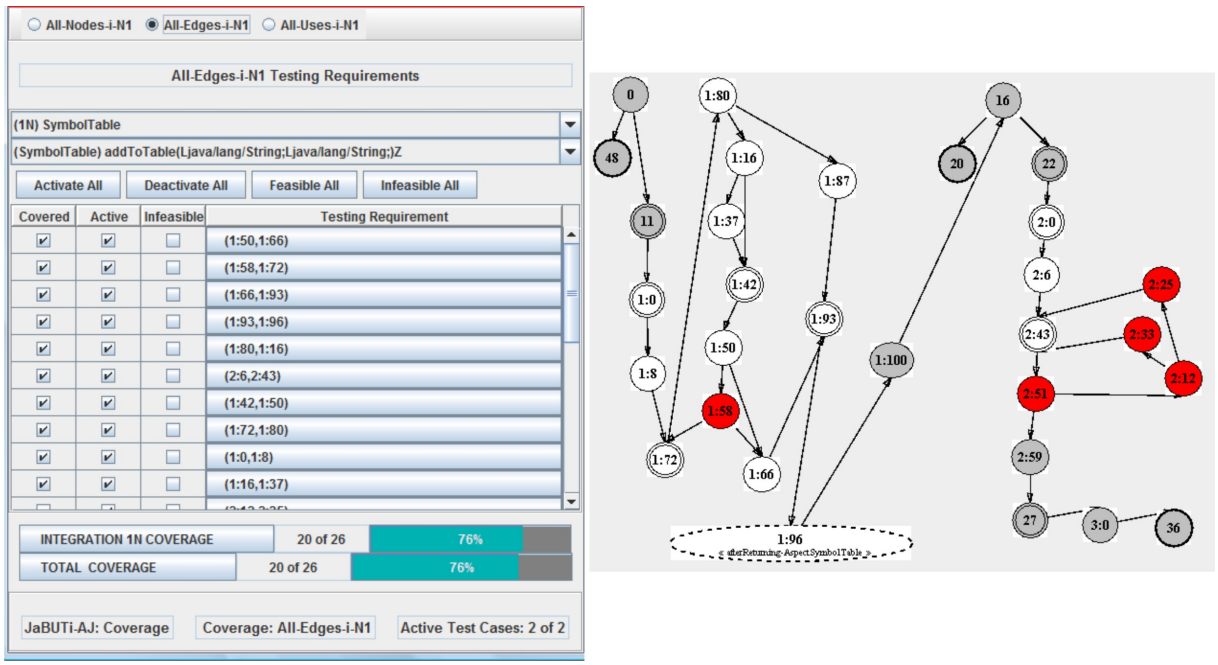

Figura 6.12: Cobertura para o critério todas-arestas-integradas-N1 tendo como base o método `addToTable`

6.5 Estratégia de Uso da Ferramenta JaBUTi/AJ

Conforme discutido na Seção 2.7, a atividade de teste pode ser realizada em três fases: (1) Teste de Unidade, (2) Teste de Integração e (3) Teste de Sistema. Seguindo essa ordem, os

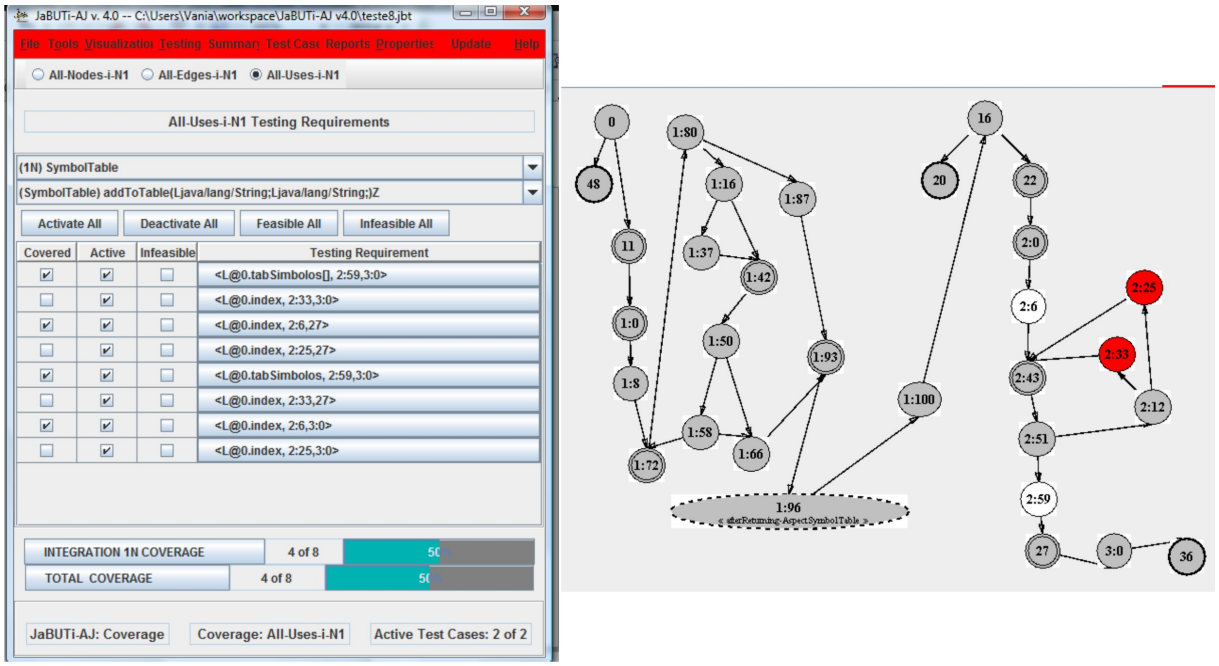

Figura 6.13: Cobertura para o critério todos-usos-integrados-N1 tendo como base o método `addToTable`

critérios desta abordagem devem ser mais efetivos se aplicados depois do teste de unidade no programa. Assim, a estratégia mais lógica a ser seguida nesse contexto seria: (1) enfatizar cada unidade testando cada método e adendo isoladamente (utilizando, por exemplo, os critérios propostos anteriormente por Lemos (2005) e por Vincenzi (2004); (2) enfatizar a interação entre essas unidades utilizando os critérios propostos nesta dissertação; (3) enfatizar os interesses transversais testando cada adendo em cada um dos pontos de junção afetados, utilizando os critérios propostos em Lemos (2009). Além do mais, o conjunto de casos de teste utilizado nos níveis precedentes pode servir de ponto de partida para os próximos níveis, sendo melhorado conforme necessário.

