Uma contribuição para a minimização do número de stubs no teste de integração de programas orientados a aspectos

Reginaldo Ré

Uma contribuição para a minimização do número de stubs no teste de integração de programas orientados a aspectos

Reginaldo Ré

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

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

USP - São Carlos
Fevereiro/2009

Dedico este trabalho à Tatiane,
à Páscoa, à Regiani e ao Aldecir

Agradecimentos

Agradeço a Deus pelos momentos felizes e pelos momentos tristes, que nos lembram o quanto é necessário ter fé.

À Tatiane, minha esposa, companheira e amiga, pelo carinho, amor, compreenção e incentivo.

À minha família, e à minha nova familia, por terem me dado o que tenho de mais precioso e o que é primordial à pessoa humana, amor incondicional.

Ao meu orientador, professor Masiero, por compartilhar comigo seu trabalho, pela tenacidade em me ensinar, pela compreenção das minhas dificuldades e, principalmente, pelo exemplo de conduta na vida profissional.

Aos muitos amigos que conviveram comigo durante os anos de estudo, devo grande parte deste trabalho a vocês. Dinho, Douglas, Julinho, Rui, Taka e Tião, obrigado pelos momentos de felicidade que compartilhamos e por nunca se cansarem de ouvir desculpas pelas minhas ausências. Rogério, Menotti e Marinho, obrigado pelo apoio, incentivo e companheirismo durante o tempo de pós-graduação. Apenas quem já foi ou é aluno da pós-graduação do ICMC/USP sabe o quanto isso é importante. Chis, Eduardo e Sérgio, obrigado por permitir a mim e a Tatiane fazermos parte de sua família. Adenilso, André, Fabiano e Otávio, obrigado pelo auxílio em questões técnicas surgidas durante a pós-graduação e pela convivência enriquecedora.

Aos amigos da COINF-CM, Frank, Igor, Ivanilton, Kawamoto, Lúcio, Marco Aurério, Radames, Rafael, Rogério, Silvano e Schwerz, por assumir minhas responsabilidades na UTFPR-CM enquanto me dediquei a este trabalho.

Aos amigos Marcos Piza, Paulo e Raquel, e aos demais amigos da UTFPR-CM, por tornar a vida em Campo Mourão mais feliz.

Ao amigo Narci, como Gerente de Ensino da UTFPR-CM, e ao amigo Celso, como Diretor da UTFPR-CM, pelo apoio profissional, mesmo frente às dificuldades encontradas.

Aos professores e funcionários do ICMC-USP, pela disposição e atenção.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Resumo

A programação orientada a aspectos é uma abordagem que utiliza conceitos da separação de interesses para modularizar o software de maneira mais adequada. Com o surgimento dessa abordagem vieram também novos desafios, dentre eles o teste de programas orientados a aspectos. Duas estratégias de ordenação de classes e aspectos para apoiar o teste de integração orientado a aspectos são propostas nesta tese. As estratégias de ordenação tem o objetivo de diminuir o custo da atividade de teste por meio da diminuição do número de *stubs* implementados durante o teste de integração. As estratégias utilizam um modelo de dependências aspectuais e um modelo que descreve dependências entre classes e aspectos denominado AORD – (*Aspect and Oriented Relation Diagram*) – também propostos neste trabalho. Tanto o modelo de dependências aspectuais como o AORD foram elaborados a partir da sintaxe e semântica da linguagem AspectJ. Para apoiar as estratégias de ordenação, idealmente aplicadas durante a fase de projeto, um processo de mapeamento de modelos de projeto que usam as notações UML e MATA para o AORD é proposto neste trabalho. O processo de mapeamento é composto de regras que mostram como mapear dependências advindas da programação orientada a objetos e também da programação orientada a aspectos. Como uma forma de validação das estratégias de ordenação, do modelo de dependências aspectuais e do AORD, um estudo exploratório de caracterização com três sistemas implementados em AspectJ foi conduzido. Durante o estudo foram coletadas amostras de casos de implementação de *stubs* e *drivers* de teste. Os casos de implementação foram analisados e classificados. A partir dessa análise e classificação, um catálogo de *stubs* e *drivers* de teste é apresentado.

Abstract

Aspect-oriented programming is an approach that uses principles of separation of concerns to improve the sofware modularization. Testing of aspect-oriented programs is a new challenge related to this approach. Two aspects and classes test order strategies to support integration testing of aspect-oriented programs are proposed in this thesis. The objective of these strategies is to reduce the cost of testing activities through the minimization of the number of implemented stubs during integration test. An aspectual dependency model and a diagram which describes dependencies among classes and aspects called AORD (Aspect and Object Relation Diagram) used by the ordering strategies are also proposed. The aspectual dependency model and the AORD were defined considering the syntax constructions and the semantics of AspectJ. As the proposed estrategies should be applied in design phase of software development, a process to map a desing model using UML and MATA notations into a AORD is proposed in order to support the ordering strategies. The mapping process is composed by rules that show how to map both aspect and object-oriented dependencies. A characterization exploratory study using three systems implemented with AspectJ was conducted to validate the ordering strategies, the aspectual dependency model and the AORD. Interesting samples of stubs implementations were collected during the study conduction. The stubs were analyzed and classified. Based on these analysis and classification a catalog of stubs and drivers is presented.

CAPÍTULO

1

Introdução

1.1 Teste de programas orientados a aspectos

A engenharia de software tem evoluído com o surgimento de diversas propostas com o objetivo de oferecer soluções para os problemas relacionados ao desenvolvimento de software. As propostas procuram, de modo geral, oferecer meios para melhorar a qualidade do software, reduzir os custos de produção e facilitar a evolução e manutenção do software (Tarr *et al.*, 1999). Para tanto, elas buscam reduzir a complexidade, promover a melhoria da compreensão e enfatizar o reúso de software. A separação de interesses é um conceito que possibilita o alcance desses objetivos por meio de um mecanismo de decomposição que permite a identificação e o encapsulamento de comportamentos transversais em entidades manipuláveis (Kienzle *et al.*, 2003). Essas entidades contêm comportamento que, em uma linguagem de programação que não considera a separação de interesses, está espalhado e embaralhado por todo o código fonte (Baniassad e Clarke, 2004). O diferencial das propostas que usam o conceito de separação de interesses está no fato de que o desenvolvedor pode manter todos os interesses em um mesmo nível de abstração, sem ser obrigado a eleger um interesse principal e decompor os outros interesses em função de um único interesse dominante.

A separação de interesses não é um conceito novo, mas recentemente tem estimulado pesquisas pelo seu potencial em modularizar o software de maneira mais adequada. Com a modularização de interesses, a estrutura do código fonte é simplificada, melhorando a legibilidade e aumentando a possibilidade de reúso. Além disso, a manutenibilidade é promovida graças a sua característica não invasiva (Tarr *et al.*, 1999). O potencial desse conceito é reconhecido pela comunidade especializada, fato corroborado pela quantidade de trabalhos que vêm surgindo recentemente sobre esse assunto. Não obstante o reconhecimento, esse conceito traz consigo

novos desafios, e pesquisas devem ser feitas para explorar e comprovar seus benefícios e, ao mesmo tempo, identificar e reconhecer seus principais problemas e propor soluções para eles.

A programação orientada a aspectos (OA) (Kiczales *et al.*, 1997) apoia o conceito de separação de interesses. Na programação OA os interesses de um sistema são decompostos em diferentes unidades modulares que são posteriormente compostas em um único sistema. Essas unidades são as classes-base e os aspectos, que representam, respectivamente, os interesses considerados básicos e transversais de um sistema. AspectJ (The AspectJ Team, 2002) é a linguagem de programação OA mais difundida atualmente.

Dentre os diferentes focos de pesquisa nessa área, o teste de programas OA é um dos pontos explorados mais recentemente. Algumas propostas direcionadas para o teste de programas OA foram publicadas nos últimos anos, como por exemplo: Alexander e Bieman (2004) identificam as possíveis origens dos defeitos em programas OA e apresentam um modelo de defeitos; Lemos (2005), Lemos *et al.* (2004) e Zhao (2003a) tratam do teste estrutural de unidades; Yamazaki *et al.* (2005) e Lopes e Ngo (2005) tratam do teste funcional de unidades; Lemos *et al.* (2007), Lemos *et al.* (2006) e Franchin *et al.* (2007) tratam do teste estrutural de integração; Xu *et al.* (2005) tratam do teste baseado em estados; e, Ferrari *et al.* (2008) e Anbalagan e Xie (2006) tratam do teste de mutação.

Uma área ainda pouco explorada é o teste de integração de programas OA[1]. Analogamente ao que ocorre com os programas orientados a objetos (OO), na programação OA é necessário o estudo de problemas ligados ao teste de integração, como por exemplo, estratégias de ordenação de classes e aspectos que diminuam o custo da atividade de teste durante a integração de unidades de software. Muitas estratégias de ordenação de classes com vistas a minimizar o número de *stubs* de teste e indicar uma ordem de implementação de classes, têm sido propostas no contexto da programação OO (Abdurazik e Offutt, 2006; Badri *et al.*, 2005; Briand *et al.*, 2002, 2003; Jaroenpiboonkit e Suwannasart, 2007; Kung *et al.*, 1995a; Labiche *et al.*, 2000; Le Hanh *et al.*, 2001; Le Traon *et al.*, 2000; Lima e Travassos, 2004; Malloy *et al.*, 2003; Mao e Lu, 2005; Tai e Daniels, 1999). O problema da ordenação de classes ocorre por causa de ciclos de dependência entre as classes. As estratégias de ordenação visam a minimizar o custo da atividade de teste pela minimização do número de *stubs* necessários para quebrar dependências cíclicas entre as classes.

Seguindo essa linha geral, o trabalho apresentado nesta tese está relacionado com as fases do teste de integração de programas OA, especificamente em estratégias de ordenação de classes e aspectos.

[1]Tecnicamente seria mais correto usar o termo teste de integração de "sistemas OA", uma vez que são necessários mais de uma classe e de um aspecto para que o teste de integração OA seja conduzido. No entanto, usar-se-á muitas vezes o termo teste de integração de "programas OA" para simplificar a redação.

1.2 Motivações

Durante o teste de integração de programas OO ocorre o problema da ordenação de classes quando existem ciclos de dependência (Kung *et al.*, 1995a). No contexto do teste de programas OA, alguns autores propõem o uso da estratégia incremental para a condução do teste (Ceccato *et al.*, 2005; Massicotte *et al.*, 2005; Zhou *et al.*, 2004). De maneira geral, a estratégia incremental determina que primeiramente são testadas as classes-base – teste de unidade e integração – e, posteriormente, os aspectos são testados individualmente e gradativamente integrados.

O principal argumento para a utilização da estratégia incremental é que a programação OA é uma extensão da programação OO, e os aspectos apenas adicionam comportamento às classe-base de maneira não invasiva. O argumento é baseado em um conceito da programação OA chamado inconsciência (*obliviousness*) (Kiczales *et al.*, 1997). Segundo esse conceito, as classes-base não são conscientes da ação dos aspectos. Portanto, os trabalhos dessa linha consideram que as classes não são dependentes dos aspectos.

Não obstante a estratégia incremental ser uma proposta aplicável nos casos em que o conceito da inconsciência é seguido, alguns trabalhos mostram evidências de que existem interdependências entre classes e aspectos e apenas entre aspectos, o que viola a propriedade da inconsciência. Especificamente no trabalho de Ceccato *et al.* (2005), existe menção ao problema da implementação frequente de *stubs* para simular o comportamento de aspectos dependentes de classes-base quando a estratégia incremental é usada. O mesmo problema é citado por Soares *et al.* (2002) e por Rashid e Chitchyan (2003), em que *stubs* são inicialmente implementados para simular persistência e distribuição, sendo posteriormente substituídos por implementações baseadas em aspectos em fases posteriores de desenvolvimento. Essa característica é denominada por Filman e Friedman (2005) de inconsciência incompleta. A inconsciência incompleta é explorada por Griswold *et al.* (2006). Esses autores propõem que os pontos em que os aspectos agem nas classes-base sejam explicitamente declarados na fase de projeto por meio de uma interface de entrecorte (XPI) para melhorar a modularidade de programas OA. Baseado nessa ideia, Kulesza *et al.* (2006) propõem a utilização de pontos de junção de extensão (do inglês, Extension Joint Points), que possibilitam a extensão sistemática de *frameworks* por meio do uso de aspectos de variabilidade. Douence *et al.* (2004) reafirmam o conceito da inconsciência incompleta e vão mais além ao afirmar que tratar a interação entre aspectos é um problema fundamental da programação OA. Além da afirmação de Douence *et al.* (2004), existem vários trabalhos que mostram a existência de interdependências entre aspectos (Cibrán *et al.*, 2003; Figueiredo *et al.*, 2008; Kienzle e Guerraoui, 2002; Kienzle *et al.*, 2003; Loughran e Rashid, 2004; The AspectJ Team, 2002). Alguns desses trabalhos, como os de Cibrán *et al.* (2003),

(Figueiredo *et al.*, 2008) e Loughran e Rashid (2004), utilizam aspectos para implementar interesses funcionais, o que aumenta as chances de ocorrência de ciclos de dependência.

Dessa forma, pode-se supor que a diretriz seguida pela estratégia incremental não produz bons resultados em todos os casos. Essa intuição é de certa forma reafirmada nos próprios trabalhos que propõem essa estratégia. Os trabalhos citados anteriormente não apresentam detalhes sobre como conduzir as atividades de teste e integração. As propostas não possuem diretrizes que sugerem qual deve ser a ordem de teste e, portanto, não mostram como ordenar as classes, e tampouco mostram como ordenar os aspectos. Além disso, não apontam soluções para problemas decorrentes da escolha de uma ordem de teste, como por exemplo, tratar interdependências entre aspectos, efetuar o teste de uma classe que depende de um aspecto, ou implementar *stubs* de aspectos.

Conclui-se, a partir da discussão desses trabalhos, que em projetos OA reais há interdependências entre classes e aspectos que podem causar ciclos de dependência. Portanto, o mesmo problema de ordenação no teste de integração de programas OO ocorre no teste de integração de programas OA: determinar a ordem de teste de classes e aspectos interdependentes com o objetivo de diminuir o custo dessa atividade por meio da minimização do número de *stubs*. Esta é, portanto, a motivação principal deste trabalho.

1.3 Objetivos

O objetivo geral desta tese é contribuir para melhorar a eficiência da atividade de teste de integração de programas OO e OA por meio da minimização do número de *stubs* necessários. Para atingir esse objetivo geral, três objetivos mais específicos são buscados nesta tese, que em conjunto auxiliam a atingir o objetivo geral.

O primeiro objetivo é definir um modelo que represente as dependências entre classes e aspectos a partir do qual se possa obter uma ordenação otimizada, usando-se um algoritmo eficiente, preferencialmente já existente. Além disso, pretende-se investigar e analisar diferentes estratégias de ordenação que considerem os novos relacionamentos entre classes e aspectos na descrição das dependências pelo modelo, visando a propor uma estratégia que seja melhor do ponto de vista de produzir o menor número de *stubs*.

O segundo objetivo é estabelecer um processo geral de mapeamento do modelo de projeto de um sistema modelado com base nos conceitos de separação de interesses para o modelo de dependências entre classes e aspectos, a partir do qual a ordenação possa ser obtida. Este objetivo é importante para a aplicação prática da estratégia de ordenação.

O terceiro objetivo é investigar como devem ser construídos *stubs* de aspectos e para o teste de aspectos a serem usados durante o teste de integração e propor um catálogo de *stubs* genéricos que aparecem de forma recorrente durante o processo de integração.

1.4 Organização do trabalho

No Capítulo 2 apresenta-se uma introdução ao teste de software que busca prover a fundamentação para o entendimento desta tese. Ainda nesse capítulo, o problema da ordenação de classes no teste de integração de programas OO é amplamente discutido. No Capítulo 3 é apresentada uma introdução ao desenvolvimento baseado em interesses para introduzir os conceitos básicos relacionados a esse assunto. Adicionalmente, apresenta-se também uma introdução ao teste de software nesse novo cenário, mostrando que alguns problemas relacionados ao teste de software persistem e são, algumas vezes, mais complexos. O objetivo principal desses dois capítulos é consolidar os conhecimentos sobre o problema da ordenação de classes e aspectos e sobre os conceitos da programação OA para concluir que esse problema está presente no teste de programas OA.

No Capítulo 4 é apresentado um modelo de dependências aspectuais – elaborado com base na sintaxe e semântica de AspectJ – uma extensão de um modelo de representação de dependências entre classes e aspectos e duas estratégias de ordenação de classes e aspectos. O modelo de dependências aspectuais é a base para o diagrama proposto e este é a base para as estratégias de ordenação propostas.

No Capítulo 5 é apresentado um processo de mapeamento que mostra como produzir um diagrama que representa as dependências de classes e aspectos a partir de um modelo de projeto. O processo é exemplificado por meio de sua aplicação à modelagem de um sistema usando a notação MATA (Whittle e Jayaraman, 2007), que usa como base alguns diagramas da UML (OMG, 2007). O processo tem dois objetivos: o primeiro é auxiliar no problema da ordenação e o segundo é validar o modelo de dependências, mostrando que ele é factível mesmo na fase de projeto de software.

No Capítulo 6 é apresentado e discutido um estudo exploratório de caracterização das estratégias de ordenação propostas. O estudo é usado para validar o diagrama que representa dependências e as estratégias propostas. Além disso, o estudo foi usado para a coletar e classificar amostras de *stubs* implementados durante sua condução.

No Capítulo 7 é apresentado um catálogo com uma amostra de casos de implementação de *stubs* e *drivers*. O catálogo mostra os principais casos de implementação de *stubs* e *drivers* ocorridos durante a condução do estudo de caracterização.