Rapps e Weyuker (1982) mostraram que seus critérios de fluxo de dados formam uma hierarquia de acordo com a relação de inclusão, sob certas restrições nos programas e desconsiderando caminhos não executáveis. Assim, o critério todos-usos inclui o critério todas-arestas que, por sua vez, inclui o critério todos-nós. Esses mesmos resultados podem ser aplicados aos critérios de teste de unidade e também aos critérios de teste estrutural de integração nível um, ou seja, sob as mesmas condições da hierarquia de Rapps e Weyuker (1982), o critério todos-usos-integrados-N1 inclui o critério todas-arestas-integradas-N1 que, por sua vez, inclui o critério todos-nós-integrados-N1. Levando isso em consideração, na fase (2) da estratégia apresentada deve-se seguir uma ordem de aplicação dos critérios, começando-se pelo mais fraco e incrementando-se os casos de teste em direção à adequação aos critérios mais fortes. Assim,

a ordem mais lógica seria aplicar, primeiramente, o critério todos-nós-integrados-N1, depois o critério todas-arestas-integradas-N1, e depois o critério todos-usos-integrados-N1, conforme a necessidade de se construirem mais casos de teste para adequação dos conjuntos aos critérios mais fortes.

Outro ponto a ser considerado é em relação à ordem das unidades de teste. É recomendável que seja seguida uma estratégia para determinar essa ordem de forma a diminuir o esforço durante a atividade de teste, diminuindo também o número de stubs. Para gerar a ordem das unidades de teste, esta dissertação utiliza a estratégia conjunta, proposta por Ré (2009), que determina a ordem de teste e integração de classes e aspectos, considerando que eles sejam testados e integrados conjuntamente (ver Seção 4.4). Embora a estratégia conjunta tenha sido proposta para ordenar classes e aspectos, ela pode ser facilmente adaptada para ordenar métodos e adendos.

Para melhor explicar a estratégia considere-se a Figura 6.14, que representa a hierarquia de chamadas do programa apresentado na Figura 6.15. Nessa figura, cada circunferência representa uma unidade. As unidades com nomes iniciados pela letra *m* representam métodos e os iniciados pela a letra *a* representam adendos. As linhas contínuas representam chamadas a um método e as linhas pontilhadas significam que a unidade destino está sendo afetada pelo adendo na unidade de origem. Dessa forma, m_1 chama os métodos m_2, m_3 e m_4 e é afetado pelo adendo a_1. O método m_2 chama os métodos m_5 e m_6. O método m_3 chama os métodos m_4, m_7, m_8 e m_9. O método m_4 chama o método m_1. O método m_7 chama o método m_3. O método m_8 chama o método m_4. O método m_9 é afetado pelo adendo a_1.

O adendo a_1 faz uma chamada ao método m_{10} e é afetado pelo adendo a_2. A Tabela 6.2 mostra a ordem de implementação e testes utilizando a estratégia conjunta. Dessa forma, seguindo a estratégia de teste, deve-se realizar os seguintes passos:

1. realizar o teste de unidade para os métodos m_5, m_6 e m_2. Ao completar 100% de cobertura para essas unidades, prossegue-se com o teste de integração nível um tendo como unidade base o método m_2.

2. realizar o teste de unidade para os métodos m_4, m_7, m_8, m_{10} e para o adendo a_2. Como m_4 chama m_1, que ainda não foi testado, para testá-lo é necessário implementar um *stub* de m_1. Da mesma forma, para testar m_7 é necessário implementar um *stub* de m_3. Após atingir 100% de cobertura nos critérios de unidade, prossegue-se com o teste de integração nível um, tendo o adendo a_1 como unidade base.

3. realizar o teste de unidade para o método m_9 e após atingir 100% de cobertura nesse critério, o teste de integração nível um deve ser realizado tendo como base o método m_9.

4. realizar o teste unitário para as unidades m_3 e m_1 e, em seguida, o teste de integração nível um, tendo m_3 como unidade base.

5. realizer o teste de integração nível um tendo m_1 como unidade base. Após ter alcançado 100% de cobertura para o teste de integração nível um (salvo, quando há caminhos não executáveis), prossegue-se com o teste baseado em conjuntos de junção.

6. realizar o teste baseado em conjuntos de junção considerando-se o adendo a_2 como unidade base, que afeta o adendo a_1.

7. realizar o teste baseado em conjuntos de junção considerando-se o adendo a_1 como unidade base, que afeta os métodos m_1 e m_9.

Notar que essa ordem não é única: m_6 pode ser testado antes de m_5 ou m_8 antes de m_9, por exemplo.

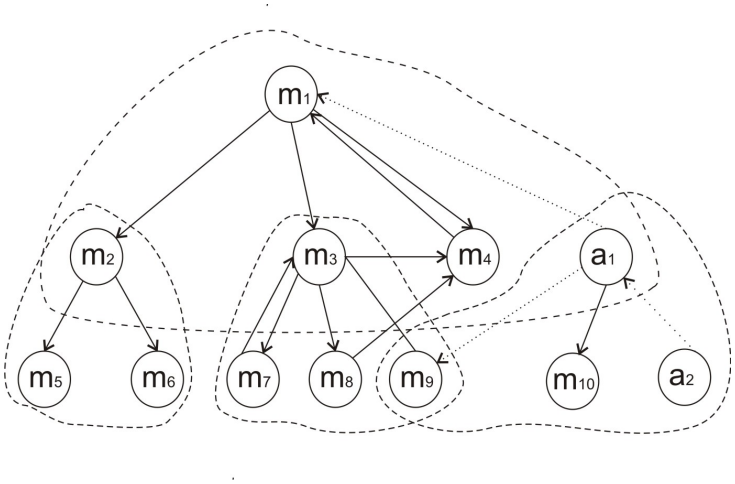

Figura 6.14: Exemplo de hierarquia de chamadas

6.5.1 Segundo Exemplo

Para exemplificar o uso da estratégia de teste, considere-se o código Java para o algoritmo heapsort apresentado na Figura 6.16. O código é composto por uma classe (`HeapSort`) e um aspecto (`Change`). A classe, por sua vez, é composta de três métodos (`heapsort`, `buildMaxHeap` e `swap`) e o aspecto é composto por um adendo do tipo `before` (`pcSwap`). O método `heapsort` tem como entrada um vetor v qualquer e a saída esperada é esse vetor ordenado. Para isso, ele chama os métodos `buildMaxHeap`, `swap` e `maxHeapify`. O método `buildMaxHeap` recebe como parâmetro esse mesmo vetor e o retorna rearranjado

```
public class Classe1 {
  public void m1(){
    System.out.println("Método 1");
    System.out.println("Chamei: ");
    m2();
    m3();
    m4();
  }

  public void m2(){
    System.out.println("Método 2");
    System.out.println("Chamei: ");
    m5();
    m6();
  }

  public void m3(){
    System.out.println("Método 3");
    System.out.println("Chamei: ");
    m4();
    m7();
    m8();
    m9();
  }

  public void m4(){
    int i = 5;
    System.out.println("Método 4");
    if (i>5) m1();
  }

  public void m5(){
    System.out.println("Método 5");
  }

  public void m6(){
    System.out.println("Método 6");
  }

  public void m7(){
    System.out.println("Método 7");
    m3();
  }

  public void m8(){
    System.out.println("Método 8");
    m4();
  }

  public void m9(){
    System.out.println("Método 9");
  }
}

public aspect Aspecto1 {
  pointcut pcm1_9(): execution(* *.m1()) ||
      execution(* *.m9());
  pointcut pcm10(): execution(* *.m10());

  before(): pcm1_9(){
    System.out.println("Adendo a1");
    System.out.println("entrecortado por a1");
    System.out.println("Chamei: ");
    m10();
  }

  public void m10(){
    System.out.println("Método 10");
  }

  after(): pcm10(){
    System.out.println("Adendo a2");
    System.out.println("entrecortado por a2");
  }
}
```

Figura 6.15: Código fonte exemplo.

Tabela 6.2: Ordem de implementação e teste do exemplo

Ordem	Módulo	Stub
1	m_5	
2	m_6	
3	m_2	
4	m_4	m_1
5	m_7	m_3
6	m_8	
7	m_{10}	
8	a_2	
9	a_1	
10	m_9	
11	m_3	
12	m_1	

com as propriedades de heap máximo. O método `swap` recebe um vetor v como parâmetro e dois índices desse vetor. A saída esperada é o vetor com os elementos dos índices dos parâmetros trocados de posição. Com os elementos trocados, o vetor não atende às propriedades de heap máximo e, por isso, é chamado o método `maxHeapify` para rearranjá-lo. Ele recebe como entrada o vetor v, o índice da raiz e o tamanho do vetor. A saída esperada é o subvetor com as propriedades de heap máximo.

```
public class Heapsort {
  public static Integer[] heapSort(Integer v[]){
    buildMaxHeap(v);
    int n = v.length-1;
    for (int i = n; i > 0; i--){
      swap(v, i , 0);
      maxHeapify(v, 0, i);
    }
    return v;
  }

  private static void buildMaxHeap(Integer v[]){
    for (int i = v.length/2 - 1; i >= 0; i--)
      maxHeapify(v, i , v.length);
  }

  private static void maxHeapify(Integer v[],
                      int pos, int n){
  int max = 2 * pos + 1, right = max + 1;
  if (max < n){
    if (right<n && v[max]<v[right])
      max = right;
      if (v[max] > v[pos]){
        swap(v, max, pos);
        maxHeapify(v, max, n);
      }
    }
  }
```

```
  public static void swap(Integer[] v, int j,
      int aposJ){
    int aux = 0;
    if (v[j] != v[aposJ]){
      aux = v[j];
      v [j] = v [ aposJ ];
      v [ aposJ ] = aux;
    }
    else{
      aux = v[v.length-1];
      v[v.length-1] = v[0];
      v[0] = aux;
    }
  }
}

import java.util.Arrays;

public aspect Change {
  pointcut pcSwap(Integer[] v, int j, int aposJ):
      call(* *.swap(..)) && args(v, j, aposJ);

  before (Integer[] v, int j, int aposJ)
      : pcSwap(v, j, aposJ) {
    System.out.println("v: "+Arrays.toString(v));
  }
}
```

Figura 6.16: Código fonte do algoritmo heapsort.

Para exemplificar o uso de aspectos, a chamada ao método `swap` foi entrecortada pelo adendo `before`. A finalidade desse adendo é apenas imprimir o novo vetor cada vez que um elemento mudar de posição. Um defeito foi inserido no método `swap`: quando o elemento de índice j for igual ao de índice $aposJ$, o último elemento do vetor troca de posição com o primeiro.

Considerando-se a Figura 6.17, para realizar o teste seguindo a estratégia de teste da seção anterior, deve-se realizar os seguintes passos:

1. realizar o teste de unidade para o método `swap`, depois para o método `maxHeapify` e, em seguida, para o método `buildMaxHeap`. Com o conjunto de casos de teste inicial,

apresentado da Tabela 6.3, é possível obter 100% de cobertura para todos os critérios de unidades.

2. realizar o teste de integração nível um tendo o método `buildMaxHeap` como unidade sob teste. Utilizando-se o mesmo conjunto é possível obter 100% de cobertura nos critérios todos-nós-integrados-N1 e todos-usos-integrados-N1. O critério todas-arestas-integradas-N1 alcança 93% de cobertura, cobrindo 14 de 15 requisitos. A aresta não coberta foi a (1:0, 1:122), que só será executada se a condição `if max<n`, da linha 20, não for verdadeira. Porém, a análise do código mostra que, quando o método `maxHeapify` for chamado por `buildMaxHeap`, essa condição sempre será verdadeira e, assim, essa aresta nunca será executada nessas condições. Dessa forma, a aresta (1:0, 1:122) é assinalada como *infeasible* na ferramenta e, dessa forma, obtém cobertura total em todos os critérios de teste de integração nível um.