Por fim, no Capítulo 8 são apresentadas as contribuições dessa teste, juntamente com suas limitações e propostas de novos trabalhos.

CAPÍTULO

2

Introdução ao teste de software com ênfase para o teste de integração

2.1 Considerações iniciais

O teste de software é uma atividade importante do processo de desenvolvimento de software que visa a aumentar confiabilidade do produto desenvolvido. Apesar dos princípios da atividade de teste permanecerem os mesmos, abordagens técnicas e critérios de teste acompanham a evolução dos novos paradigmas de programação, fornecendo recursos específicos para tratar dos desafios emergentes. A programação OA – descrita na Seção 3.3 – é uma evolução da programação OO com características próprias. Dessa maneira, do ponto de vista de teste de software, ela apresenta problemas semelhantes à programação OO e também problemas relacionados aos seus novos conceitos. É importante, portanto, compreender os conceitos clássicos de teste de software e o estado em que se encontram as propostas relevantes ao teste de programas OO para entender e atacar os novos problemas advindos da programação OA.

O teste de software pode ser analisado segundo duas dimensões: em uma delas, chamada de técnicas e critérios, pode-se dispor o teste segundo a técnica ou critério utilizados para derivar os requisitos de teste: técnica funcional, estrutural, baseada em erros e baseada em modelos; em uma segunda dimensão, chamada de fases de teste de software, pode-se dispor o teste de software segundo o tamanho e a complexidade dos elementos do foco do teste: unidade, integração e sistema. Essa divisão clássica da atividade de teste com respeito à granularidade do teste de entidades dos programas é um ponto importante deste trabalho, uma vez que pretende-se abordar um problema que acontece durante o teste de integração de programas OA: a ordem de teste de classes e aspectos. Para tanto, dar-se-á ênfase nesta revisão às estratégias – ou abordagens – de ordenação de classes durante o teste de integração de programas OO para que seja possível atacar o problema da ordenação de classes e aspectos no teste de integração de programas OA.

Na Seção 2.2 é apresentada uma breve definição da atividade de teste, suas características básicas e seu papel como elemento constituinte das atividades de validação, verificação e teste. Ainda na mesma seção são descritas algumas características da programação OO que influenciam no teste de software. Subsequentemente, na Seção 2.3, são discutidas as principais técnicas e critérios de teste de software. Na Seção 2.4 é feita a apresentação da divisão clássica da atividade de teste com respeito à granularidade do teste de entidades de programas, no qual são descritos objetivos e características de cada uma das fases de teste, tanto de programas procedimentais quando de programas OO. Uma discussão aprofundada, mostrando a evolução e as diferentes características das estratégias de ordenação de classes no teste de software OO, é apresentada na Seção 2.5. Uma breve descrição das principais ferramentas de teste disponíveis é apresentada na Seção 2.6. Por fim, na Seção 2.7 são apresentadas as considerações finais.

2.2 Conceitos básicos de teste de software

As atividades de VV&T – verificação, validação e teste – fazem parte de um processo geral de garantia de qualidade de software, e visam a minimizar a ocorrência de erros inerentes ao processo de desenvolvimento de software, aumentando sua confiabilidade. O teste de software, como uma técnica integrante das atividades de VV&T, tem por objetivo encontrar e corrigir os problemas do produto de software resultantes dos enganos cometidos durante o processo de desenvolvimento de software, uma atividade humana que envolve uma séria de atividades de produção suscetíveis a falhas (Pressman, 2004). Outras atividades de VV&T usam análise estática para verificar as representações do produto de software e, apesar de se mostrarem eficientes, elas podem apenas checar a correspondência entre o software e sua especificação, não podendo demonstrar que o software é operacionalmente útil (Sommerville, 2004). Embora não se possa provar a corretude de um programa (Myers, 2004), as atividades de teste de software são um ponto crítico da garantia da qualidade de software, já que são responsáveis pela revisão final da especificação, do projeto e da codificação do software antes dele se tornar operacional (Pfleeger, 2001), mostrando-se relevante para a identificação e eliminação de erros que persistem (Maldonado, 1991).

Vários são os termos usados para denominar a relação de causa e efeito ligadas ao mau funcionamento de programas durante as atividades de teste. O padrão IEEE 610.12-1990 (IEEE, 1990) define precisamente os termos empregados para descrever o mau funcionamento e o comportamento indesejado observado em programas durante o teste: um “engano” (*mistake*) é uma ação humana que produz um resultado incorreto; um “defeito” (*fault*) é um passo, processo ou definição de dados incorreto; um “erro” (*error*) é a diferença entre o valor computado, obser-

vado ou medido, e o valor real; e uma "falha" (*failure*) é um resultado incorreto em relação a uma especificação. De forma geral, os erros presentes em um produto de software podem ser classificados como: erro computacional, que é provocado quando uma sequência de comandos esperada é executada, porém uma computação incorreta é obtida; e erro de domínio, que é provocado quando uma sequência de comandos diferente da sequência esperada é executada, ou por um erro computacional ou por uma condição incorreta de um comando de decisão. Essa não é uma terminologia unânime nas comunidades científica e profissional, mas é amplamente aceita como padrão.

Ao contrário das técnicas de VV&T que aplicam análise estática de programas, o teste de software tem o propósito de encontrar erros por meio da verificação dinâmica do produto, usando casos de teste que evidenciem defeitos e, consequentemente, resultem em um teste bem sucedido (Myers, 2004). O projeto dos casos de teste é um fator fundamental para o sucesso da atividade de teste porque eles devem ser representativos em relação ao domínio de entrada para cobrir a lógica do programa (Pfleeger, 2001), uma vez que é impraticável usar todo o domínio de entrada para testar um programa, e também para melhorar a qualidade do produto final, mesmo ao não encontrar defeitos (Maldonado, 1991).

Considerando as limitações inerentes à atividade de teste de software (Delamaro *et al.*, 2007), o fato de que é impraticável utilizar todo o domínio de entrada para avaliar os aspectos funcionais e operacionais de um produto em teste (Delamaro *et al.*, 2007) e que é extremamente difícil ou até impossível encontrar todos os defeitos de um software (Myers, 2004), evidencia-se a relevância de selecionar casos de teste que proporcionem ao mesmo tempo eficiência e baixo custo. Adicionalmente, a seleção de casos de teste torna-se mais complexa porque estabelecer o final da atividade de teste a partir da suficiência de um conjunto de casos de teste – ou simplesmente conjunto de teste – é um problema latente. Portanto, o conjunto de casos de teste deve ser suficientemente grande para oferecer uma medida satisfatória de confiança, porém não tão grande que torne impraticável a atividade de teste.

Um caso de teste é definido como um par $(d, S(d))$, em que d é um dado de teste do domínio de entrada D de um programa P em teste, e $S(d)$ corresponde à saída esperada da execução de P. Uma verificação completa de um determinado programa P poderia ser obtida testando-se P com um conjunto de casos de teste T que inclui todos os elementos D. Entretanto, como geralmente o conjunto de elementos do domínio é infinito ou muito grande, torna-se necessária a obtenção de subconjuntos desses casos de teste (Delamaro *et al.*, 2007). Para isso, podem ser utilizados critérios de teste que fornecem tanto um método para a avaliação de conjuntos de teste como uma maneira de selecionar casos de teste. Os critérios de teste devem indicar os casos de teste que devem ser aplicados para aumentar as chances de revelar defeitos ou, quando estes não forem revelados, estabelecer um nível elevado de confiança em relação ao programa

(Maldonado e Barbosa, 2003). Dessa forma, os diferentes critérios de teste têm caráter complementar, e é fundamental o estabelecimento de estratégias de teste capazes de explorar as vantagens de cada critério.

Na orientação a objetos a ênfase da programação é dada ao objeto, elemento básico que encapsula dados e funções, e interage com outros objetos por intermédio de troca de mensagens, gerenciando seus próprios estados e seu próprio comportamento (Sommerville, 2004). Em alguns casos, o comportamento encapsulado no objeto só pode ser determinado em tempo de execução, em razão de mecanismos inerentes do paradigma OO, como por exemplo, o uso do polimorfismo e a capacidade de ligação dinâmica (*dynamic binding*). Os objetos são instâncias agrupadas em classes que podem se relacionar diretamente de diferentes maneiras – herança, agregação e associação, entre outras – para prover os serviços que lhe são atribuídos. Por sua vez, as classes podem se apresentar fortemente acopladas, trabalhando em conjunto para fornecer uma funcionalidade, formando um aglomerado de classes (*cluster*) (Kung *et al.*, 1995a,b; Pressman, 2004). Um aglomerado de classes é formado por um conjunto de classes fortemente relacionadas que trabalha cooperativamente para realizar uma ou mais funções específicas (Chen, 2003).

Embora a linguagem de programação não seja o fator determinante na geração de defeitos, já que eles são resultado de enganos no desenvolvimento que independem da linguagem utilizada, os seus tipos são dependentes da linguagem de programação utilizada (Binder, 1999). As características próprias de cada linguagem e, consequentemente, o paradigma de programação, possuem diferentes particularidades que propiciam ao programador cometer enganos, porque elas apresentam diferentes riscos de falhas (*fault hazard*). Algumas das características da programação OO importantes para o contexto de teste são: o encapsulamento, porque o ocultamento de atributos torna difícil a verificação dos detalhes internos e, consequentemente, do estado dos objetos (Binder, 1999); a herança, responsável pela criação de riscos de defeito, tanto pela sua própria característica quanto pelo seu mau uso (Armstrong e Mitchell, 1994; Binder, 1999); o polimorfismo, porque traz indecidibilidade para o teste baseado em programa (Leavens, 1991; McGregor e Sykes, 2001); e classes abstratas e genéricas, porque classes abstratas não podem ser testadas diretamente e porque classes genéricas devem ser instanciadas com vários tipos diferentes para serem testadas de maneira satisfatória (Binder, 1999; Firesmith, 1996; Overbeck, 1994, 1995). Além dessas, outras características específicas da programação OO que propiciam maior risco de falhas são: invocações implícitas, coerções, iniciação incorreta de objetos e sequência de mensagens incorretas que levam objetos a estados inválidos (Vincenzi, 2004).

2.3 Técnicas e critérios de teste

As técnicas de teste fornecem diretrizes para projetar testes que exercitam a lógica interna dos componentes do software, assim como seus domínios de entrada e saída (Pressman, 2004). Diversos critérios de teste de software têm sido propostos com o objetivo de auxiliar a atividade de teste. Critérios de teste são derivados a partir de diferentes técnicas de teste: funcional, estrutural, baseada em defeitos e baseada em modelos (Delamaro *et al.*, 2007).

A técnica funcional, também chamada de teste caixa-preta, tem o objetivo de determinar se o programa satisfaz aos requisitos funcionais e não-funcionais encontrados na especificação. A técnica estrutural ou teste caixa-branca, por sua vez, concentra-se na cobertura de partes do código-fonte a partir dos casos de teste, sendo portanto baseada em uma implementação específica. Os critérios estruturais são baseados na idéia de que não se pode confiar em um programa se alguns elementos estruturais nunca foram executados durante a atividade de teste. O teste baseado em defeitos utiliza informações sobre os tipos de erros freqüentemente cometidos no processo de desenvolvimento de software para derivar os requisitos de teste. Nesse contexto, modelos de defeitos característicos à tecnologia de implementação do software geralmente são utilizados, sendo elaborados a partir da experiência dos engenheiros de software e de dados experimentais. Por fim, o teste baseado em modelos utiliza uma representação baseada em estados para modelar o comportamento do sistema ou unidade que será testada. Com base nesse modelo, critérios de geração de seqüências de teste podem ser utilizados no teste de programas.

2.3.1 Teste funcional

A geração de casos de teste na técnica funcional não é baseada em qualquer conhecimento dos detalhes de implementação do código, ao contrário, o comportamento de um componente em teste só pode ser determinado a partir de suas entradas e saídas relacionadas. A técnica funcional é direcionada para encontrar: omissão ou incorreções de funções, defeitos de interface, defeitos em estruturas de dados ou em acesso a dados externos, defeitos de desempenho e comportamento e defeitos de iniciação ou de finalização (Pressman, 2004).

Um ponto importante do teste funcional é que ele pode se aplicado em todas as fases de teste e com produtos desenvolvidos em qualquer paradigma de programação, pois não levam em consideração detalhes de programação (Delamaro *et al.*, 2007). Um dos problemas dessa técnica é a dependência da experiência do testador na geração de casos de teste, uma vez que eles precisam utilizar seu conhecimento do domínio de entrada para selecionar casos de teste com boa probabilidade de revelar defeitos (Sommerville, 2004). No entanto, o uso de uma abordagem sistemática de seleção de casos de teste com base em bons critérios é necessário

para complementar o conhecimento heurístico dos testadores. Os principais critérios de teste da técnica funcional são: Particionamento de Equivalência, Análise de Valor Limite, Teste funcional Sistemático, Grafo de Causa-Efeito e *Error-Guessing* (Delamaro *et al.*, 2007).

O teste funcional no paradigma OO tem o mesmo objetivo que o teste procedimental: utilizar a especificação para derivar requisitos de teste (Barbey e Strohmeier, 1994; Doong e Frankl, 1994; Harrold *et al.*, 1992; Kung *et al.*, 1998; Pezzè e Young, 2004). Isso faz com que as técnicas e os critérios funcionais utilizados no teste procedimental possam também ser utilizados no teste de programas OO. As técnicas baseadas em especificação variam de acordo com a notação e o nível de formalidade, permitindo, o mapeamento entre as entidades da especificação e do programa e, consequentemente, facilitando a geração de casos de teste. O teste funcional, assim como caracterizado no teste procedimental, pode ser aplicado no teste de unidade, de integração e de teste de sistema (Ammann e Black, 1999; Chen *et al.*, 1998, 2001; Dalal *et al.*, 1999; Doong e Frankl, 1994; Harrold *et al.*, 1992; Hartmann *et al.*, 2000; Kobayashi *et al.*, 2002; Paradkar *et al.*, 1997; Turner e Robson, 1995).

Note-se que, para melhorar as atividades de teste, algumas propostas combinam diferentes técnicas e critérios durante essa atividade (Chen *et al.*, 1998, 2001; Helm *et al.*, 1990). Os modelos de especificações feitas utilizando o paradigma OO, como por exemplo os diagramas UML (OMG, 2007), têm suas próprias características, possibilitando que novas abordagens de teste funcional sejam aplicadas (Tsai *et al.*, 1999b).

2.3.2 Teste estrutural

Um dos problemas relacionados ao teste funcional é que muitas vezes a especificação do programa é feita de modo descritivo e não formal. Dessa maneira, os requisitos de teste derivados de tais especificações são também, de certa forma, imprecisos e informais (Maldonado *et al.*, 1998). O teste estrutural usa uma abordagem complementar ao teste funcional para contornar esse problema, porque ele se preocupa com a extensão que os casos de teste atingem ao exercitar ou cobrir a lógica do programa. Na técnica estrutural os requisitos de teste são estabelecidos com base na implementação do programa e a estrutura e os detalhes do código são considerados para derivar casos de teste.

Em geral, os critérios da técnica estrutural utilizam o grafo de fluxo de controle – ou grafo de programa – para auxiliar a geração de casos de teste, uma vez que ele ilustra o fluxo de controle lógico do programa. Um programa pode ser decomposto em um conjunto de blocos disjuntos de comandos. A definição dos blocos baseia-se no fato de que a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem em que as instruções são dadas. Todos os comandos de um bloco, possivelmente com exceção

do primeiro, têm um único predecessor e exatamente um único sucessor, exceto, possivelmente, o último comando (Maldonado *et al.*, 1998).

Vários critérios de teste estrutural são encontrados na literatura. De forma geral, os critérios da técnica estrutural segundo Maldonado e Fabbri (2001) são três: baseados em fluxo de controle; baseados em fluxo de dados; e, baseados na complexidade. Zhu *et al.* (1997) incorporam os critérios baseados em complexidade aos critérios baseados em fluxo de controle, como uma forma de aprimoramento dos critérios baseados em fluxo de controle.

Um ponto que deve ser discutido é o uso de técnicas estruturais no teste de integração. Os critérios estruturais têm sido utilizados principalmente no teste de unidade, uma vez que os requisitos de teste por eles exigidos limitam-se ao escopo da unidade. Segundo Vincenzi (2004) várias propostas apresentam extensões de técnicas e critérios estruturais para o teste de integração e, o que diferencia esses critérios é o modelo de fluxo de dados no qual se baseiam. O modelo de fluxo de dados é que define precisamente o que caracteriza a definição e uso de variáveis em cada contexto. Várias propostas de extensão dos critérios de fluxo de dados para serem utilizados no teste de programas OO são encontradas na literatura (Boujarwah *et al.*, 2000; Chen e Kao, 1999; Harrold e Rothermel, 1994; Martena *et al.*, 2002; Sinha e Harrold, 1999; Souter e Pollock, 2003).

2.3.3 Teste baseado em erros

Nesta técnica, os tipos de defeitos mais frequentes encontrados no software são utilizados para derivar requisitos de teste. A ênfase da técnica está nos enganos que o programador pode cometer durante o desenvolvimento que resultem em defeitos e erros do software, e nas abordagens que podem ser usadas para detectar a sua ocorrência (Maldonado *et al.*, 1998). Nesse contexto, modelos de defeitos característicos à tecnologia de implementação do software geralmente são utilizados, sendo elaborados a partir da experiência dos engenheiros de software e de dados experimentais.

O critério de teste mais conhecido da técnica baseada em erros é a Análise de Mutantes (*Mutation Analysis*) ou Teste de Mutação, mas existe também a Semeadura de Erros (*error seeding*). Com relação às fases de teste, a técnica de mutação se mostra bastante flexível, e pode-se mudar o enfoque do teste, passando de teste de unidade para o teste de integração (Delamaro *et al.*, 2007).

Atualmente, vários pesquisadores têm trabalhado na definição de operadores de mutação para programas OO. Dentre eles, destacam-se os trabalhos de Kim *et al.* (1999, 2000), Bieman *et al.* (2001), Ma *et al.* (2002), Vincenzi (2004) e Chevalley (2001).

2.3.4 Teste baseado em modelos

Frequentemente o software é especificado por meio de modelos para evitar a ambiguidade e imprecisão da linguagem natural. Os modelos capturam o que é essencial do software descrevendo seu comportamento e características. Alguns modelos oferecem a possibilidade de simulação e, nesse caso, podem ser usados tanto como oráculo quando como base para a geração de casos de teste. Um oráculo decide se o resultado obtido a partir do uso de caso de teste é adequado ou não. Assim, um modelo deve ser testado tanto quando o próprio software. As Máquinas de Estado Finito (MEFs)) (Chow, 1978; Fujiwara *et al.*, 1991) são os modelos simuláveis mais simples (Delamaro *et al.*, 2007).

Nessa técnica, sequências de casos de teste são geradas por diferentes métodos, que podem avaliar se o comportamento esperado da máquina de estados é obtido (Vincenzi, 2004). Vários critérios de teste baseados em máquinas de estado podem ser encontrados na literatura (Chow, 1978; Fabbri *et al.*, 1994; Fujiwara *et al.*, 1991; Gönenç, 1970; Sabnani e Dahbura, 1988).

Segundo Barbey e Strohmeier (1994), o teste de programas OO difere do teste de programas procedimentais porque deve se concentrar no estado dos objetos que compõem o sistema, verificando sua consistência após a execução dos métodos. O estado de um objeto é dependente de seus atributos que, podem, eventualmente, ser instâncias de outras classes, cada uma com seu próprio estado. Por causa dessa característica, o teste baseado em estados descritos por modelos é amplamente utilizado no teste de programas OO (Binder, 1999; Gao *et al.*, 1995; Hoffman e Strooper, 1993; Kung *et al.*, 1996a; Pezzè e Young, 2004; Tonella, 2004; Tsai *et al.*, 1999a; Turner e Robson, 1993).

Segundo Binder (1999) as máquinas de estado podem ser utilizadas em qualquer fase de teste OO: classes, aglomerados de classes e subsistemas. Além disso, elas podem revelar defeitos de comportamento individual ou coletivo das entidades testadas. Porém, o teste baseado em estados não é capaz de revelar todos os tipos de defeitos, exigindo que critérios de teste baseado em programa sejam utilizados, visando maximizar a detecção de defeitos.

2.4 Fases de teste

Segundo Pressman (2004), o teste de software deve ser conduzido a partir dos menores elementos do programa e seguir em direção aos maiores elementos, integrando-os gradativamente. Isso deve ser feito para isolar a unidade em teste e encontrar defeitos específicos daquele ponto, sem a interferência de outras partes do código e, então, integrar as partes já testadas para encontrar defeitos provenientes da comunicação entre as unidades. Por fim, os testes de sistema devem ser

conduzidos em um ambiente operacional. Com isso, o testador pode se concentrar em aspectos diferentes do software e utilizar diferentes formas de seleção de casos de teste e medidas de cobertura em cada uma das fases. Nesse contexto, podem ser definidas três fases da atividade de teste: teste de unidade, teste de integração e teste de sistema. Existe ainda, uma outra fase de teste, chamada de teste de regressão. O teste de regressão é conduzido quando o software é alterado e precisa ser testado novamente para verificar se as alterações estão adequadas e se não adicionam novos defeitos.

Pequenas discordâncias são encontradas na literatura quanto à divisão das fases de teste para programas OO. No paradigma procedimental, as fases de teste são claramente identificáveis, já que as unidades possuem um relacionamento hierárquico – possibilitando que os testes sejam aplicados inicialmente à unidade, posteriormente à integração das unidade e, por fim, o sistema integrado. No paradigma OO pode não existir predominância hierárquica no relacionamento entre os elementos que compõem o software (Kung *et al.*, 1995a), os objetos interagem requisitando serviços e fornecendo serviços. Além disso os programas OO possuem características semânticas e sintáticas bastante diferentes, o que requer uma abordagem própria de teste dos elementos e de sua integração.

Segundo o padrão IEEE 610.12-1990 (IEEE, 1990), uma unidade é um componente de software que não pode ser subdividido. No paradigma procedimental, o teste de um procedimento é considerado teste de unidade, porque um procedimento é o menor elemento funcional; similarmente, no paradigma OO, pode-se considerar que o método é a menor unidade testável (Vincenzi, 2004). Além disso, um método não pode ser testado isoladamente, sem a classe à qual pertence, visto que, dinamicamente, ele é dependente de uma instância da classe. Nessa visão, durante o teste do método, a classe pode ser considerada um *driver* de teste[1], que contém todos os detalhes para que o método seja testado.

Alguns autores entendem que a classe é a menor unidade no paradigma OO ((Binder, 1999; McDaniel e McGregor, 1994; Perry e Kaiser, 1990)), e o teste de unidade envolveria os testes intramétodo, intermétodo e intraclasse proposto por Harrold e Rothermel (1994). A Tabela 2.1 de Vincenzi (2004) sintetiza as diferentes fases de teste considerando o método como menor unidade ou a classe como menor unidade.

Adicionalmente, outros níveis de teste OO podem ser definidos se abstrações frequentemente utilizadas no paradigma OO forem considerados, como por exemplo, aglomerados de classes e componentes de software. Independentemente da fase de teste, existem algumas etapas bem definidas para a execução da atividade de teste (Delamaro *et al.*, 2007; Pressman, 2004; Sommerville, 2004): planejamento, projeto de casos de teste, execução e análise dos resultados.

[1] O termo *driver* de teste é definido na Seção 2.4.1.

Tabela 2.1: Relação entre fases de teste de programas procedimentais e OO (Vincenzi, 2004).

Menor Unidade: Método		
Fase	Teste Procedimental	Teste OO
Unidade	Intraprocedimental	Intramétodo
Integração	Interprocedimental	Intermétodo, Intraclasse, Interclasse
Sistema	Toda a Aplicação	Toda a Aplicação

Menor Unidade: Classe		
Fase	Teste Procedimental	Teste OO
Unidade	Intraprocedimental	Intramétodo, Intermétodo e Intraclasse
Integração	Interprocedimental	Interclasse
Sistema	Toda a Aplicação	Toda a Aplicação

2.4.1 Teste de unidade

No teste de unidade o foco da atividade de teste concentra-se na menor unidade funcional do software, que podem ser testadas paralelamente. No teste de software do paradigma procedimental a menor unidade de teste é o procedimento, por isso essa fase de teste também pode ser chamada de teste intraprocedimental. O teste caixa preta pode ser utilizado na fase de teste de unidade, porém, o tipo de teste predominantemente utilizado nessa fase é o teste caixa branca.

O objetivo do teste de unidade é procurar erros de lógica e de programação por meio do fornecimento de dados de entrada e da checagem dos resultados computados pela unidade, verificando que a função especificada para ela é produzida satisfatoriamente. Para executar o teste de unidade, é necessário criar um ambiente que simule as condições de funcionamento da unidade. Como a menor unidade é o procedimento, é necessário simular o comportamento de um procedimento chamador, responsável por iniciar a execução da unidade em teste, denominado módulo pseudocontrolador (*driver* de teste). Eventualmente, um procedimento em teste pode chamar outros procedimentos, que também devem ser simulados e retornar valores fixos esperados, denominados módulos pseudocontrolados (*stub* de teste). O uso de *drivers* e *stubs* aumenta o custo do desenvolvimento, porque não são parte integrante do produto final. Segundo Pressman (2004) e Kung *et al.* (1995a), a implementação de *stubs* pode não ser uma tarefa trivial, dificultando consideravelmente a atividade de teste.

2.4.2 Teste de integração

No teste de integração as unidades testadas individualmente são gradativamente integradas e testadas, predominantemente utilizando o teste caixa preta (Pressman, 2004). Uma questão que

pode ser levantada logo após a conclusão dos teste de unidade diz respeito à necessidade de efetuar testes de integração em unidades que já foram testadas individualmente. Esse questionamento, aparentemente correto, expõe claramente o foco do teste de integração: exercitar as chamadas entre as unidades para encontrar defeitos em suas interfaces, porque o teste de unidade não pode garantir que cada unidade testada individualmente trabalhe adequadamente quando integradas (Delamaro, 1997).

As unidades podem ser integradas de forma aleatória, uma maneira de realizar o teste de integração que recebe o nome de integração *bing bang* (Myers, 2004). Frequentemente, o resultado desse tipo de abordagem de integração não apresenta um resultado satisfatório (Myers, 2004; Pfleeger, 2001). Apesar de vários defeitos serem encontrados, rastrear e isolar as causas do problema e corrigir os defeitos torna-se uma tarefa complexa, por causa da possível complexidade de interação ente as unidades e ao caráter recursivo da atividade, porque à medida que os defeitos são corrigidos, novos defeitos aparecem, iniciando um novo ciclo do processo (Myers, 2004; Pressman, 2004; Sommerville, 2004). Duas maneiras bastante conhecidas de realizar o teste de integração das unidades são a integração descendente (*top-down*) e a integração ascendente (*bottom-up*) (Myers, 2004). Além dessas, várias outras podem ser encontradas na literatura (Binder, 1999; McGregor e Sykes, 2001).

2.4.3 Teste de sistema

Após os teste de integração existe mais uma fase de teste, denominada teste de sistema, com vários objetivos e propósitos diferentes (McGregor e Sykes, 2001; Myers, 2004; Pfleeger, 2001; Pressman, 2004). Os testes de sistema devem ser feitos com o intuito de garantir que o sistema funcione de acordo com as especificações feitas pelos usuários e utiliza-se basicamente o teste caixa preta, já que a geração de casos de teste depende em grande parte dos requisitos do sistema. Vários tipos de teste podem ser conduzidos, como por exemplo, teste de desempenho, teste de unidade e teste de segurança, dependendo do plano de teste estabelecido no início do desenvolvimento. Portanto, além da validação de requisitos funcionais do produto de software, o teste de sistema deve também verificar o cumprimento de requisitos não funcionais previamente estabelecidos.

2.5 Estratégias de ordenação de classes no teste de integração OO

Conforme brevemente discutido na Seção 2.4, existem vários níveis de integração no teste de programas OO. O objetivo do teste intermétodo não difere do objetivo do teste de unidade: detectar defeitos que não ultrapassem a fronteira do módulo. Em contrapartida, o teste de integração torna-se mais complexo em função das características próprias do paradigma OO. Além disso, várias classes podem ter vários métodos com, geralmente, pouca implementação, que interagem de acordo com os objetivos especificados, fazendo com que o número de unidades envolvidas na execução de uma funcionalidade e a complexidade de chamadas de métodos seja grande. Isso faz com que maior atenção seja dada à integração das unidade que compõem o software.

Vários trabalhos que atacam diferentes problemas do teste de integração de programas OO podem ser encontrados na literatura (Alexander e Offutt, 2000; Harrold *et al.*, 1992; Hartmann *et al.*, 2000; Labiche *et al.*, 2000; McDaniel e McGregor, 1994; McGregor e Korson, 1994). Um dos problemas encontrados durante o teste de integração é a ordem em que as classes são testadas (Tai e Daniels, 1999), já que isso influencia a ordem em que as classes são desenvolvidas, a ordem em que os erros são encontrados e o número de *stubs* e *drivers* implementados. O problema de ordenação das classes surge quando o sistema a ser testado é constituído de classes que possuem ciclos de dependência. Várias propostas foram feitas para ordenar a implementação e teste de classes com a intenção de minimizar o número de *stubs* e, consequentemente, minimizar o esforço durante a atividade de teste (Abdurazik e Offutt, 2006; Badri *et al.*, 2005; Briand *et al.*, 2002, 2003; Jaroenpiboonkit e Suwannasart, 2007; Kung *et al.*, 1995a,b; Le Hanh *et al.*, 2001; Le Traon *et al.*, 2000; Lima e Travassos, 2004; Malloy *et al.*, 2003; Mao e Lu, 2005; Tai e Daniels, 1999). Esse é um problema porque, segundo Kung *et al.* (1995a,b), embora a geração automática de casos de teste e a geração automática de *drivers* seja viável – respeitando as limitações de cada técnica –, não se pode automatizar o processo de geração de *stubs*. Os *stubs* simulam comportamentos específicos para cada caso de teste derivados a partir da análise semântica dos módulos reais. Simular o comportamento de um *stub* é difícil e custoso porque o testador tem que compreender e simular o comportamento de membros da classe, incluindo o estado e comportamento dos objetos, inclusive na presença de hierarquias de herança e polimorfismo (Malloy *et al.*, 2003).

2.5.1 Estratégia de Kung *et al.*

O trabalho de Kung *et al.* (1995a) foi um dos primeiros a apresentar uma solução para o problema de ciclos de dependência e discutir esse tipo de problema no teste de unidade e no teste de integração para programas OO. O problema surge do relacionamento, muitas vezes complexo, de classes que compõem um programa OO. Para resolver o problema, os autores propuseram um diagrama chamado ORD, ilustrado na Figura 2.1, para representar as dependências dos relacionamentos de herança "I", de agregação "Ag" e de associação "As" entre classes. O ORD é um dígrafo no qual os vértices representam as classes e as arestas representam os relacionamentos entre as classes. As arestas que representam relacionamento de associação entre as classes podem ser removidas, e as outras não, já que herança e agregação apresentam além do acoplamento de controle, acoplamento de dados e dependência de código.

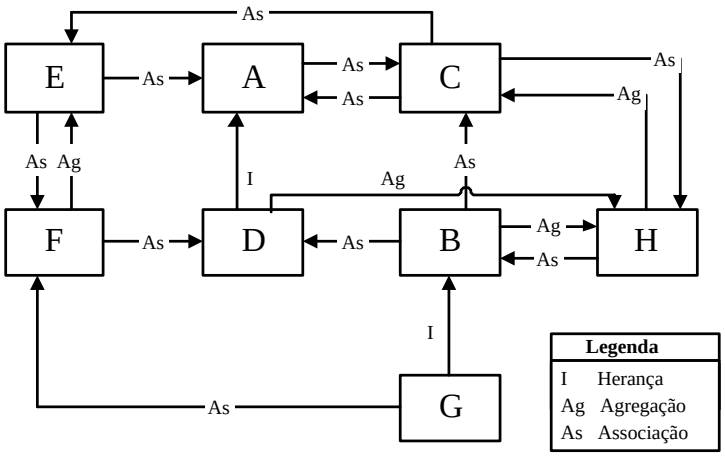


Figura 2.1: Exemplo do ORD estudado por Briand *et al.* (2003).

A proposta de Kung *et al.* (1995b) foi um aprimoramento de um trabalho anterior (Kung *et al.*, 1995a), direcionado para o teste de regressão de programas OO. No modelo de mapeamento proposto, as construções sintáticas da linguagem em questão não podem ser mapeadas de maneira automatizada para o ORD (Kung *et al.*, 1995a,b). Isso acontece porque os relacionamentos de herança podem ser facilmente identificados a partir do código fonte, mas não se pode diferenciar sintaticamente um relacionamento de agregação de um relacionamento de associação apenas por intermédio de código fonte (Briand *et al.*, 2002, 2001). O ORD pode ser construído de duas formas: com base em engenharia reversa do código fonte e com base na especificação. Um ORD pode ser construído automaticamente da especificação do nível de projeto, desde que a especificação diferencie cada uma das associações, como é feito, por exemplo, na UML, (Clarke e Malloy, 2005; Labiche *et al.*, 2000; Lima e Travassos, 2004). A estratégia apresentada por Kung *et al.* (1995a,b) é baseada em dois conceitos principais: aglomerado de

classes, que é o conjunto com o máximo de vértices mutuamente alcançáveis no ORD, também denominado componente fortemente conexa (SCC, *Strongly Connected Component*); e a quebra de ciclos, que é a remoção temporária de uma aresta com o objetivo de tornar o ORD acíclico. Um *stub* deve ser implementado para cada aresta temporariamente removida.

O teste é executado diferentemente segundo dois casos: quando o ORD é acíclico e quando o ORD é cíclico. No primeiro caso, uma ordenação topológica reversa é usada para estabelecer a ordem em que as classes devem ser testadas. No entanto, quando existe dependência cíclica, a estratégia é um pouco mais refinada. Pode existir mais de um SCC em um ORD cíclico, SCCs podem ser triviais, quando são compostos de apenas um vértice, e não triviais, quando são compostos por mais de um vértice. A ideia usada pela maioria das abordagens de ordenação de classes no teste de integração, inclusive Kung *et al.* (1995a,b), é remover temporariamente arestas para quebrar os ciclos dos ORDs e transformá-los em ORDs acíclicos.

Os autores apoiam-se na definição de que um dígrafo cíclico pode ser transformado em um dígrafo acíclico. Um ORD pode ser representado por um dígrafo $G = (V, L, E)$, no qual G é o dígrafo, V é o conjunto finito de vértices representando as classes do ORD, $L = \{I, Ag, As\}$ é o conjunto de rótulos das arestas que representam relacionamentos entre classes – herança, agregação e associação – e $E \subseteq V \times V \times L$ é o conjunto de arestas direcionadas rotuladas. O dígrafo cíclico $G = (V, L, E)$ pode ser transformado no dígrafo acíclico $G' = (V', L', E')$, de forma que V' seja o conjunto de SCCs pertencente a G, E' seja o conjunto de arestas entre os SCCs de G e L' seja o conjunto de rótulos das arestas de E'. Nesse caso, uma ordenação topológica reversa pode ser aplicada para derivar a ordem de teste. Os passos do algoritmo são:

1 Transformar um dígrafo cíclico G em um dígrafo acíclico G';

2 Encontrar a ordem de teste maior, dada por uma ordenação topológica reversa em G';

3 Para cada SCC não trivial de G':

   3.1 Para cada ciclo de G', deve-se selecionar e remover arestas para quebrar o ciclo em questão;

   3.2 Aplicar uma ordenação topológica reversa no dígrafo G' cujas arestas foram removidas, resultando em uma ordenação chamada de ordem de teste menor.