3. realizar o teste de integração nível um tendo-se o método `maxHeapify` como unidade sob teste. Com o conjunto utilizado anteriormente, obteve-se 86% de cobertura para o critério todos-nós-integrados-N1, cobrindo 13 de 15 requisitos; 64% de cobertura para o critério todas-arestas-integradas-N1, cobrindo 11 de 17 requisitos e 44% de cobertura para o critério todos-usos-integrados-N1, cobrindo 20 de 45 requisitos. Obedecendo a ordem de aplicação dos critérios, o teste prossegue tentando gerar mais casos de teste para aumentar a cobertura do critério todos-nós-integrados-N1. Dessa forma, foi incluído o caso de teste que tem o vetor <7, 6, 5, 4, 3, 2, 1> como entrada. Com a adição desse caso de teste foi possível cobrir mais um nó. Por meio da análise do código-fonte é possível observar que o nó restante, o 2:34, só será executado se a condição `v[j]!=v[aposJ]` for falsa. No entanto, o método `swap` só é chamado por `maxHeapify` se a condição `v[max]>v[pos]` for verdadeira, onde `max` corresponde a `j` e `pos` corresponde a `aposJ`, ocasionando então que o nó 2:34 nunca será executado nessas condições, devendo ser assinalado como *infesible* e, com isso, atingindo cobertura total nesse critério. Com mais esse caso de teste, a cobertura do critério todas-arestas-integradas-N1 aumentou para 88%, cobrindo 15 de 17 requisitos. As arestas não cobertas foram (2:0, 2:34) e (2:34, 2:68) que, como foi explicado anteriormente, nunca serão executadas. Assinalando-as como *infeasible* também obteve-se cobertura total nesse critério. A cobertura do critério todos-usos-integrados-N1 aumentou para 54%, cobrindo 24 de 45 requisitos. No entanto, pode-se perceber que os requisitos não cobertos do critério todos-usos-integrados-N1 têm ou uma definição ou um uso no nó 2:34, podendo, dessa forma, ser assinalados como *infeasible*, e atingindo, então, cobertura total também para esse critério.

4. realizar o teste de unidade para o adendo `before`. Utilizando-se o mesmo conjunto foi possível obter 100% de cobertura para todos os critérios de unidade.

5. realizar o teste de integração nível um tendo como base o método `heapsort`. Utilizando-se o mesmo conjunto foi possível obter cobertura total para os critérios todos-nós-integrados-N1 e todas-arestas-integradas-N1, como pode ser visto nas Figuras 6.18 e 6.19, respectivamente. O critério todos-usos-integrados-N1 obteve 79% de cobertura, cobrindo 39 de 49 requisitos, como pode ser visto na Figura 6.20. O caso de teste que tem como entrada o vetor <2, 2, 3, 2, 3, 3> foi gerado para tentar cobrir o requisito <l@5, 3:34, (4:22, 4:38)>, mas ao executar esse caso de teste um erro de integração aconteceu, revelando então o defeito. Ao corrigir o programa, é necessário refazer os testes para garantir que nenhum novo defeito foi inserindo. Ao refazer os testes, verifica-se o resultado esperado, cobrindo todos os requisitos, inclusive para o critério todos-usos-integerados-N1, tendo o método `heapsort` como unidade base.

6. realizar o teste com os critérios baseados em conjunto de junção. Dessa forma, a unidade sob teste seria o adendo `before`. Com o mesmo conjunto foi possível obter 100% de cobertura em todos os critérios dessa abordagem.

Notar que nesse exemplo não há a necessidade de criar *stubs* para fazer o teste de unidades de todos os métodos e adendos. A Tabela 6.4 apresenta o conjunto completo de casos de teste em que se obtém 100% de cobertura em todos os critérios. A Tabela 6.5 apresenta a ordem de implementação e testes de classes e aspectos para esse exemplo.

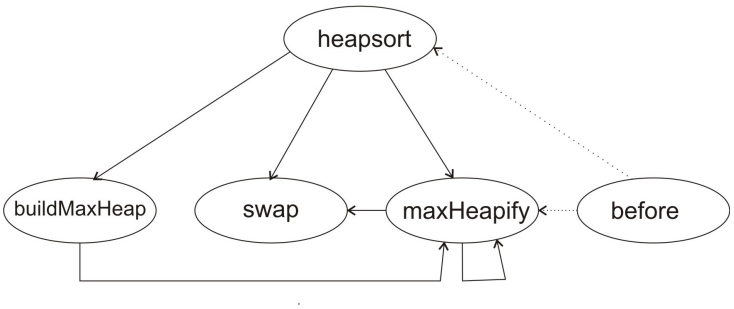

Figura 6.17: Exemplo de hierarquia de chamadas

6.6 Considerações Finais

Neste capítulo foi descrita a implementação da abordagem de teste estrutural de integração nível um proposta neste trabalho para teste de programas OO e OA. Esta abordagem foi implementada

Tabela 6.3: Exemplo de conjunto de casos de teste para o teste de unidade

CT	Vetor de Entrada	Saída Esperada
1	$< [2, 1, 4, 5, 6, 7, 3] >$	$< [1, 2, 3, 4, 5, 6, 7] >$
2	$< [2, 1, 4, 5, 6, 7, 3] >$	$< [7, 6, 4, 5, 1, 2, 3] >$
3	$< [2, 1, 4, 5, 6, 7, 3] >$	$< [2, 1, 7, 5, 6, 4, 3] >$