Para Kung *et al.* (1995a,b), o esforço é diretamente proporcional ao número de *stubs* implementados para o teste. No entanto, a estratégia não estabelece regras para a escolha da aresta a ser removida. Isso é feito aleatoriamente, o que pode ocasionar a implementação de uma grande quantidade de *stubs*.

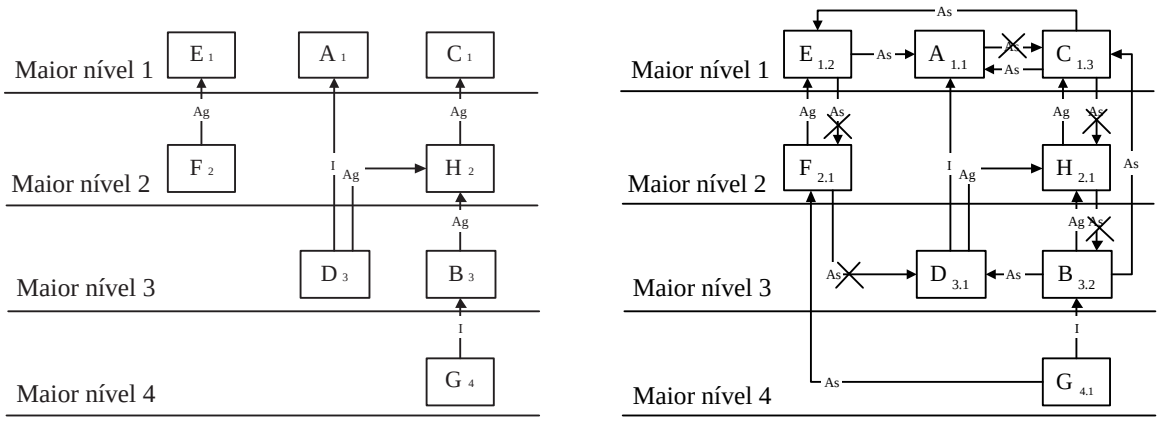
2.5.2 Estratégia de Tai e Daniels

O trabalho de Tai e Daniels (1999) refina a técnica proposta por Kung *et al.* (1995a,b). O algoritmo que seleciona arestas calcula um valor de maior nível para cada classe, baseado apenas em herança e agregação, que não formam ciclos, e posteriormente, adiciona as arestas que representam associações, calculando para cada classe um número de menor nível. O algoritmo remove arestas segundo dois critérios: a determinação do peso das arestas, calculado a partir das arestas de chegada no vértice de origem e das arestas de saída do vértice de destino; e a remoção sistemática das arestas de associação que ligam vértices com diferentes valores de maior nível, o que em alguns casos causa uma solução subótima ao indicar a geração de *stubs* desnecessários.

O número de maior nível de um grafo G de um ORD qualquer é associado segundo as seguintes definições:

- $Maior(G, 1)$ é o conjunto de todos os vértices que não possuem arestas de saída de herança ou agregação presentes no grafo G e, portanto, seu número de maior nível é 1; e
- $Maior(G, i)$ é o conjunto de todos os vértices com arestas de saída de herança ou agregação para um vértice pertencente ao conjunto $Maior(G, j)$, para $j < i$, e com pelo menos uma aresta de saída de herança ou agregação para um vértice pertencente ao conjunto $Maior(G, i - 1)$.

A Figura 2.2(a) apresenta um ORD parcial, derivado do ORD da Figura 2.1 após o cálculo dos números de maior nível.



(a) Cálculo do número de maior nível.

(b) Cálculo do número de menor nível.

Figura 2.2: Calculo dos números de maior e menor nível e remoção de arestas para quebrar ciclos de dependência.

O cálculo do número de menor nível, ilustrado na Figura 2.2(b), é feito obedecendo às seguintes regras:

1 Seja $Menor(G, i)$ o subgrafo de G que contém apenas vértices que pertencem a $Maior(i)$ e arestas de associação entre esses vértices, e i um número de maior nível;

2 Seja i o número de maior nível, j o número de menor nível e $i.j$ o número de nível para cada aresta em $Menor(G, i)$;

3 Existem três opções:

- 3.1 $Menor(G, i)$ contém apenas um vértice e seu menor nível é 1;
- 3.2 $Menor(G, i)$ contém dois ou mais vértices cujas arestas formam ciclos e, nesse caso, uma ordenação topológica reversa pode ser feita para atribuir o número de menor nível a cada aresta;
- 3.3 $Menor(G, i)$ contém um ou mais vértices cujas arestas formam ciclos, e os seguintes passos devem ser seguidos:
  - 3.3.1 Deve-se identificar SCCs em $Menor(G, i)$;
  - 3.3.2 Deve-se remover algumas arestas ordenadamente de cada SCC em $Menor(G, i)$ com o objetivo de quebrar ciclos da seguinte maneira:
    - A. Para toda aresta $a \in Menor(G, i)$, seja e o número de arestas de entrada do nó origem de a, e s o número de arestas de saída do nó destino de a, o peso da aresta a, denominado $Peso(a)$, é calculado como: $Peso(a) = e \times s$;
    - B. Deve-se remover cada aresta a com maior valor dado por $Peso(a)$, até que os ciclos sejam quebrados;
  - 3.3.3 Deve-se aplicar uma ordenação topológica reversa em $Menor(G, i)$ para associar para cada classe um número de menor nível, de acordo com a ordenação.

As classes devem ser integradas segundo uma busca, iniciada primeiramente pelos número de maior nível em ordem crescente e, posteriormente, pelos números de menor nível em ordem crescente. Assim, a ordem de integração para as classes representas no ORD da Figura 2.2 e os *stubs* necessários são apresentados na Tabela 2.2.

Pode-se concluir que a estratégia proposta por Tai e Daniels (1999) é significativamente melhor que a estratégia proposta por Kung *et al.* (1995a,b). No entanto, ela apresenta solução subótima, porque as arestas cujo vértice de origem está em um maior nível i e o vértice de destino está em um maior nível j, para $i < j$, são removidas sistematicamente, como é o caso das arestas: (E, F), (C, H), (H, B) e (F, D). Conforme pode ser visto, isso gera *stubs* desnecessários.

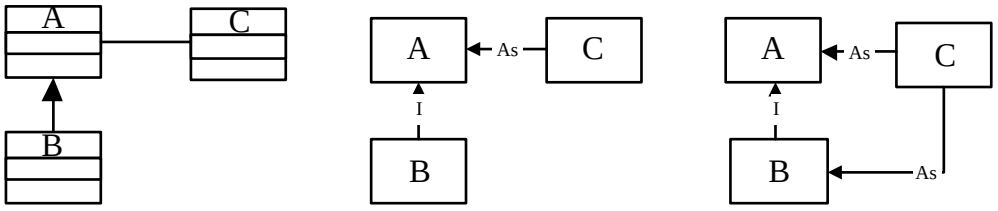
Número de Maior Nível	Número de Menor Nível	Classe em teste
1	1	`A` com *stub* de `C`
	2	`E` com *stub* de `F`
	3	`C` com *stub* de `H`
2	1	`F` com *stub* de `D`
		`H` com *stub* de `B`
3	1	`D`
	2	`B`
4	1	`G`

Tabela 2.2: Ordem de teste e número de *stubs* necessários da estratégia de Tai e Daniels (1999).

2.5.3 Estratégia de Le Traon *et al.*

Le Traon *et al.* (2000) propõem um outro tipo de representação gráfica das dependências entre classes, denominada TDG (*Test Dependency Graph*). Os tipos de relacionamento que podem ser representados no TDG são os mesmos que podem ser representados no ORD. No entanto, o TDG apresenta algumas diferenças em relação ao ORD:

- É possível representar não somente a dependência entre classes, mas também a dependência entre métodos das classes, mesmo para classes diferentes;
- O mapeamento de relacionamentos polimórficos de agregação e de associação é diferente, conforme pode ser observado na Figura 2.3.



(a) Diagrama UML de um conjunto de classes.

(b) ORD correspondente ao conjunto de classes.

(c) TDG correspondente ao conjunto de classes.

Figura 2.3: Diferenças de mapeamento de relacionamentos polimórficos a partir de um Diagrama UML entre o TDG e o ORD.

A estratégia de Le Traon *et al.* (2000), denominada Triskell (Le Hanh *et al.*, 2001), usa o algoritmo de Tarjan (1972) para identificar SCCs triviais e não triviais. O algoritmo de Tarjan baseia-se na busca em profundidade e é aplicado recursivamente em cada SCC não trivial depois da exclusão de uma ou mais arestas que quebram ciclos dentro do SCC. A ordem de teste é

dada pela aplicação recursiva do algoritmo de Tarjan e da remoção de arestas. Contudo, o resultado da aplicação de Triskell é não determinístico porque seu algoritmo escolhe aleatoriamente um vértice para iniciar a ordenação. Portanto, existem várias possibilidades de ordenação das classes. No exemplo, apresentado na Figura 2.4 e na Figura 2.5, escolhe-se como vértice inicial o vértice A e aplica-se o algoritmo de Tarjan. Na proposta, o algoritmo de Tarjan é ligeiramente modificado para, ao fazer a travessia do grafo, calcular um fator de importância para os vértices do grafo. O cálculo é feito a partir da soma de todas as arestas *frond*, de entrada ou de saída, para cada nó do grafo. Arestas *frond*, denotadas pelas linhas tracejadas nas figuras, são aquelas que durante a travessia vão de um vértice v qualquer para um vértice w antecessor. O resultado do cálculo para a aplicação do algoritmo de Tarjan é: $A = 3$, $B = 1$, $C = 3$, $D = 2$, $E = 2$, $F = 1$, $G = 0$ e $H = 2$. A heurística para remoção de arestas baseada do cálculo é: escolher um vértice que participe do maior número possível de ciclos, que no caso é A. O próximo passo é remover todas as arestas de entrada de A, que provoca a eliminação de alguns SCCs, resultando no segundo grafo da Figura 2.4.

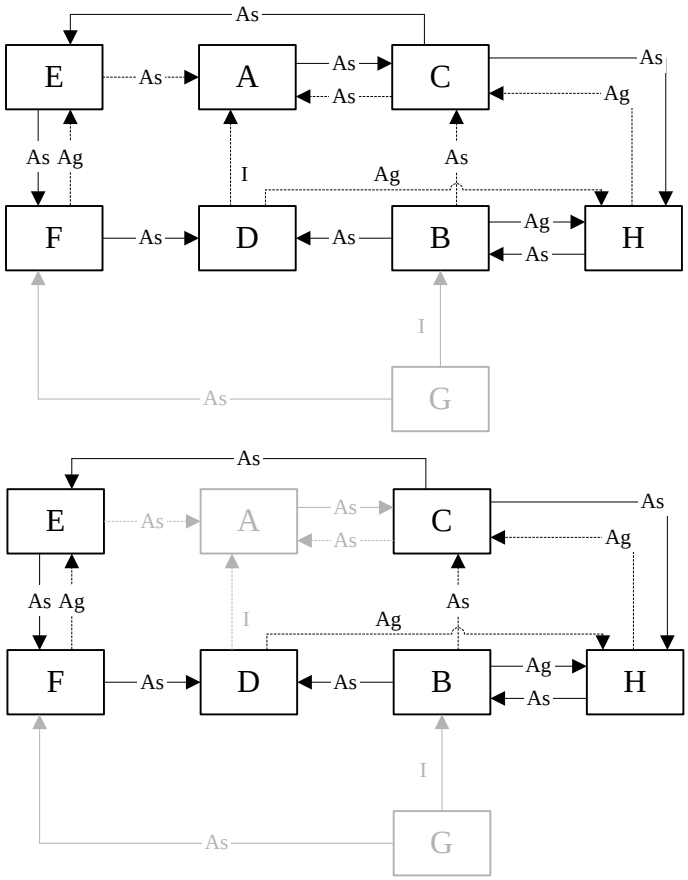


Figura 2.4: Exemplo parcial da aplicação de Triskell no ORD da Figura 2.1.

O algoritmo de Tarjan é aplicado recursivamente e as arestas do vértice escolhido são removidas até que não haja mais SCCs. As aplicações seguintes do algoritmo resultam nos grafos mostrados na Figura 2.5. A ordem de teste e integração é dada pelos vértices excluídos a cada aplicação do algoritmo. A ordem completa e os *stubs* necessários são apresentados na Tabela 2.3;

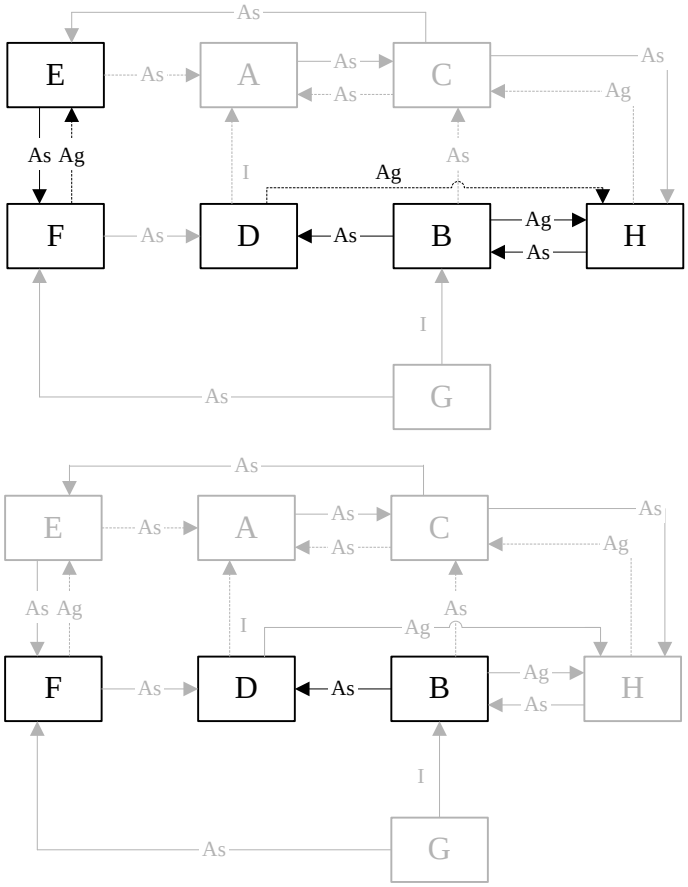


Figura 2.5: Exemplo de aplicação da estratégia de Le Traon *et al.* (2000) no ORD da Figura 2.1.

A abordagem de Le Traon *et al.* (2000) apresenta dois pontos bastante diferentes de Kung *et al.* (1995a,b) e de Tai e Daniels (1999): arestas que representam herança ou agregação podem ser removidas para que os ciclos sejam quebrados, uma vez que, para os autores, ciclos podem ser formados apenas em presença de agregação e herança; e o algoritmo é feito para minimizar o número de *stubs* realísticos ao invés de *stubs* específicos.

Os *stubs* específicos são diferentes dos *stubs* realísticos porque são implementados para fornecer somente os serviços que cada cliente necessita individualmente. Os *stubs* realísticos simulam todos os serviços que a classe original deve implementar, não importando qual cliente está sendo testado (Beizer, 1990; Le Hanh *et al.*, 2001).

Ordem	Classe em teste
1	`A` com *stub* de `C`
2	`C` com *stub* de `E` e de `H`
3	`H` com *stub* de `B`
4	`E` com *stub* de `F`
5	`F`
6	`D`
7	`B`
8	`G`

Tabela 2.3: Ordem de teste e número de *stubs* necessários da estratégia de Le Traon *et al.* (2000).

É importante ressaltar que a estratégia é não determinística e que, dependendo do vértice inicial que foi escolhido, o algoritmo pode apresentar resultados diferentes e subótimos. Outro ponto de indeterminísmo acontece quando mais de um vértice possui o mesmo peso. Nesse caso a escolha deve ser aleatória, fazendo com que *stubs* desnecessários sejam implementados.

2.5.4 Estratégia de Briand *et al.*

Briand *et al.* (2003) apresentam uma estratégia que utiliza algumas características das propostas de Le Traon *et al.* (2000) e Tai e Daniels (1999): usa o algoritmo de Tarjan recursivamente para identificar SCCs e associa pesos às arestas que representam dependência de associação para indicar quais devem ser removidas e quebrar os ciclos de dependência. A estratégia minimiza o número de *stubs* específicos, ao contrário de Le Traon *et al.* (2000), e o resultado não é subótimo como Tai e Daniels (1999) e Le Traon *et al.* (2000). No entanto, existe um ponto de indeterminísmo, pois quando duas arestas têm o maior peso a escolha para a remoção de uma delas é aleatória. Esse indeterminísmo não afeta o número de *stubs* implementados, apesar de alterar a ordem de teste.

A estratégia inicia com a aplicação do algoritmo de Tarjan em um grafo construído a partir de um ORD. Deve-se, recursivamente, remover arestas dos SCCs não triviais para quebrar os ciclos e aplicar o algoritmo de Tarjan. A aresta que deve ser removida é aquela que apresenta maior peso, obtido por meio da multiplicação das arestas de entrada do vértice de origem pelas arestas de saída do vértice de destino de cada aresta de associação. Esses passos devem ser seguidos até não existirem mais SCCs não triviais. A estratégia tem seu foco nas arestas, ao contrário da proposta de Le Traon *et al.* (2000) que tem o foco nos vértices.

A Figura 2.6 mostra um exemplo da aplicação da estratégia, em que é possível observar um ORD e as arestas que foram removidas para a quebra dos ciclos. Nesse exemplo, após a aplica-

ção inicial do algoritmo de Tarjan são encontradas o SCC não trivial $\{A, C, E, F, D, H, B\}$ e o SCC trivial $\{G\}$. O peso das arestas é, então, calculado:

$(H, B) = 3 * 3 = 9$ $(B, D) = 1 * 2 = 2$ $(B, C) = 1 * 3 = 3$ $(A, C) = 3 * 3 = 9$
$(C, A) = 3 * 1 = 3$ $(C, E) = 3 * 2 = 6$ $(E, A) = 2 * 1 = 2$ $(E, F) = 2 * 2 = 4$
$(F, D) = 1 * 2 = 2$ $(C, H) = 3 * 2 = 6$

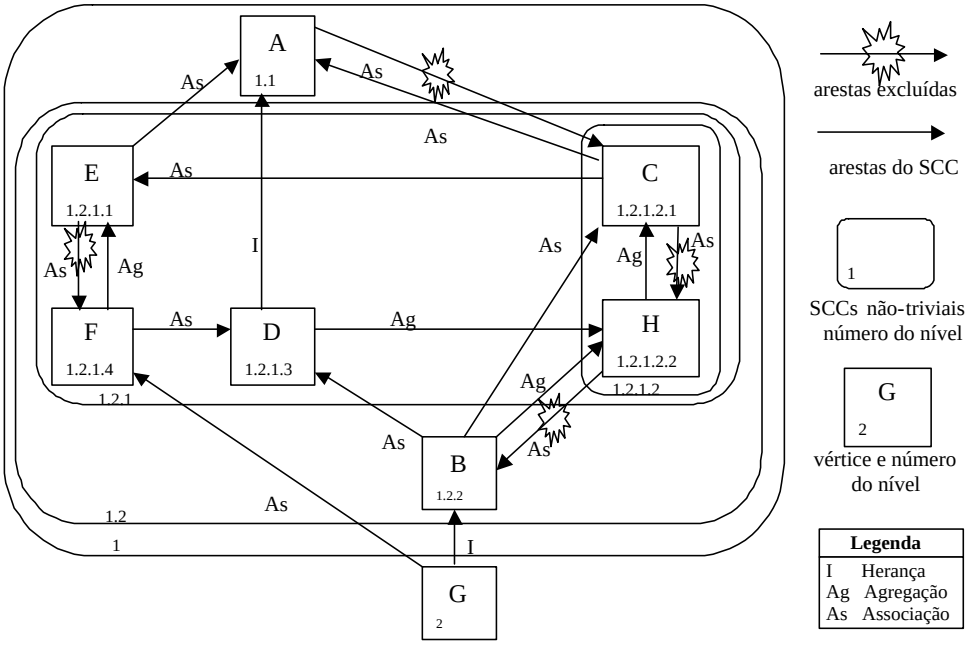


Figura 2.6: Aplicação da estratégia de Briand *et al.* (2003).

ORD={A, B, C, D, E, F, G}
1. {A, C, E, F, D, H, B} : aresta de maior peso $(A, C) = 9$
  1.1 {A}
  1.2 {B, C, E, F, D, H} : aresta de maior peso $(H, B) = 9$
    1.2.1 {C, E, F, D, H} : aresta de maior peso $(E, F) = 4$
      1.2.1.1 {E}
      1.2.1.2 {C, H} : aresta de maior peso $(C, H) = 1$
        1.2.1.2.1 {C}
        1.2.1.2.2 {H}
      1.2.1.3 {D}
      1.2.1.4 {F}
    1.2.2 {B}
2. {G}

Figura 2.7: Aplicação recursiva do algoritmo de Tarjan.

Tabela 2.4: Ordem de teste e número de *stubs* necessários da estratégia de Briand *et al.* (2003).

Ordem	Classe em teste
1	`A` com *stub* de `C`
2	`E` com *stub* de `F`
3	`C` com *stub* de `H`
4	`H` com *stub* de `B`
5	`D`
6	`F`
7	`B`
8	`G`

No exemplo, a remoção da aresta (A, C) ou da aresta (H, B) é indiferente, pois como ambas possuem o mesmo peso geram o mesmo número de *stubs*. Ao se eliminar a aresta (A, C) o algoritmo de Tarjan deve ser aplicado ao SCC $\{A, C, E, F, D, H, B\}$, que gera o SCC não trivial $\{B, C, E, F, D, H\}$ e o SCC trivial $\{A\}$. Novamente, uma aresta do SCC não trivial deve ser eliminada para quebrar os ciclos. Por fim, a sequência de aplicação do algoritmo de Tarjan de acordo com a eliminação das arestas resulta na ordem de implementação e teste das classes, mostrada na Figura 2.7. Na Tabela 2.4 apresentam-se os *stubs* necessários para implementar as classes.

2.5.5 Outras estratégias de ordenação de classes no teste de integração

Labiche *et al.* (2000) apresentam propostas de teste de integração que levam em consideração o uso de polimorfismo na ordem de integração das classes, assim como na proposta de Le Traon *et al.* (2000). Especificamente, Labiche *et al.* (2000) baseiam-se no polimorfismo e na presença de classes abstratas para propor uma técnica de ordenação de classes. A estratégia remove primeiramente arestas estáticas, que não levam em consideração o polimorfismo, e posteriormente, as arestas dinâmicas. Os autores denominam essa diferenciação de níveis de teste. Segundo eles, a estratégia minimiza o número de *stubs* levando em consideração o teste de classes abstratas. Na prática, há um aumento do número de ciclos por incluir no ORD arestas que não são necessárias ao teste, porque são consideradas todas as dependências polimórficas e não àquelas que ocorrem apenas em presença de classes abstratas. A estratégia apresenta um estudo feito em um sistema real, mas compara seus resultados com o trabalho de Kung *et al.* (1995a), que usa um algoritmo que remove arestas de maneira aleatória.

Como o problema de ordenação de classes é um problema NP-Completo (Le Traon *et al.*, 2000), existem propostas de ordenação que utilizam algoritmos de otimização semirrandômicos, como os algoritmos genéticos propostos por Le Hanh *et al.* (2001) e por Briand *et al.* (2002). Essas duas propostas diferem-se das demais porque não utilizam fortemente o uso teoria dos grafos para solucionar o problema. O foco deste trabalho são algoritmos baseados em teoria dos grafos e, por isso, as duas estratégias não são discutidas detalhadamente nesta revisão.

Lima e Travassos (2004) propõem uma técnica de ordenação de classes para o teste de integração baseado em diagramas de classe da UML. A estratégia é diferenciada em relação às anteriores porque trabalha com um modelo de representação com nível de abstração mais alto que o ORD, o próprio diagrama de classes. Um conjunto de critérios que determina a precedência entre as classes é proposto para estabelecer a ordem de integração das classes durante o teste. Para viabilizar a aplicação dos critérios de precedência e estabelecer a ordem de prioridade foram definidas duas propriedades: fator de influência (FI), e fator de integração tardia (FIT), cujo cálculo indica a ordem de integração das classes. O processo é recursivo e a cada passo uma classe é excluída do diagrama de classes para quebrar ciclos e o cálculo dos fatores é refeito até que todos os ciclos tenham sido eliminados. O ponto fraco da proposta é que ela não apresenta estudos com casos reais, apenas exemplifica sua aplicação em um exemplo proposto por Briand *et al.* (2003).

Badri *et al.* (2005) apresentam uma abordagem que estende o ORD para um modelo chamado CDM (*Class Dependency Model*) para representar as dependências entre as classes. O CDM possui arestas para representar: relacionamento de herança "I" e relacionamento de uso "U". Nesse modelo também são representados os métodos que causam as dependências entre as classes. Desse fato, os autores identificam dois tipos de ciclos no CDM: ciclos de dependência efetiva e ciclos de de dependência não efetiva. Os ciclos de dependência efetiva são caracterizados quando, por exemplo, uma método $m1$ da classe A tem uma referência para um método $m2$ da classe B, e o método $m2$ da classe B tem uma referência para o método $m1$ da classe A. O ciclo de dependência não efetiva é caracterizado quando uma classe A possui dependência da classe B, por exemplo da seguinte forma: um método $m1$ da classe A tem um referência a um método $m2$ da classe B e um método $m3$ da classe B tem uma referência ao método $m4$ da classe A. A quebra dos ciclos é iniciada com a mesma estratégia usada por Tai e Daniels (1999), mas considera o fato de que as dependências são diferentes. Para efetuar o teste, pode-se usar o conceito de desativação do método. A dependência não efetiva permite a definição do conceito de integração parcial, no qual a integração é feita por meio da desativação de métodos não necessários para o teste do método em questão. Quando a dependência é não efetiva pode-se desativar os métodos que não fazem parte do ciclo para efetuar a integração.

As classes são categorizadas segundo um número de maior nível, calculado por meio dos relacionamentos de herança. Posteriormente, dependência dos outros tipos são adicionados ao modelo. Os ciclos de dependência efetiva de cada um dos níveis é quebrado com base no cálculo semelhante ao cálculo de fator de influência proposto por Lima e Travassos (2004), isto é, o número de classes de que uma classe em questão depende. A classe com o maior valor é simulada com um *stub* e suas arestas de chegadas são eliminadas do grafo. Esse processo é refeito até que não haja mais ciclos de dependência efetiva. Posteriormente, os ciclos de dependência não efetiva são considerados seguindo os mesmos passos.

O trabalho de Badri *et al.* (2005) apresenta um estudo feito com 4 sistemas que mostra que não foi necessário implementar *stubs* quando a técnica é utilizada. Segundo os autores, esse resultado foi possível porque não haviam dependências efetivas nos sistemas estudados. O ponto fraco da estratégia é o conceito de desativação de métodos, que desconsidera o fato de que o estado de um objeto é importante para execução dos métodos e para o teste. Além disso, note-se que quando um método é desativado para remover uma dependência ele pode ser necessário para a execução de outros métodos que precisam estar presentes no teste de uma dependência não desativada. Por exemplo, o método `A.m1()` depende de `B.m2()` que utiliza o método `B.m3()`. Por sua vez, o método `B.m3()` depende do método `A.m4()`. Segundo a estratégia, a dependência não efetiva entre `A` e `B` pode ser removida ao desativar o método `B.m3()`, porque ele invoca `A.m4()`. Porém, não é possível executar `B.m2()` necessário para testar `A.m1()` se `B.m3()` não for implementado. Existe ainda o fato de que a estratégia desconsidera a utilização de referências diretas a atributos públicos das classes, ou seja, dependências que são originárias de atributos e não de métodos.

Jaroenpiboonkit e Suwannasart (2007) propõem uma estratégia que se assemelha à proposta de Badri *et al.* (2005). A estratégia usa o conceito de fatiamento de classes (*class slicing*), uma maneira de desativar partes das classes e efetuar a integração parcial. Pode-se considerar que a proposta de Jaroenpiboonkit e Suwannasart (2007) corrige um dos pontos fracos da proposta de Badri *et al.* (2005), porque leva em consideração a utilização de métodos e atributos dentro de uma mesma classe para efetuar o fatiamento. A remoção das dependências baseia-se na técnica de fatiar uma classe em partes especificamente necessárias ao teste de outras classes. Para isso a estratégia de Jaroenpiboonkit e Suwannasart (2007) precisa de um diagrama com informações sobre as invocações de métodos pelas classes. Por esse motivo é usado o TDG (*Test Dependency Graph*) proposto por Le Traon *et al.* (2000). O problema da referência direta a atributos não é tratada, assim como a proposta de Badri *et al.* (2005). Os autores utilizam um exemplo hipotético usado por Briand *et al.* (2003) para declarar que não são necessários *stubs* quando a técnica é utilizada. O ponto fraco do trabalho é que não são apresentados estudos para mostrar a eficiência da estratégia em número de *stubs*.

Um ponto importante a ser discutido nas propostas de Badri *et al.* (2005) e Jaroenpiboonkit e Suwannasart (2007) é que quando uma classe é fatiada, ou parcialmente desativada segundo Badri *et al.* (2005), existe a falsa impressão de que é possível implementá-la parcialmente e testá-la, ou que é possível usar uma outra classe parcialmente implementada já testada, o que na verdade é um *stub* realístico.

Malloy *et al.* (2003) apresentam uma estratégia que usa um ORD estendido para representar as dependências entre classes que, ao contrário de Badri *et al.* (2005) – que diminui os tipos de arestas em relação ao ORD original –, identificam mais tipos diferentes de dependências: associação, composição, dependência, herança, *ownedElement* e polimorfismo. Os cinco primeiros tipos são os mesmos utilizados pela notação da UML e a dependência caracterizada pelo polimorfismo é a mesma utilizada por Labiche *et al.* (2000) e mostrada na Figura 2.3. Malloy *et al.* (2003) argumentam que em virtude de classes *Template*, um recurso poderoso da linguagem C++, ciclos podem ser formados sem a presença do relacionamento de associação. Como resultado dessas alterações o ORD se torna um multigrafo, que pode possuir mais de uma aresta partindo de um nó v qualquer para um nó w qualquer. Para remover arestas, um modelo de custo de ordenação é proposto. O modelo possui as seguintes características:

- diferentes valores são previamente associados às arestas do ORD, representando o peso relativo de cada tipo de aresta;
- durante o processo de ordenação, arestas que partem de uma mesma origem e chegam a um mesmo destino são juntadas e o peso da aresta resultante é calculado com base na soma dos pesos das arestas originais.

O processo de ordenação é composto pelos seguintes passos:

1 O ORD estendido G é produzido com base em engenharia reversa;

  1.1 Os pesos são associados às arestas do ORD;
  1.2 As arestas são juntadas e os pesos recalculados;

2 G é particionado de acordo com seus SCCs;

3 O modelo de custo é usado para quebrar os SCCs da seguinte maneira:

  3.1 Um SCC c não trivial é escolhida;
  3.2 A aresta com menor peso é removida;
  3.3 O passo 2 deve ser repetido para encontrar SCCs em c;
  3.4 O passo 3 deve ser repetido até não existirem mais ciclos;

4 A ordem de teste é produzida pelos seguintes passos:

4.1 Cada aresta a removida durante o passo 3 deve ser removida de G, resultando em G';

4.2 As classes em G' são ordenadas por uma ordenação topológica reversa.

O trabalho de Malloy *et al.* (2003) apresenta um estudo feito com 6 sistemas implementados na linguagem C++ que mostra seu desempenho em número de *stubs* considerando a existência de ciclos entre classes relacionadas apenas com herança e associação. No entanto, não existe no estudo comparação com outras estratégias propostas. Além disso, na proposta de Malloy *et al.* (2003) relacionamentos de herança podem ser removidos do ORD estendido.

Mao e Lu (2005) propõem um algoritmo chamado AICTO (*An Improved Algorithm for Planning Inter-class Test Order*) para o ordenação de classes no teste de integração. O algoritmo usa uma extensão do ORD chamado WORD (*Weighted Object Relation Diagram*). Nessa extensão arestas de associação possuem três pesos: CW (*Cycling Weight*), que representa o peso da aresta na quebra de ciclos; DF (*Direction Factor*), que representa o peso da aresta em relação a outras arestas de herança e agregação; e RD, que representa a intensidade do relacionamento da associação. Nesse algoritmo, apenas arestas de associação podem ser removidas para quebrar ciclos. O CW de uma aresta é calculado com base no número de arestas de entrada do vértice de origem e do número de arestas de saída do vértice destino, assim como Briand *et al.* (2003). O cálculo é ligeiramente diferente porque leva em consideração dependências bidirecionais entre duas classes, isto é, quando uma classe `A` depende de uma classe `B`, e a classe `B` depende da classe `A`. O DF de uma aresta é calculado por meio de uma busca em profundidade feita no ORD considerando apenas arestas de herança e agregação. Os autores não dão detalhes de como calcular o DF. O RD de uma aresta é calculado com base no número de usos de atributos públicos, chamadas de métodos e instâncias de uma classe usada por outra classe.

O algoritmo usa os seguintes passos para remover as dependências: :

1 deve-se remover a aresta que tem o maior CW;

2 caso existam arestas com CW iguais, deve-se remover aquela que tem o maior DF;

3 caso existam arestas com CW e DF iguais, deve-se remover aquela que possui o menor RD;

4 caso existam arestas com CW, DF e RD iguais, deve-se remover aleatoriamente qualquer uma delas.

A ordenação é feita no WORD que não tem ciclos por meio de uma ordenação topológica reversa. Segundo os autores, o algoritmo produz o mesmo número de *stubs* que a estratégia

de Briand *et al.* (2003), porém com menos iterações. Além disso, eles argumentam que os *stubs* necessários são menos complexos em virtude da heurística usada pelos pesos CW, DF e RD. O primeiro ponto fraco do trabalho é que não existem estudos para mostrar a eficiência do algoritmo em número de *stubs*, os resultados são obtidos por meio de um exemplo hipotético contido no trabalho de Briand *et al.* (2003). O segundo ponto é que o cálculo do peso DF não é mostrado com exatidão. O terceiro, e último ponto, é que o peso RD usa como parte do cálculo o número de instâncias que uma classe usa em um relacionamento de associação. Note-se que essa é uma informação que pode não ser obtida em certas implementações de classes, o que torna impossível o cálculo desse peso. Os autores não propõem uma solução que contorne esse problema.