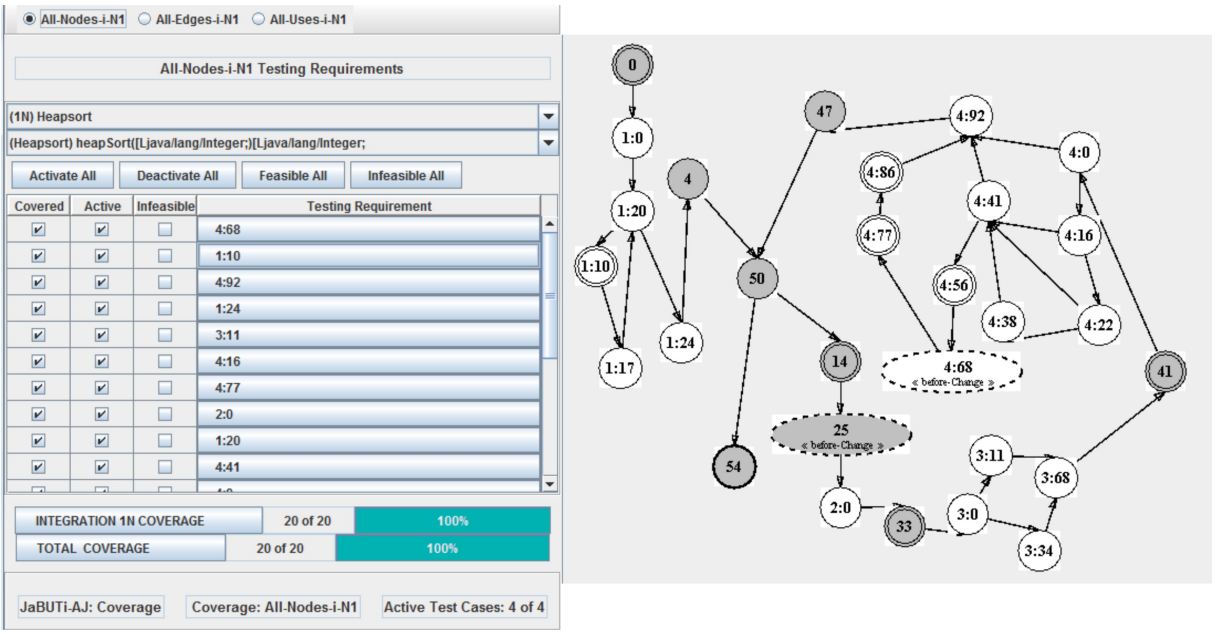

Figura 6.18: Grafo $\mathcal{INIP}$ e requisitos de teste para o critério todos-nós-integrados-N1 do método `heapsort`

Tabela 6.4: Conjunto completo de casos de teste para o algoritmo heapsort

CT	Vetor de Entrada	Saída Esperada
1	$< [2, 1, 4, 5, 6, 7, 3] >$	$< [1, 2, 3, 4, 5, 6, 7] >$
2	$< [2, 2, 2, 2, 2, 2, 2] >$	$< [2, 2, 2, 2, 2, 2, 2] >$
3	$< [2, 1, 4, 5] >$	$< [1, 2, 4, 5] >$
4	$< [7, 6, 5, 4, 3, 2, 1, 0] >$	$< [0, 1, 2, 3, 4, 5, 6, 7] >$
5	$< [2, 2, 3, 2, 3, 3] >$	$< [2, 2, 2, 3, 3, 3] >$

Tabela 6.5: Ordem de implementação e teste para o exemplo do heapsort

Ordem	Módulo	Stub
1	`swap`	
2	`maxHeapify`	
3	`buildMaxHeap`	
4	`before`	
5	`heapsort`	

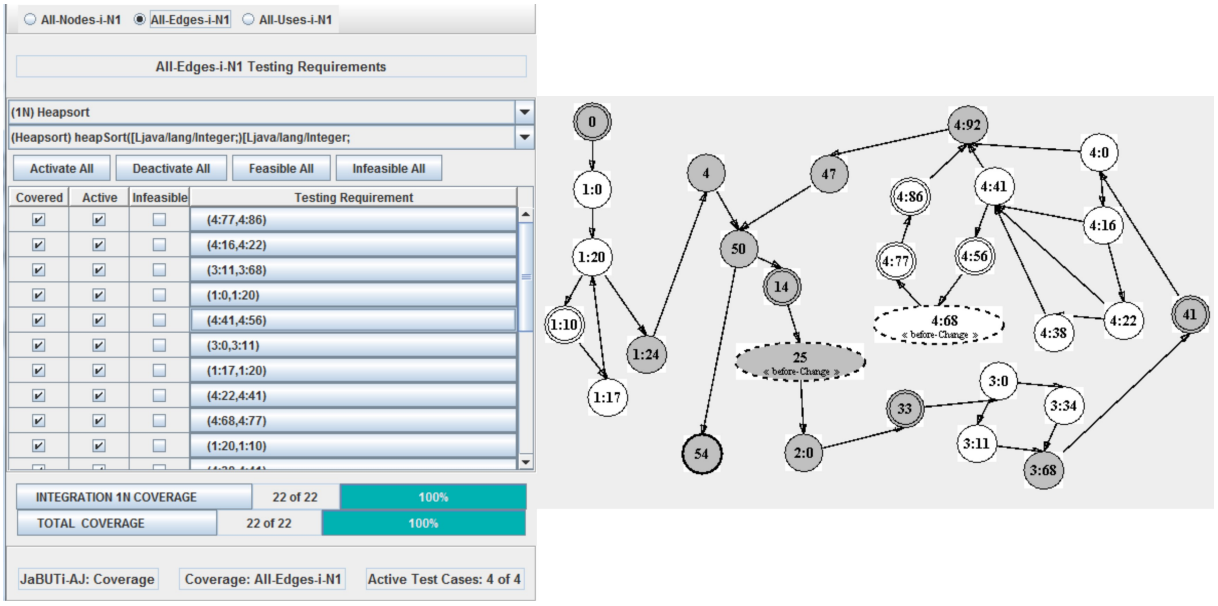

Figura 6.19: Grafo $\mathcal{INIP}$ e requisitos de teste para o critério todas-arestas-integrados-N1 do método `heapsort`

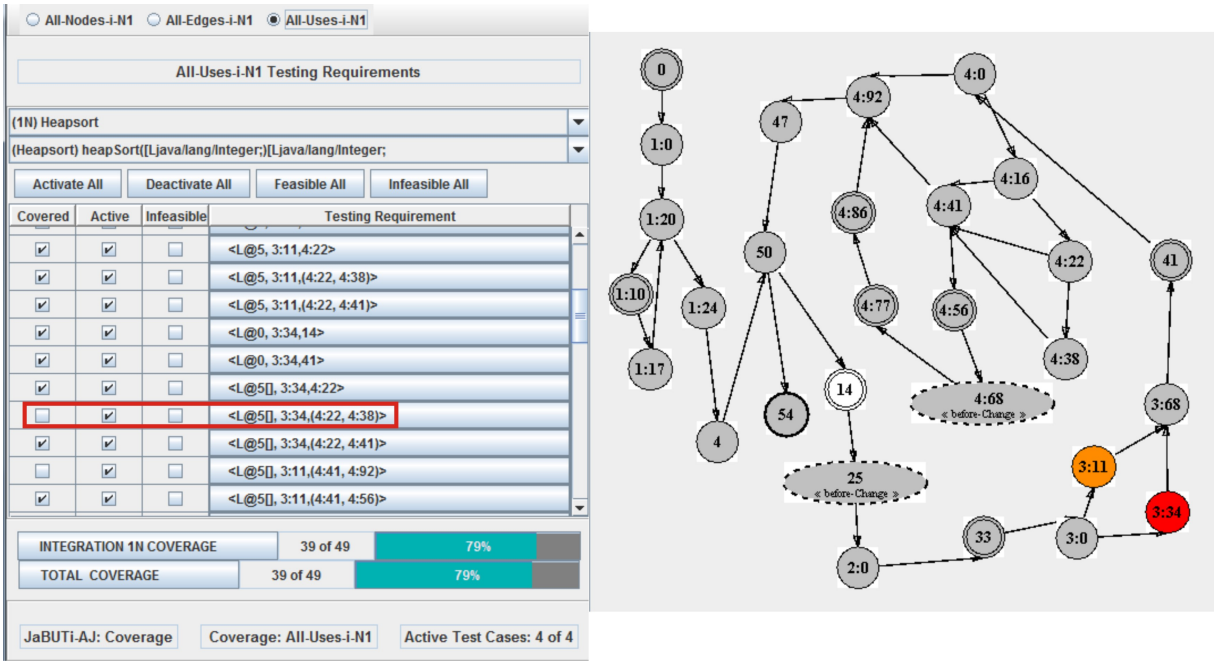

Figura 6.20: Grafo $\mathcal{INIP}$ e requisitos de teste para o critério todos-usos-integrados-N1 do método `heapsort`

na ferramenta JaBUTi/AJ. Assim, com essa nova versão da ferramenta JaBUTi/AJ, além dos testes de unidade, par-a-par e baseados em conjuntos de junção, é possível efetuar também o teste de integração nível um para programas OO e OA escritos em Java e AspectJ.

Neste capítulo também foram apresentados exemplos de uso da ferramenta e uma estratégia de teste, com o objetivo de tornar a atividade de teste mais eficiente.

Um ponto a ser considerado é no caso em que a unidade base é um adendo do tipo `around` e que faz chamada ao método `proceed`. Nesse caso, a chamada ao método `proceed` não será integrada, uma vez que, no contexto do adendo, não faz sentido testar a integração com o método original entrecortado que, inclusive, pode ser mais que um. No entanto, como já explicado na Seção 5.6.2, quando um adendo desse tipo afetar a unidade base, o método original será integrado.

Com essa implementação também é possível realizar o teste de sequências de chamadas limitando-se ao primeiro nível. Por exemplo, considerando as operações de um diagrama de sequências (ou de comunicação) poder-se-ia criar um método do tipo pseudo-controlador (*driver*) que contém as chamadas para cada operação e a ferramenta criaria o grafo integrado e apoiaria a aplicação dos critérios definidos neste trabalho. Para exemplificar, considere-se o exemplo da tabela de símbolos. Harrold e Rothermel (1994), em seu trabalho, insere um defeito e afirma que para encontrá-lo é necessário testar a sequência <`addToTable`, `addToTable`>. Desse modo, basta criar um método (por exemplo, um método "`main`") que faz duas chamadas ao método `addToTable` e, em seguida, integrá-lo utilizando a ferramenta JaBUTi/AJ. Assim, o critério todos-usos-integrados-N1 revelaria o defeito inserido.

No próximo capítulo são apresentadas as conclusões finais do trabalho, as contribuições e os trabalhos futuros a serem realizados.

CAPÍTULO

7

Conclusão

7.1 Considerações Finais

Neste trabalho foi proposta uma abordagem de teste estrutural de integração nível um de programas orientados a objetos e a aspectos, adaptando critérios de fluxo de controle e fluxo de dados anteriormente propostos para esta nova abordagem. Nesta abordagem de teste considera-se todo o fluxo de execução (fluxo de controle e de dados) que ocorre entre a unidade chamadora e todas as unidades que interagem diretamente com ela, seja a chamada de um método ou o entrecorte de um adendo. Isto é, uma unidade pode chamar ou ser afetada por outras unidades em seu escopo. Cada vez que isso acontece o fluxo de execução é passado para a unidade chamada (ou a que a entrecorta). Esta executa e, após a sua execução, o fluxo retorna à unidade que chamou (ou que foi afetada), que continua sua execução. Para representar esse fluxo de execução entre as unidades foi definido o grafo $\mathcal{INIP}$, que é uma abstração formada pela integração dos grafos $\mathcal{AODU}$ da unidade chamadora com as unidades que ela chama ou que a afetam diretamente.