Abdurazik e Offutt (2006) propõem uma estratégia que usa alguns conceitos de Malloy *et al.* (2003) e Mao e Lu (2005). Os autores utilizam vários tipos de dependências como Malloy *et al.* (2003), mas diferentemente deles não removem arestas de herança e agregação. No trabalho, são usados diferentes pesos associados a arestas como Mao e Lu (2005), e também como Malloy *et al.* (2003). A idéia da proposta de Abdurazik e Offutt (2006) é que o peso de uma aresta deve representar o custo de implementação de um *stub* caso a aresta seja removida. Por isso a estratégia é baseada no acoplamento entre classes (*coupling-based*). Os autores argumentam que os vértices também possuem pesos, mas no trabalho isso não é mostrado. São considerados 10 tipos de relacionamentos entre classes para o cálculo do acoplamento que uma aresta denota no WORD (*Weighted Object Relation Diagram*) por eles proposto. O cálculo é feito com base no tipo de relacionamento em função do número de atributos referenciados, de métodos invocados, de parâmetros de métodos e de valores retornados simulados pelo *stub* caso a aresta seja removida. A remoção de uma aresta é feita com base no cálculo do peso que representa o acoplamento entre os dois vértices ligados pela aresta. As remoções e os cálculos dos pesos das arestas são feitos recursivamente até que não existam mais ciclos no WORD. Com o WORD sem ciclos, a ordem de teste de integração é produzida por meio de uma ordenação topológica reversa. O trabalho apresenta um estudo que usa o mesmo sistema dos trabalhos de Briand *et al.* (2002, 2003). O estudo mostra que a estratégia proposta por Abdurazik e Offutt (2006) produz o mesmo número de *stubs* de Briand *et al.* (2003). Contudo, segundo os autores os *stubs* são menos complexos.

Na Tabela 2.5 é apresentado um resumo com as principais características de algumas estratégias aqui descritas. As estratégias de Briand *et al.* (2002) e Le Hanh *et al.* (2001) não estão listadas porque não são baseadas em grafos. A primeira coluna da tabela indica a estratégia, a segunda mostra qual é o modelo usado como dígrafo para que a estratégia produza a ordem de integração e a terceira mostra se a estratégia utiliza *stubs* específicos e realísticos. A quarta coluna diz se a estratégia considera ou não polimorfismo e ligação dinâmica e a quinta se a

estratégia utiliza o conceito de integração parcial – desativação de métodos ou fatiamento. A sexta coluna mostra a quantidade de estudos em que a estratégia foi aplicada em sistemas reais e a última coluna mostra qual foi o artefato de software utilizado na construção do modelo usado pela estratégia, mostrado na segunda coluna. Apesar de estudos serem conduzidos em algumas estratégias, em nenhuma delas a ordenação produzida foi usada efetivamente, uma vez que o teste não foi conduzido. O número de *stubs* computados por cada uma é calculado de forma automatizada por meio dos modelos que cada uma usa em seu algoritmo de ordenação.

Tabela 2.5: Resumo com as principais características das estratégias de ordenação revisadas.

Proposta	**Modelo de dependências**	**Tipo dos *stubs***	**Considera polimorfismo?**	**Usa integração parcial?**	**Número de estudos**	**Origem das informações nos estudos**
Kung *et al.*	ORD	Específico	Não	Não	1	Código fonte
Tai e Daniels	ORD	Realístico	Não	Não	0	–
Le Traon *et al.*	TDG	Realístico	Não	Não	1	Diagrama de classes
Labiche *et al.*	ORD	Específico	Sim	Não	1	Diagrama de classes
Briand *et al.*	ORD	Específico	Não	Não	5	Código fonte
Malloy *et al.*	WORD*	Específico	Sim	Não	5	Código fonte
Lima e Travassos	Diagrama de Classes	Realístico	Não	Não	0	Diagrama de classes
Badri *et al.*	CDM	Realístico	Não	Sim	4	Código fonte
Mao e Lu	WORD*	Específico	Não	Não	0	–
Abdurazik e Offutt	WORD*	Específico	Não	Não	1	Código fonte
Jaroenpiboonkit e Suwannasart	TDG	Realístico	Não	Sim	0	–

* Diferentes estratégias estendem o ORD e chamam o novo modelo de WORD, porém o cálculo dos pesos das arestas desses WORDs diferem de estratégia para estratégia.

As estratégias que não foram aplicadas a sistemas reais não possuem indicação do artefato de software utilizado para a construção do modelo usado em seu algoritmo. A estratégia de Lima e Travassos (2004) usa diretamente o diagrama de classes da UML para produzir a ordem de teste.

Analisando os dados da Tabela 2.5, pode-se concluir que:

- a maioria dos trabalhos considera o uso de *stubs* específicos como a melhor escolha;
- a maioria dos trabalhos usa o ORD ou extensões do ORD em que pesos são associados às arestas;
- poucos trabalhos consideram polimorfismo e usam o conceito de integração parcial;
- alguns trabalhos não apresentam estudos com sistemas reais que validam suas estratégias, o que enfraquece essas propostas;
- apenas uma proposta utiliza o próprio diagrama de classes como modelo de dependências;

- apenas o trabalho de Le Traon *et al.* (2000) apresenta um regras de mapeamento entre modelos de projeto e um modelo usado na produção da ordem de teste – de diagramas UML para o TDG.

Além dessas conclusões, uma análise mais detalhada das estratégias e suas diferentes técnicas para a ordenação de classes no teste de integração mostra pontos bastante importantes: a estratégia de Briand *et al.* (2003) produz os melhores resultados em número de *stubs* específicos e por isso é usada como base ou como comparação pelas demais estratégias; nenhum dos trabalhos apresenta estudo exploratório em que sistemas tenham sido realmente testados usando sua ordenação produzida e os *stubs* implementados tenham sido coletados, contabilizados e analisados; e nenhum trabalho mostra o processo de mapeamento entre um modelo de projeto e o ORD similarmente à proposta de Le Traon *et al.* (2000), que define regras de mapeamento entre diagrama de classes e o TDG.

2.6 Ferramentas de teste

O teste de software é uma tarefa que necessita de apoio ferramental para que seja uma tarefa menos custosa e propensa a erros (Delamaro *et al.*, 2007). Assim, para mostrar um panorama das ferramentas de teste existentes nessa, seção são listadas algumas das principais ferramentas de teste.

A qualidade e a produtividade da atividade de teste dependem do critério de teste utilizado e da existência de uma ferramenta que o apoie. Sem a existência de uma ferramenta, a aplicação de um critério torna-se uma atividade propensa a erros e, de certa forma, limitada (Delamaro *et al.*, 2007). Diversos trabalhos apresentam ferramentas de teste para apoiar a aplicação de critérios de teste. Por exemplo, como ferramentas de apoio à aplicação dos critérios baseado em análise de fluxo de controle e de fluxo de dados em programas procedimentais pode-se citar as ferramentas *PokeTool* (Chaim, 1991; Maldonado *et al.*, 1989), *Asset* (Frankl e Weyuker, 1985), *Atac* (Horgan e Mathur, 1992), *ATACOBOL* (Sze, 2000) e *xSuds* (Agrawal *et al.*, 1998; Telcordia Technologies, Inc., 1998), sendo que esta última também pode ser utilizada no teste de programas em C++. No que se refere ao teste baseado em erros, as principais ferramentas desenvolvidas são a *Mothra* (Demillo, 1980; DeMillo *et al.*, 1988) para Fortran, a *Proteum* (Delamaro, 1993) e a *Proteum/IM* (Delamaro, 1997) para C/C++. Estas duas últimas foram integradas em um único ambiente de teste que apoia a realização de testes de unidade e de integração em programas procedimentais chamado *Proteum/IM 2.0*. Detalhes das ferramentas desenvolvidas no ICMC/USP podem ser obtidos no trabalho de Maldonado *et al.* (1998).

Domingues (2005) avalia um conjunto de ferramentas de teste de diferentes linguagens para verificar quais critérios de teste são utilizados. Vincenzi (2004) apresenta as características de algumas ferramentas OO disponíveis e constata que a maioria das ferramentas não oferece suporte ao uso de critérios baseados em fluxo de dados. Algumas ferramentas contidas nesse trabalho são: um conjunto de ferramentas da Parasoft Parasoft Corporation (2005) para o teste de programas C++, Java e aplicações Web; a *Panorama C/C++* (International Software Automation, 2005a) e a *Panorama for Java* (International Software Automation, 2005b) que, na verdade, são um conjunto de cinco ferramentas, a saber, *OO-Test*, *OO-SQA*, *OO-Analyser*, *OO-Browser* e *OO-Diagrammer*; a *TCAT C/C++* (Software Research, Inc., 2001a) e a *TCAT Java* (Software Research, Inc., 2001b), que são ferramentas que fornecem medidas para determinar o quanto de código já foi testado; a *xSuds Toolsuite* (Agrawal *et al.*, 1998; Telcordia Technologies, Inc., 2005) que, na verdade, é um conjunto de sete ferramentas para o entendimento, a análise e o teste de software para códigos C/C++, a saber, *xATAC*, *xRegress* e *xVue*; a *JProbe Suite* (Quest Software, 2005), que é um conjunto de três ferramentas que ajudam a eliminar gargalos de execução causados por algoritmos ineficientes em códigos Java e aponta as causas de perdas de memória nessas aplicações, a saber, *JProbe Profiler and Memory Debugger*, *JProbe Threadalyzer* e *JProbe Coverage*; o JUnit (JUnit.org, 2009), que é um *framework* de teste muito utilizado em metodologias ágeis de desenvolvimento que viabiliza a documentação e execução automática de casos de teste; e a *JaBUTi* (*Java Bytecode Understanding and TestIng*) (Vincenzi, 2004), que é uma ferramenta de teste desenvolvida no ICMC/USP para teste de programas Java. É importante salientar que a *JaBUTi* possui várias extensões, dentre elas para o teste de programas OA.

No contexto de ordenação de classes, quase todas propostas apresentadas na Seção 2.5 são apoiadas por ferramentas. No entanto, não existem mais detalhes sobre como elas funcionam além da descrição dada em cada trabalho.

2.7 Considerações finais

Conforme pôde ser constatado, o problema da ordenação de classes durante o teste de integração de programas OO é um problema importante abordado em diversos trabalhos. O uso de uma estratégia adequada pode diminuir o custo da atividade de teste por meio da minimização do número de *stubs* implementados durante o teste de integração, da possibilidade de rastreamento e eliminação de defeitos de maneira sistemática, e do gerenciamento da integração das unidades. Dentre os trabalhos apresentados, a proposta de Briand *et al.* (2003) se destaca por obter melhor resultado que as demais, considerando o número de *stubs* específicos. Além disso, ela apresenta

características interessantes: possui um modelo de representação de dependências que pode ser mapeado a partir de documentos de projeto, o ORD; segue uma conjectura amplamente aceita de que relacionamentos de herança e agregação representam maior acoplamento entre classes do que outros tipos de relacionamento; e seu algoritmo de ordenação possui um bom desempenho, considerando que a ordenação de classes é um problema NP-completo.

Um ponto a ser observado é que as fases de teste são diferenciadas no teste de programas OO porque, ao contrário do teste procedimental, existe discordância na determinação da menor unidade do software. Outra característica importante, é que as interações entre os objetos são muito diferentes das interações que ocorrem no paradigma procedimental (Binder, 1999): objetos possuem estado, objetos podem se relacionar por herança, agregação e associação e objetos possuem muitos métodos com pouca implementação que interagem fortemente com outros objetos em busca de um objetivo comum, dentre outras características. Nesse contexto, o teste de integração ganha importância em relação ao paradigma procedimental em virtude da complexidade dos relacionamentos entre suas entidades, com isso, a necessidade de propostas que abordam o uso de estratégias de ordenação se torna evidente.

CAPÍTULO

3

Desenvolvimento de software baseado em separação de interesses

3.1 Considerações iniciais

As propostas de desenvolvimento baseado em separação de interesses são bastante recentes e apresentam novos desafios na área de teste de software. A programação OA, como uma vertente do desenvolvimento baseado em interesses, possui características semelhantes à programação OO mas também novas características que devem ser estudadas para que as abordagens de teste específicas para a programação OA possam ser introduzidas.

Além dos conceitos básicos que descrevem as principais características da programação OA, um ponto importante que deve ser considerado são as abordagens de desenvolvimento baseado em separação de interesses. Ao contrário da programação OO, em que já existem métodos de desenvolvimento consolidados (Jacobson *et al.*, 1999; Larman, 2002), com processos e modelos que descrevem artefatos de software bem definidos, na programação OA esse ainda é um tópico em evolução (Clarke e Baniassad, 2005; Schauerhuber *et al.*, 2007). É necessário entender os processos e modelos propostos para a programação OA para uma melhor compreensão dos problemas que envolvem o teste de programas OA.

De forma semelhante ao ocorrido com o advento da programação OO, novas abordagens, técnicas e critérios de teste específicos para a programação OA têm surgido. Esse é um outro ponto importante a ser analisado, uma vez que ajuda a evidenciar os rumos das pesquisas em teste de software no contexto de programação OA.

Na Seção 3.2 são apresentados os conceitos básicos que descrevem as principais características do desenvolvimento baseado em interesses e, assim, prover a fundamentação teórica necessária à compreensão do restante do trabalho. Na Seção 3.3 é apresentada uma descrição

mais detalhada da programação OA porque essa é a abordagem utilizada nos próximos capítulos. Na Seção 3.4 são apresentadas duas diferentes propostas de modelagem baseada em interesses, uma vez que, ao contrário da programação OO, esse é um assunto ainda não consolidado na programação baseada em interesses. Uma breve introdução ao teste OA é feita na Seção 3.5, com destaque para as propostas de novas técnicas e critérios de teste e para a reformulação do conceito de fases de teste de software. Uma coletânea de ferramentas de teste de software que apoiam a programação OA é mostrada na Seção 3.6. Por fim, na Seção 3.7, são apresentadas as considerações finais deste capítulo.

3.2 Conceitos básicos

Modularizar o software para tornar gerenciáveis os problemas relacionados ao seu desenvolvimento é uma estratégia clássica da engenharia de software. A modularização faz com que a complexidade do software seja menor e, ao mesmo tempo, torna-o mais compreensível. A capacidade de modularização é diretamente ligada à capacidade de manter os interesses relacionados à especificação do software separados (Tarr *et al.*, 1999).

A separação de interesses é o princípio básico do desenvolvimento baseado em interesses. Embora não exista uma definição amplamente aceita do termo, o princípio da separação de interesses diz respeito à habilidade de identificar, encapsular e manipular partes do software que são relevantes para um conceito, meta, propósito ou requisito em particular (Kienzle *et al.*, 2003). Muitas abordagens de desenvolvimento identificam e dedicam esforços com o intuito de projetar e implementar o software em relação a um interesse "mais importante". Por exemplo, em linguagens OO, a atenção no desenvolvimento é dada às classes e às suas instâncias, que devem encapsular as funcionalidade do sistema. Isso faz com que outros interesses, como por exemplo, persistência e distribuição, fiquem fracamente encapsulados em um segundo plano, ou seja, o software é decomposto segundo um interesse dominante. Geralmente, os interesses relegados ao segundo plano – ou interesses transversais (*crosscutting concerns*) – são não funcionais, embora existam abordagens que declaram que mesmo entre as diferentes funcionalidades de um software pode haver a separação de interesses (Baniassad e Clarke, 2004), e outras que consideram funcionais todos os interesses relacionados ao software (Kienzle *et al.*, 2003). Como resultado da decomposição dominante, a implementação referente aos interesses transversais fica, sob o ponto de vista de um interesse transversal, espalhada por vários pontos do código fonte do interesse principal, e, sob o pronto de vista do interesse principal, as implementações dos interesses transversais ficam emaranhadas em meio ao código fonte do interesse principal.

Várias propostas têm sido feitas com a intenção de fornecer recursos que apoiam a separação de interesses no desenvolvimento de software, como por exemplo, a programação OA (Kiczales *et al.*, 1997), a programação adaptativa (Lieberherr, 1995), os filtros de composição (Aksit *et al.*, 1992), a programação orientada a assuntos (Harrison e Ossher, 1993; Ossher *et al.*, 1995) e sua sucessora, a separação multidimensional de interesses (Tarr *et al.*, 1999), e a abordagem *Theme* (Baniassad e Clarke, 2004).

3.3 Programação orientada a aspectos

Nesta seção é discutida apenas a programação OA porque ela vem se destacando em relação às outras abordagens. A programação OA é uma proposta que tem o objetivo de apoiar o desenvolvimento de software baseado em interesses por meio de mecanismos para a construção de programas em que os interesses transversais ficam separados dos interesses-base, em vez de espalhados pelo sistema.

A diferença principal apresentada pela programação OA com relação às outras técnicas de programação é a de possibilitar a implementação de módulos[1] isolados – os aspectos – que têm a capacidade de afetar vários outros módulos do sistema de forma transversal. Na programação OA, um único aspecto pode contribuir para a implementação de diversos outros módulos que implementam as funcionalidades básicas (chamado de código-base). Esse mecanismo que permite um aspecto afetar vários pontos do sistema é chamado de quantificação. Apesar de trazer vantagens para a localização dos interesses e coesão dos módulos, o poder do mecanismo de quantificação oferecido pela programação OA faz com que dependências sejam criadas entre vários módulos diferentes, além de poder levar a novos tipos de defeitos.

Aspectos podem adicionar comportamento em outros módulos por meio do mecanismo de quantificação e, para isso, é necessário que se possa identificar pontos específicos da execução do programa, chamados de pontos de junção (*join points*), para que o comportamento adicional possa ser definido. Dessa forma, uma linguagem OA deve fornecer um modelo por meio do qual possam ser identificados esses pontos na execução do programa (chamado modelo de pontos de junção).