Foram também propostos três critérios específicos para derivar requisitos de teste no nível um de profundidade. Entre eles, dois critérios são baseados em fluxo de controle (todos-nós-integrados-N1 e todas-arestas-integradas-N1) e um critério é baseado em fluxo de dados (todos-usos-integrados-N1). A ferramenta JaBUTi/AJ foi estendida para dar apoio à abordagem de teste estrutural de integração nível um proposta. Com essa extensão é possível efetuar tanto testes de unidade (Lemos, 2005), teste de integração par-a-par (Franchin, 2007)

e teste de integração baseado em descritores de conjuntos de junção (Lemos, 2009), quanto teste de integração nível um de programas OO e OA escritos em Java e AspectJ com apoio da ferramenta.

7.2 Contribuições

As contribuições deste trabalho foram as seguintes:

1. Desenvolvimento de uma abordagem de teste estrutural de integração nível um para programas OO e OA escritos em Java e AspectJ, que inclui: a identificação de todas as unidades que se relacionam diretamente, a definição do grafo $\mathcal{IN}1\mathcal{P}$ utilizado para representar o fluxo de execução entre as unidades e a definição de três critérios de teste estrutural de integração nível um: todos-nós-integrados-N1, todas-arestas-integradas-N1 e todos-usos-integrados-N1.

2. Tratamento especial para o polimorfismo — cuja construção é feita como se fosse um frame composto pelos grafos $\mathcal{AODU}$ dos métodos possíveis e selecionados pelo usuário — e para o caso de uso do adendo `around` com chamada ao método `proceed` — em que o grafo do método entrecortado é entrecortado após a chamada ao método `proceed`. Para os casos em que o grafo $\mathcal{AODU}$ da unidade integrada for linear, ela será representado com um único nó. Esse nó traz consigo todas as informações def-uso dos outros nós que foram removidos.

3. Extensão da ferramenta JaBUTi/AJ para apoiar a abordagem proposta. Essa extensão envolveu: a implementação do grafo $\mathcal{IN}1\mathcal{P}$, construído a partir dos grafos $\mathcal{AODU}$ da unidade chamadora com as unidades que interagem diretamente com ela, a implementação dos critérios de teste estrutural de integração nível um, a implementação dos casos de polimorfismo, do adendo `around` com chamada ao método `proceed`, implementação da otimização do grafo $\mathcal{IN}1\mathcal{P}$ para os casos em que a unidade integrada for linear e desenvolvimento do ambiente de teste de integração nível um.

7.3 Trabalhos Futuros

Como trabalhos futuros podem ser realizados experimentos utilizando diversos programas reais escritos em Java e AspectJ com a finalidade de avaliar a eficácia da abordagem proposta em revelar defeitos existentes tanto na interface entre essas unidades quanto na execução de sequência de chamadas. Mais especificamente, experimentos também poderiam ser feitos com

o apoio da ferramenta JaBUTi/AJ estendida para o teste de integração nível um para avaliar a efetividade desta abordagem.

A realização dos experimentos também auxiliaria no teste da ferramenta em que poder-se-ia revelar eventuais defeitos remanescentes na ferramenta. Outro trabalho futuro importante seria investigar os critérios aqui propostos no contexto de uma estratégia de teste. Ou seja, com os experimentos seria possível avaliar a eficiência/utilidade dos critérios em encontrar defeitos após ter sido realizado o teste de unidades.

Outra linha de pesquisa relevante seria investigar refinamentos dos critérios propostos para diminuir a quantidade de pares def-uso gerado pelo critério de fluxo de dados proposto.

Um trabalho futuro que já está em andamento é a extensão deste trabalho que aumenta o nível de profundidade da integração (ou linearização) para um k parametrizado, com k>1, generalizando a proposta apresentada nesta dissertação.

Referências Bibliográficas

ALEXANDER, R. T.; BIEMAN, J. M.; ANDREWS, A. A. *Towards the systematic testing of aspect-oriented programs*. Technical report, Department of Computer Science, Colorado State University, 2004.

ASPECTJ TEAM The AspectJ development environment guide. Online, disponível em `http://www.eclipse.org/aspectj/doc/released/devguide/index.html` - Último acesso em 10/11/09, 2005.

BECK, K.; GAMMA, E. JUnit Cookbook. Online, disponível em `http://junit.sourceforge.net/doc/cookbook/cookbook.htm` - Último acesso em 12/08/08, 2008.

BINDER, R. V. *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

BOEHM, B. A view of 20th and 21st century software engineering. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*, New York, NY, USA: ACM, 2006, p. 12–29.

BONÉR, J.; VASSEUR, A. AspectWerkz - Plain Java AOP - Overview. Online, disponível em `http://aspectwerkz.codehaus.org/` - Último acesso em 16/11/09, 2008.

BRIAND, L. C.; LABICHE, Y.; WANG, Y. An investigation of graph-based class integration test order strategies. *IEEE Transactions of Software Engeneering*, v. 29, n. 7, p. 594–607, 2003.

CAPRETZ, L. F. A brief history of the object-oriented approach. *SIGSOFT Softw. Eng. Notes*, v. 28, n. 2, p. 6, 2003.

COELHO, R. S.; STAA, A. V.; KULESZA, U.; RASHID, A.; LUCENA, C. J. P. Unveiling and taming liabilities of aspects in the presence of exceptions: A static analysis based approach. *Information Sciences*, accepted for publication, 2009.

COUTO, C. F. M. *Um arcabouço orientado por aspectos para implementação automatizada de persistência*. Dissertação de Mestrado, Universidade Federal de Minas Gerais, Belo Horizonte/MG - Brasil, 2006.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao teste de software*, v. 394 de *Campus*. Elsevier, 2007.

DOMINGUES, A. L. S. *Avaliação de critérios e ferramentas de teste para programas OO*. Dissertação de Mestrado, ICMC/USP, São Carlos/SP - Brasil, 2002.

ELRAD, T.; AKSIT, M.; KICZALES, G.; LIEBERHERR, K.; OSSHER, H. Discussing aspects of aop. *Communications of the ACM*, v. 44, n. 10, p. 33–38, 2001a.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Commun. ACM*, v. 44, n. 10, p. 29–32, 2001b.

FERRARI, F. C.; MALDONADO, J. C.; RASHID, A. Mutation testing for aspect-oriented programs. In: *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, Washington, DC, USA: IEEE Computer Society, 2008, p. 52–61.

FRANCHIN, I. *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 - Brasil, 2007.

GOSLING, J.; MCGILTON, H. The Java Language Environment. online, disponível em Sun Developer Network: `http://java.sun.com/docs/white/langenv/index.html` - Último acesso em 21/05/08, 1996.

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.

HILSDALE, E.; HUGUNIN, J. Advice weaving in aspectj. In: *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA: ACM, 2004, p. 26–35.

IEEE *Ieee standard glossary of software engineering terminology*. Standard 610.12, Institute of Electric and Electronic Engineers, 1990.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. Getting started with aspectj. *Commun. ACM*, v. 44, n. 10, p. 59–65, 2001.

KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J. M.; IRWIN, J. Aspect-oriented programming. In: AKŞIT, M.; MATSUOKA, S., eds.

Proceedings European Conference on Object-Oriented Programming, v. 1241, Berlin, Heidelberg, and New York: Springer-Verlag, p. 220–242, 1997.
Disponível em `citeseer.ist.psu.edu/kiczales97aspectoriented.html`

KUNG, D. C.; GAO, J.; HSIA, P.; LIN, J.; TOYOSHIMA, Y. Class firewall, test order, and regression testing of object-oriented programs. 1993.

LABICHE, Y.; THÉVENOD-FOSSE, P.; WAESELYNCK, H.; DURAND, M.-H. Testing levels for object-oriented software. In: *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, New York, NY, USA: ACM, 2000, p. 136–145.

LEMOS, O.; RÉ, R.; MASIERO, P. *Introdução ao teste de software*, v. 394 de *Campus*, cáp. 7 Elsevier, p. 175–207, 2007.

LEMOS, O. A. *Teste estrutural de integração de programas orientados a aspectos: Uma abordagem baseada em conjuntos de junção para aspectj*. Tese de Doutoramento, ICMC/USP, São Carlos, SP - Brasil, 2009.

LEMOS, O. A. L. *Teste de programas orientados a objetos e a aspectos: Uma abordagem estrutural para aspectj*. Dissertação de Mestrado, ICMC/USP, São Carlos/SP - Brasil, 2005.

LINDHOLM, T.; YELLIN, F. The Java Virtual Machine Specification. Online, disponível em `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html` - Último acesso em 21/05/08, 2008.

LINNENKUGEL, U.; MÜLLERBURG, M. Test data selection criteria for (software) integration testing. In: *ISCI '90: Proceedings of the first international conference on systems integration on Systems integration '90*, Morristown, New Jersey, United States: IEEE Press, 1990, p. 709–717.

LOPES, C. V.; NGO, T. Unit-testing aspectual behavior. In: *Workshop on Testing Aspect-Oriented Programs*, 2005.

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

MASSICOTTE, P.; BADRI, L.; BADRI, M. Towards a tool supporting integration testing of aspect-oriented programs. *Journal of Object Technology*, v. 6, n. 1, p. 67–89, 2007.

MCDIRMID, S.; HSIEH, W. C. Aspect-oriented programming with jiazzi. In: *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, New York, NY, USA: ACM Press, 2003, p. 70–79.

MEZINI, M.; OSTERMANN, K. CaesarJ Project. Online, disponível em `http://caesarj.org/` - Último acesso em 16/11/09, 2009.

MYERS, G. *The art of software testing*. John Wiley and Sons, 2004.

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.

PARASOFT CORPARATION Jtest: Java Unit Testing Code Compliance - Parasoft. Online, disponível em `http://www.parasoft.com/` - Último acesso em 12/08/08, 2008.

PRESSMAN, R. S. *Software engineering - a practitioner's approach*. 5th. ed. McGraw-Hill, 2000.

QUEST SOFTWARE Java Profiler for J2EE and Java Performance Monitoring with JProbe by Quest Software. Online, disponível em `http://www.quest.com/jprobe/` - Último acesso em 12/08/08, 2008.

RÉ, R. *Teste de integração de programas baseados em interesses*. Tese de Doutoramento, ICMC/USP, São Carlos, SP - Brasil, 2009.

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.

RAPPS, S.; WEYUKER, E. J. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, v. SE-11, n. 4, p. 367–375, 1985.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de software: Teoria e prática*. São Paulo, SP: Prentice Hall, 2001.

SAUNDERS, B.; STAMEY, J.; CAMERON, M. Aspect Oriented PHP. Online, disponível em `http://www.aophp.net/` - Último acesso em 28/08/08, 2008.

SPINCZYK, O.; LOHMANN, D.; URBAN, M. AspectC++: an AOP Extension for C++. Online, disponível em `http://www.aspectc.org/` - Último acesso em 28/08/08, 2008.

SUN MICROSYSTEMS The Java Tutorial. Online, disponível em `http://java.sun.com/docs/books/tutorial/index.html` - Último acesso em 21/05/08, 2008.

TARJAN, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, v. 1, n. 2, p. 146–160, 1972.

TARR, P.; OSSHER, H.; SUTTON, JR., S. M. Hyper/j™: multi-dimensional separation of concerns for java™. In: *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA: ACM, 2002, p. 689–690.

VINCENZI, A. M. R. *Orientação a objeto: Definição, implementação e análise de recursos de teste e validação*. Tese de Doutoramento, ICMC/USP, São Carlos, SP - Brasil, 2004.

XIE, T.; ZHAO, J.; MARINOV, D.; NOTKIN, D. Automated test generation for aspectj programs. In: *AOSD 2005: Workshop on Testing Aspect-Oriented Programs*, 2005.

XU, D.; XU, W.; NYGARD A state-based approach to testing aspect-oriented programs. In: *SEKE'05: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, Taiwan, 2005, p. 14–16.

ZHAO, J. Tool support for unit testing of aspect-oriented software. In: *OOPSLAŠ2002 Workshop on Tools for Aspect-Oriented Software Development*, Seatle/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*, Washington, DC, USA: IEEE Computer Society, 2003, p. 188–197.

ZHU, H.; HALL, P.; MAY, J. Software unit test coverage and adequacy. *ACM Computing Surveys*, v. 29, n. 4, p. 366–427, 1997.