Um conjunto de pontos de junção – ou somente conjunto de junção (*pointcut*) – identifica diversos pontos de junção em um sistema. O conjunto de junção é utilizado pelo comportamento transversal, que é implementado por uma construção similar a um método, chamada de adendo (*advice*), que executa antes, depois ou no lugar dos pontos de junção identificados. Esses adendos são chamados de adendos anteriores, posteriores e de contorno. Por exemplo, para

[1] O termo módulo é usado para designar classes e aspectos indistintamente.

implementar um interesse de registro que imprime uma mensagem toda vez que um método é chamado, poder-se-ia utilizar um aspecto com um adendo anterior que executaria toda vez que qualquer método do sistema fosse chamado.

Depois do código-base e dos aspectos serem codificados, é necessário combinar (*weave*) os módulos em uma aplicação executável. Assim, toda linguagem de programação OA deve contar também com um combinador (*weaver*), responsável por essa tarefa. A combinação pode ser feita em diversos momentos, dependendo da decisão de implementação tomada para cada linguagem de programação específica: combinação dinâmica é o processo de combinação que ocorre em tempo de execução, enquanto que a combinação estática ocorre em tempo de compilação.

De maneira geral, os aspectos podem servir para dois propósitos: para auxiliar atividades de desenvolvimento (aspectos de "desenvolvimento"), ou para implementar interesses que estarão na aplicação final (aspectos de "produção"). No primeiro caso, os aspectos são utilizados para ajudar em atividades como depuração, garantia de que padrões de código estejam sendo utilizadas e teste de software. Nesses casos, os aspectos não estarão presentes na aplicação final, mas serão apenas utilizados durante o desenvolvimento, sendo retirados antes da entrega do produto final. No segundo caso os aspectos cumprem papéis relacionados com a funcionalidade do software e estarão presentes na aplicação final, como por exemplo, aspectos que implementam regras de negócio.

3.3.1 A linguagem AspectJ

Existem várias abordagens linguísticas para a implementação de sistemas OA, como por exemplo: AspectC++ (AspectC.org, 2009) para a linguagem C++, AspectC# (Kim, 2002) para a linguagem C#, AspectS (de Alwis e Kiczales, 2009) para a linguagem Smalltalk e AspectJ (The AspectJ Team, 2002) para a linguagem Java. Como AspectJ é a linguagem mais difundida e a que foi utilizada neste trabalho, será a única descrita com detalhes.

A linguagem de programação OA mais estável no presente momento é a linguagem AspectJ (The AspectJ Team, 2002), uma extensão de Java para apoiar a programação OA. Por ser uma extensão de Java, AspectJ contém todas as construções dessa linguagem, e portanto é construída sobre o paradigma OO.

Além das classes Java comuns, em AspectJ pode-se codificar aspectos por meio da palavra reservada `aspect`. Aspectos são módulos que combinam: especificações de pontos de junção utilizando conjuntos de junção (`pointcut`); adendos anteriores (`before`), posteriores (`after`) e de contorno (`around`); e declarações intertipos, utilizadas para introduzir atributos, métodos e heranças em outras classes ou interfaces. Versões recentes de AspectJ também

apoiam declarações de avisos e erros de compilação quando certos pontos de junção são identificados (The AspectJ Team, 2002).

O modelo de pontos de junção de AspectJ é baseado nas construções sintáticas da linguagem: chamada e execução de métodos/construtores (`call` e `execution`), leitura e escrita de atributos (`get` e `set`), execução de um determinado tratador de exceção (`handler`), e outros. Um conjunto de junção é definido a partir de designadores de conjuntos de junção, que são os `call`, `execution`, `get`, etc, que podem ser compostos utilizando operadores "e" (`&&`) e "ou" (||) para formar um único conjunto de junção. Além disso, pode ser utilizado o operador de negação (`!`) para restringir a captura de determinados pontos de junção. Por questões de simplificação, usar-se-á o termo conjunto de junção para se referir tanto ao próprio conjunto quanto aos designadores.

Existem alguns tipos de conjuntos de junção que dependem de informações dinâmicas para identificar os pontos de junção. Por exemplo, podem ser definidos conjuntos de junção compostos que identifiquem chamadas a métodos que estejam em um fluxo de controle determinado, por meio da composição de `call` com os `cflow` e `cflowbellow`. Para mais informações sobre os outros tipos de conjuntos de junção, podem ser consultadas referências que abordam a linguagem completamente, tais como Laddad (2003) e The AspectJ Team (2002).

Em AspectJ, existem três tipos de adendos posteriores: `after returning`, `after throwing` e simplesmente `after`. O primeiro é executado apenas quando o ponto de junção executa normalmente e retorna um valor de um tipo determinado ou qualquer, o segundo apenas quando uma exceção é lançada e o terceiro executa sempre que o ponto de junção é capturado.

O adendo de contorno pode fazer com que o ponto de junção alcançado prossiga com a sua execução por meio do método `proceed()`, que também pode ser utilizado para obter o eventual valor de retorno do ponto de junção. Dentro dos adendos também é possível capturar dados do contexto dos pontos de junção. Por exemplo, se o ponto de junção é uma chamada a método, os parâmetros que foram passados na chamada podem ser utilizados dentro dos adendos. Isso pode ser feito de algumas formas: utilizando os conjuntos de junção do tipo `args`, para obter os argumentos; utilizando os conjuntos de junção do tipo `this`, para obter o objeto executando no ponto de junção; utilizando os conjuntos de junção do tipo `target`, para obter o objeto alvo em um ponto de junção, por exemplo, o objeto para o qual está sendo feita uma chamada a método; ou por meio da palavra reservada `thisJoinPoint`, que funciona como um objeto que encapsula o ponto de junção alcançado.

Em AspectJ, é possível alterar a estrutura estática dos tipos da linguagem Java – classes e interfaces – e até mesmo outros aspectos. Isso pode ser feito por meio do uso do mecanismo chamado declaração intertipos. Existem diversos tipos de entrecorte estático com declarações

intertipos: introdução de elementos, alteração de hierarquia de herança, relaxamento de exceções, declaração de erros e avisos em tempo de compilação. A introdução de elementos permite, por exemplo, que métodos e atributos sejam adicionados a uma classe entrecortada, também chamada de alvo da declaração intertipo, alterando assim sua estrutura original.

3.4 Modelagem baseada em separação de interesses

Schauerhuber *et al.* (2007) apresentam um estudo que compara de maneira detalhada 8 diferentes propostas de modelagem baseada em interesses. As propostas escolhidas são as mais importantes segundo os autores, porém eles citam que foram encontradas várias outras. Essa variedade de propostas deixa evidente a falta de maturidade da modelagem OA e que este ainda é um campo aberto para futuras pesquisas. Outro ponto importante é que a comunidade científica e profissional ainda não adotou claramente uma proposta como padrão. Dessa forma, nesta seção é discutida a modelagem baseada em separação de interesses porque não existe uma proposta sedimentada e plenamente aceita, e escolheu-se não descrever a modelagem OO porque esse é um assunto bastante conhecido.

O estudo de Schauerhuber *et al.* (2007) apresenta, com base em um arcabouço (*framework*) de comparação entre as diferentes propostas, duas conclusões importantes: a maioria das propostas utilizam a UML (OMG, 2007) de alguma forma para especificar classes-base[2] e aspectos; e a abordagem *Theme*, proposta por Clarke e Baniassad (2005), é a que possui maior maturidade, sendo a mais citada na literatura especializada, a que apresenta o maior número de exemplos de aplicação e a única, juntamente com a proposta de Jacobson e Ng (2004), que possui um processo que apoia o desenvolvimento OA. Em contrapartida, essa abordagem não é apoiada integralmente por ferramentas de modelagem e geração de código. Por esses motivos, a falta de uma abordagem padrão e a condição de maturidade da proposta de Cibrán *et al.* (2003) sem apoio ferramental, são descritas nas próximas duas seções abordagens para a modelagem OA.

A primeira abordagem é a MATA (Whittle e Jayaraman, 2007), que é bastante recente. As vantagens da utilização da abordagem MATA é que ela preserva a notação da UML, com pequenas extensões e o uso de estereótipos, e possui uma ferramenta de geração automática dos aspectos em AspectJ. Posteriormente, é apresentada a abordagem *Theme* (Clarke e Baniassad, 2005). Ao contrário de MATA, a abordagem *Theme* não é apenas uma notação, é um método de desenvolvimento de software baseado em interesses que usa uma notação própria, baseada na UML, e compreende as fases de análise, projeto e composição de aspectos.

[2]Considera-se que classes-base é o termo correspondente em AspectJ para o termo código-base da programação OA.

3.4.1 Abordagem MATA

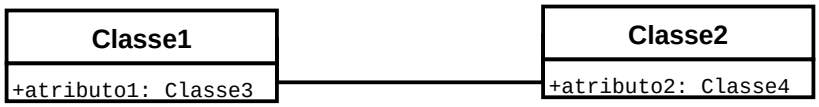
MATA é uma ferramenta de modelagem que trata a combinação de aspectos por meio de modelos de transformação. Ela usa uma notação de regras de transformação de grafos que tem o mesmo nome. A combinação dos modelos-base e dos modelos de aspectos são especificadas como regras de transformação de grafos. Assim, dado um modelo-base M_B entrecortado por um modelo de aspectos M_A, uma regra de transformação MATA combina M_B e M_A produzindo um modelo composto M_{AB}. As regras de transformação possuem dois lados, o esquerdo (LHS, *Left Hand Side*) e o direito (RHS, *Right Hand Side*). A regra de transformação $r : LHS \rightarrow RHS$ define, por meio de seu LHS, um padrão de captura de pontos de junção em M_B. O RHS da regra de transformação especifica e define como novos elementos são adicionados ou retirados de M_B.

A ferramenta MATA apoia a especificação de regras de transformação usando UML, mais especificamente, os diagramas de classe, de sequência e de estados. Ao contrário de outras abordagens as regras de transformação de MATA são definidas a partir da sintaxe concreta da linguagem de modelagem. Outras abordagens propõem transformações a partir da sintaxe abstrata. É importante ressaltar essa característica da ferramenta pois o engenheiro de software não precisa conhecer o metamodelo UML para modelar um sistema OA.

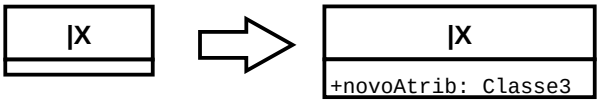
Um exemplo hipotético de aplicação de regras de transformação é apresentado na Figura 3.1. Na Figura 3.1(a) é apresentado um diagrama de classes com duas classes relacionadas, cada uma com um atributo. Na Figura 3.1(b) há uma regra de transformação que denota alteração estrutural. Na regra, um atributo chamado `novoAtrib` é introduzido em qualquer classe. Note-se que isso é descrito pela notação "|X", que ao invés de conter o nome da classe alvo da introdução, isto é, onde a alteração estrutural deve ser feita, contém uma variável. No LHS da regra não há atributos e no RHS existe um atributo, descrevendo que o atributo `novoAtrib` deve ser adicionado às classes em que a regra é aplicada. A regra pode indicar também quais elementos devem ser retirados de uma classe. Elementos que estão no LHS da regra e não estão no RHS devem ser removidos. Na Figura 3.1(c) é mostrado o resultado da aplicação da regra. Nessa figura pode-se notar que as duas classes, que antes da aplicação da regra tinham apenas um atributo, possuem o novo atributo introduzido pela aplicação da regra.

A notação MATA permite que o comportamento dos modelos-base sejam alterados. Isso é feito por meio de regras de transformação de alteração de comportamento descritas com diagramas de sequências. Para isso a ferramenta define três novos estereótipos UML:

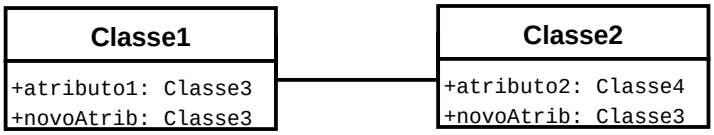
- «`create`»: que pode ser aplicado a qualquer elemento do modelo contido na regra, e declara que um elemento deve ser criado pela regra de transformação;



(a) Um exemplo de diagrama de classes.



(b) Uma regra de transformação de alteração estrutural que introduz um novo atributo.



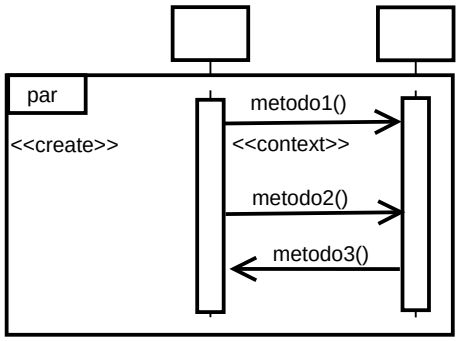
(c) O diagrama de classes transformado pela aplicação da regra.

Figura 3.1: Exemplo de aplicação de uma regra de transformação MATA de alteração estrutural (Whittle e Jayaraman, 2007).

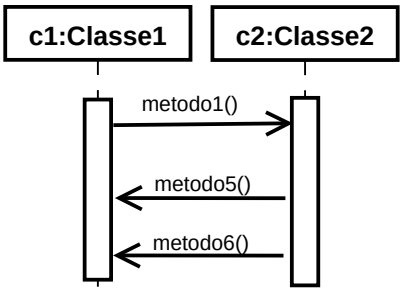
- «`delete`»: que pode ser aplicado a qualquer elemento do modelo contido na regra, e declara que um elemento deve ser removido pela regra de transformação;
- «`context`»: usado como um contêiner de elementos da regra para evitar que elementos sejam afetados pelos estereótipos «`create`» e «`delete`».

Na Figura 3.2 é exemplificado como uma regra de alteração de comportamento captura um ponto de junção. A regra de transformação de alteração de comportamento é mostrada na Figura 3.2(a). Note-se que a regra não especifica em quais classes o ponto de junção é capturado, uma vez que não existe nome das classes no diagrama de sequência. É o mesmo que especificar nas duas classes da regra "|x:|X". A regra descreve que há comportamento adicionado no ponto de junção capturado, como pode ser observado no fragmento `par`, anotado com o estereótipo «`create`». No mesmo fragmento é usado o estereótipo «`context`» ligado ao método `metodo1()`, indicando que ele representa o contexto de aplicação e não é alterado pela aplicação da regra. Além do método `metodo1()`, existe no fragmento mais dois métodos, `metodo2()` e `metodo3()`, que representam o comportamento adicionado, uma vez que o estereótipo «`create`» é utilizado.

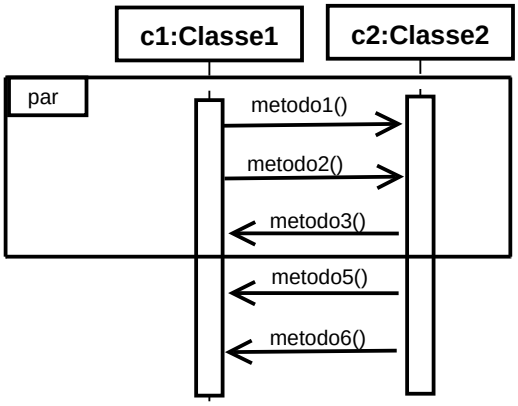
A regra apresentada na Figura 3.2(a) descreve que o ponto de junção capturado é uma chamada ao método `metodo1()`, feita de qualquer classe para qualquer classe e que, logo após a execução desse método, deve-se adicionar o novo comportamento, que são as chamadas ao método `metodo2()` e ao método `metodo3()`. Na Figura 3.2(b) é mostrado um ponto de junção capturado pela regra e na Figura 3.2(c) é mostrada a adição do comportamento feita no ponto de junção capturado.



(a) Uma regra de transformação de alteração de comportamento.



(b) Um diagrama de sequência que representa o ponto de junção capturado pela regra.



(c) Diagrama de sequência da aplicação da regra no ponto de junção capturado.

Figura 3.2: Exemplo de aplicação de uma regra de transformação de alteração de comportamento.

A ferramenta funciona acoplada à ferramenta RSM *IBM Rational Software Modeler* (IBM, 2009). Cada aspecto é modelado como um pacote. O pacote contém as regras de transformação de alteração estrutural e de alteração de comportamento. Assim, o engenheiro de software pode escolher quais aspectos devem ser combinados e a ordem de combinação. Como a ferramenta MATA usa transformações em grafos ela utiliza uma ferramenta de execução de regras de grafos chamada AGG (Taentzer, 2003). A ferramenta AGG é usada para validar os modelos e detectar possíveis conflitos entre as regras de transformação e dependências de elementos alterados. MATA possui uma notação especial para especificar alteração de comportamento em diagramas de estados. No entanto, essa capacidade não é descrita aqui. Detalhes podem ser estudados em Whittle e Jayaraman (2007).

3.4.2 Abordagem *Theme*

Clarke e Baniassad (2005) propõem uma abordagem chamada *Theme*, que usa os princípios da separação de interesses. É uma evolução da separação multidimensional de interesses (MDSOC[3], *Multidimensional Separation Of Concerns*) que incorpora o conceito de entrecorte. Ela é uma proposta que visa a identificar interesses nos requisitos, encapsulá-los em temas, projetar usando temas e indicar como os temas podem ser implementados.

A abordagem é baseada no conceito de temas, que não é sinônimo de aspectos. Um tema é mais geral que um aspecto e é mais diretamente relacionado com o conceito de interesse. Cada parte da funcionalidade, ou aspecto, ou interesse, pode ser encapsulado em um tema. O relacionamento dos temas é similar ao relacionamento existente entre os requisitos ou características ou aspectos e pode assumir duas formas: compartilhamento de conceitos e entrecorte. O compartilhamento de conceitos é caracterizado quando diferentes temas possuem elementos de projeto que representam o mesmo conceito do domínio. Por exemplo, diferentes temas podem englobar a mesma classe, que vai encapsular características diferentes e importantes para cada tema. O entrecorte é um relacionamento no qual o comportamento de um tema é disparado pelo comportamento de outros temas, analogamente a um aspecto ser sensibilizado pelo código-base. Dessa forma, na abordagem *Theme* existem temas-base e temas transversais (*crosscutting themes*), e, mais raramente, temas completamente independentes de outros temas. Os temas-base se relacionam pelo compartilhamento de conceitos e podem ser entrecortados pelos temas transversais, que por sua vez podem se relacionar por compartilhamento de conceitos com outros temas.

Como os temas são análogos a interesses ou características, é possível que eles sejam projetados separadamente e posteriormente combinados. O processo proposto por *Theme*, ilustrado esquematicamente na Figura 3.3, divide-se em três atividades: análise, na qual os temas são identificados por meio da especificação de requisitos ao aplicar *Theme/Doc*[4] e ocorre o mapeamento entre requisitos e interesses; projeto, no qual os temas são projetados usando o *Theme/UML*[5], que possibilita a identificação de classes, métodos e o relacionamento entre as classes; e composição, no qual os temas são combinados seguindo as regras estabelecidas de entrecorte e compartilhamento de conceitos. *Theme* apresenta o mapeamento para que o projeto possa ser implementado e composto com AspectJ.

[3] O MDSOC é uma abordagem que permite a separação de interesses em várias dimensões igualmente importantes e não apenas em interesses-base e interesses transversais (Tarr *et al.*, 1999).

[4] *Theme/Doc* é um conjunto de heurísticas para análise dos documentos de requisitos.

[5] *Theme/UML* é a notação usada para descrever temas-base e transversais como artefatos UML.

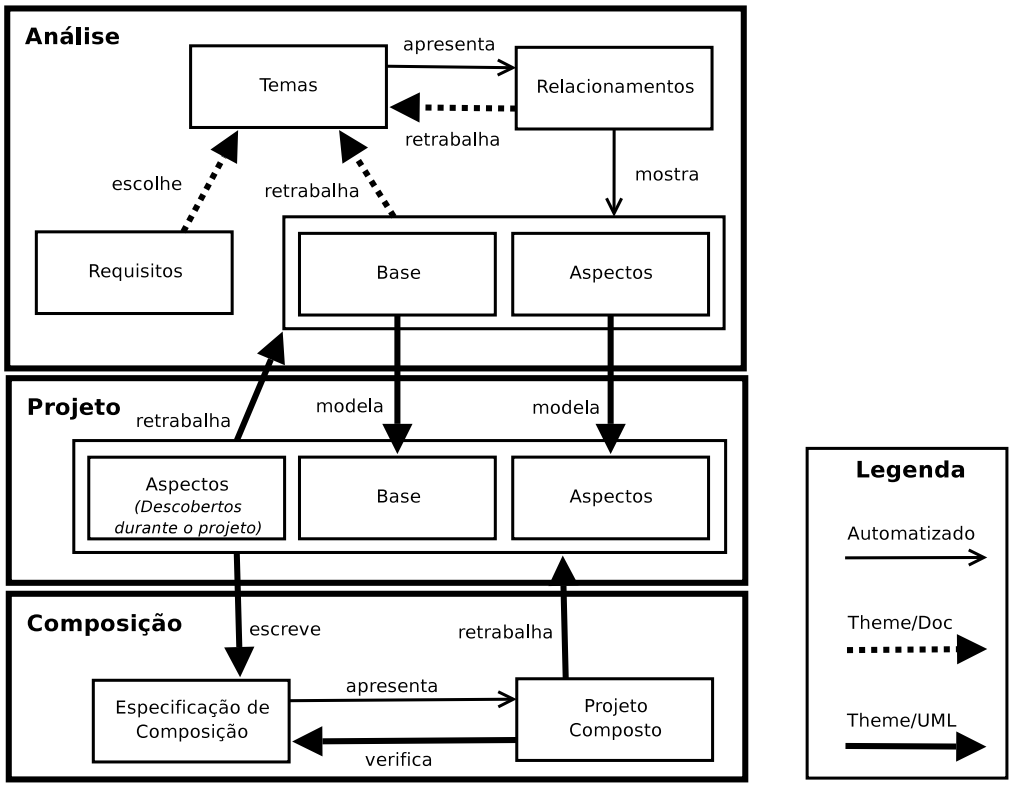


Figura 3.3: Processo do método *Theme*.

3.5 Introdução ao teste de programas orientados a aspectos

A programação OA não é uma tecnologia madura tanto quanto a programação OO, é uma abordagem recente que vêm ganhando a atenção de pesquisadores interessados em explorar seu potencial (Gradecki *et al.*, 2003; Kiczales *et al.*, 2001, 1997; Kienzle *et al.*, 2003). Com o intuito de auxiliar na atividade de teste, Alexander e Bieman (2004) identificam as possíveis origens dos defeitos em programas OA: defeitos podem estar apenas no código-base, defeitos podem estar de forma isolada apenas nos aspectos, defeitos podem ser resultado da combinação entre código-base e um aspecto; e defeitos podem ser resultado da combinação do código-base com mais de um aspecto. Pode-se perceber, então, que certos tipos de defeitos podem ser encontrados com técnicas de teste já conhecidas e outros tipos de defeitos podem ser revelados mais facilmente apenas por novas propostas. Alexander e Bieman (2004) identificam esses diferentes tipos de defeito e os classificam em um modelo de defeitos. Segundo o modelo, os defeitos podem ser de seis diferentes tipos:

Restrição incorreta em padrões de conjuntos de junção : explora enganos que podem ser cometidos na definição de conjuntos de junção, pois o desenvolvedor pode criar conjuntos de junção que são ou muito, ou pouco restritivos, capturando menos, ou mais pontos de junção do que deveriam.

Precedência incorreta de aspectos: explora enganos que o desenvolvedor pode cometer ao definir a precedência entre aspectos que afetam pontos de junção coincidentes. O problema é que diferentes conjuntos de junção utilizados por diferentes adendos de diferentes aspectos podem identificar pontos de junção coincidentes, e, neste caso, é necessário definir uma precedência entre os aspectos, o que pode ser feita erroneamente do ponto de vista semântico da aplicação.

Defeitos na preservação de pós-condições impostas ou de invariantes de estado: exploram os enganos cometidos nos aspectos que alteram dados dos componentes que afetam, violando pós-condições e invariantes de estado previamente definidas.

Foco incorreto no fluxo de controle: explora enganos cometidos em conjuntos de junção que só podem selecionar pontos de junção em tempo de execução. Pode ocorrer que pontos de junção sejam negligenciados ou tomados por engano quando esse tipo de conjunto de junção é utilizado.

Mudanças incorretas em dependências de controle: explora mudanças errôneas nas dependências de fluxo de controle que podem ocorrer a partir do mau uso de aspectos.

Ceccato *et al.* (2005) estende o modelo de defeitos de Alexander e Bieman (2004) com os seguintes tipos de defeitos:

Mudanças incorretas no fluxo de execução de exceções: explora enganos cometidos durante o lançamento de exceções ou relaxamento de exceções feitos por aspectos que alteram o fluxo de execução de classes-base ou do próprio aspecto;

Falhas originárias de declarações intertipos: explora enganos cometidos em alterações estruturais que refletem em defeitos no fluxo de execução de métodos e adendos;

Mudanças incorretas em chamadas de métodos polimórficos: explora enganos cometidos em introduções de métodos que alteram o fluxo de execução de métodos polimórficos.

Além desses, outros modelos de defeitos são propostos na literatura (Baekken e Alexander, 2006; van Deursen *et al.*, 2005; Eaddy *et al.*, 2007; Zhang e Zhao, 2007). Não obstante sejam importantes, as propostas são especializações dos modelos de Alexander e Bieman (2004) e Ceccato *et al.* (2005).

3.5.1 Técnicas e critérios teste de programas orientados a aspectos

O teste funcional de programas OA é um pouco diferente do teste em programas OO, mesmo considerando que são baseados em especificação e não em código fonte (Lopes e Ngo, 2005; Massicotte *et al.*, 2005; Xu *et al.*, 2005; Yamazaki *et al.*, 2005). O teste estrutural, por sua vez, mostra-se bastante diferente, uma vez que deve considerar detalhes de implementação de programas OA. Várias propostas abordam diferentes problemas da atividade de teste de programas OA (Anbalagan e Xie, 2006; Ferrari *et al.*, 2008; Franchin *et al.*, 2007; Lemos e Masiero, 2008; Lemos *et al.*, 2007; Zhao, 2003a; Zhou *et al.*, 2004). Com o intuito de mostrar as diferentes pesquisas no teste de programas OA, na Tabela 3.1 e na Tabela 3.2 são apresentadas breves descrições das principais propostas. Todas as propostas apresentadas nas tabelas utilizam AspectJ.

Tabela 3.1: Propostas que utilizam a técnica funcional.

Proposta	Descrição
Lopes e Ngo (2005)	Abordagem que possui dois *frameworks*, o JAML e o JAMLUnit. O JAML, baseado no JUnit (JUnit.org, 2009), usa especificações escritas em XML (W3C, 2009) para testar adendos em isolamento. O JamlUnit é usado para que os testes dos adendos possam ser escritos e executados de forma semelhante ao JUnit.
Massicotte *et al.* (2005)	Abordagem que propõe 5 critérios de teste para gerar sequências de teste baseadas na interação entre aspectos e classes especificadas em diagramas de colaboração da UML.
Xu *et al.* (2005)	Abordagem que estende o método de teste OO chamado FREE (*Flattened Regular Expression*) (Binder, 1999) para representar classes e aspectos por meio de um modelo aspectual de estados, o ASM (*Aspectual State Model*). Deriva casos de teste a partir de modelos de estados por meio da interpretação de *Statecharts* da UML com extensões específicas para a atividade de teste.
Yamazaki *et al.* (2005)	Abordagem que possui um *framework* que permite o teste de expressões de pontos de junção e de adendos sem a necessidade de combinação. A abordagem gera classes Java de teste para sensibilizar adendos e invocar métodos a partir da análise da expressão usada na assinatura do ponto de junção.

3.5.2 Fases de teste de programas orientados a aspectos

Pode-se perceber, pelos modelos de defeitos de Alexander e Bieman (2004) e Ceccato *et al.* (2005), que os defeitos podem ser resultantes da combinação de diferentes abstrações, como por exemplo, aspectos, classes, métodos e adendos, que devem ser testadas em momentos diferentes. Nesse sentido, Lemos (2005) propuseram uma classificação para as diferentes fases de teste de programas OA considerando o teste estrutural e baseando-se em Sommerville (2004) e na abordagem de Harrold e Rothermel (1994):

Tabela 3.2: Propostas que utilizam a técnica estrutural.

Proposta	Descrição
Zhao (2003a)	Abordagem que usa três tipos de grafos de programa que representam o fluxo de dados dentro de cada módulo, o fluxo de dados que envolve mais de um módulo e o fluxo de dados de cada C-Aspecto* e C-Classe*. Apesar da técnica proposta identificar módulos em C-Aspectos e C-Classes que precisam ser testados ela não define critérios de teste.
Zhou *et al.* (2004)	Abordagem que propõe um critério de teste baseado apenas na sensibilização de pontos de junção durante a integração de aspectos às classes-base.
Anbalagan e Xie (2006)	Abordagem que propõe dois operadores de mutação de conjuntos de junção para o teste de mutação de programas OA. A proposta tem uma ferramenta de geração de mutantes que faz análise estática de *bytecode* Java.
Franchin *et al.* (2007)	Abordagem em que, para que seja factível, o teste é conduzido para cada par de unidades em vez de considerar níveis arbitrários de chamadas de uma só vez. Propõe-se um grafo para representar o fluxo de controle e de dados de pares de unidades chamado grafo PWDU (*PairWise Def-Use graph*) e um conjunto de três critérios de teste baseados em fluxo de controle e de dados. O grafo é construído por meio da análise estática de *bytecode* Java usando uma ferramenta.
Lemos *et al.* (2007)	Abordagem que propõe um grafo chamado AODU (*Aspect-Oriented Def-Use graph*), construído por meio da análise estática de *bytecode* Java usando uma ferramenta. A partir do grafo são definidos critérios baseados em fluxo de dados e de controle chamado CDSTC-AOP (*Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs*).
Ferrari *et al.* (2008)	Abordagem que propõe um conjunto de operadores de mutação para o teste de mutação de programas OA. Os operadores provocam modificações em conjuntos de junção, adendos e declarações intertipos por meio da alteração de *bytecode* Java.
Lemos e Masiero (2008)	Abordagem baseada em conjuntos de junção que propõe um grafo chamado PCCFG (*Pointcut-based Control Flow Graph*) para representar regiões de execução afetadas por conjuntos de junção. O PCCFG é construído com base na análise estática de *bytecode* Java por uma ferramenta. Baseado no grafo, propõem-se dois critérios de fluxo de controle para uma medida de cobertura transversal.

*A definição dos termos C-Aspecto, C-Classe é apresentada na Seção 3.5.2.

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

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

- Intermétodo: Consiste em testar cada método público juntamente com outros métodos da mesma classe chamados direta ou indiretamente (Harrold e Rothermel, 1994) – chamadas indiretas são aquelas que ocorrem fora do escopo do próprio método, dentro de um método chamado em qualquer profundidade;
- Adendo-método: Consiste em testar cada adendo juntamente com outros métodos chamados por ele direta ou indiretamente;

- Método-adendo: Consiste em testar cada método público juntamente com os adendos que o afetam direta ou indiretamente – considerando que um adendo pode afetar outro adendo. Nesse tipo de teste não é considerada a integração dos métodos afetados com os outros métodos chamados por eles, nem com métodos chamados pelos adendos;
- Interadendo: Consiste em testar cada adendo juntamente com outros adendos que o afetam direta ou indiretamente;
- Intermétodo-adendo: Consiste em testar cada método público juntamente com os adendos que o afetam direta e indiretamente, e com métodos chamados direta ou indiretamente. Esse tipo de teste inclui os quatro primeiros tipos de teste de integração descritos anteriormente;
- Intraclasse: Consiste em testar as interações entre os métodos públicos de uma classe quando chamados em diferentes sequências (Harrold e Rothermel, 1994), considerando ou não a interação com os aspectos;
- Interclasse: Consiste em testar as interações entre classes diferentes, considerando ou não a interação dos aspectos, similar ao teste intraclasse, porém levando em conta classes que interagem entre si.

Zhao (2003a) propõem uma classificação diferente, baseada em algumas entidades presentes no programa:

- um C-Aspecto (*clustering aspect*): um aspecto juntamente com os métodos selecionados por seus pontos de junção;
- uma C-Classe (*clustering class*): uma classe juntamente com os adendos e introduções que alteram seu comportamento e estrutura;
- um C-método (*clustering method*): um método juntamente com os adendos que o afetam.

Para os autores a menor unidade em teste são os C-Aspectos e as C-Classes, conforme mostrado na Tabela 3.3. A proposta de Lemos (2005) apresenta vantagens em relação à proposta de Zhao (2003a) por permitir que os testes sejam realizados isoladamente em adendos ou em métodos, diferentemente de Zhao (2003a), cuja unidade já é a combinação entre aspectos e classes-base.

O teste de unidade em programas OA pode ser caracterizado segundo o tipo de unidade em teste. Diferentemente da programação OO, na programação OA os tipos de unidades são dois: método e adendo. O teste do método procede da mesma forma que no paradigma OO, porém na

Tabela 3.3: Relação entre fases de teste entre as propostas de Lemos (2005) e Zhao (2003a).

Menor Unidade: Método/Adendo	
Fase	Teste de programas OA
Unidade	Intramétodo e Intra-adendo
Integração	Intermétodo, Método-adendo, Adendo-método, Adendo-adendo, Intraclasse e Interclasse
Sistema	Toda a Aplicação

Menor Unidade: C-Classe/C-Aspecto	
Fase	Teste de programas OA
Unidade	Intramódulo, Intermódulo, Intraclasse e Intra-aspecto
Integração	Interclasse
Sistema	Toda a Aplicação

programação OA, o teste de adendos não é tão simples. A ideia de testar aspectos isoladamente, como da forma tradicional, não é apropriada porque aspectos não possuem identidade independente e não são instanciáveis (Alexander e Bieman, 2004; Lopes e Ngo, 2005). Além disso, aspectos possuem forte acoplamento com as informações do contexto de execução e, portanto, só podem ser testados quando combinados com as classes-base.

O teste de integração, no sentido de estabelecer uma ordem de teste entre classes e aspectos, é abordado nos trabalhos de Ceccato *et al.* (2005), Massicotte *et al.* (2005) e Zhou *et al.* (2004). No primeiro trabalho, os autores apenas discutem o teste de integração sem apresentar detalhes de como proceder o teste de integração incremental. No entanto, discutem o problema de dependências entre classes e aspectos e a necessidade de implementação de *stubs* e *drivers* para a condução da atividade de teste. O segundo trabalho propõe a integração incremental mas ignora o problema de dependências entre aspectos e classes. O terceiro trabalho propõem, também, a integração incremental, mas não discute o problema dos ciclos de dependência ou os problemas e defeitos que podem ocorrer durante essa fase de teste. O trabalho visa a selecionar casos de teste e avaliar a cobertura dos casos de teste selecionados no contexto de teste de integração.

3.6 Ferramentas de teste

Além do *framework* JAML (Lopes e Ngo, 2005) e do *framework* proposto por Yamazaki *et al.* (2005), foram encontradas apenas três ferramentas que apoiam o teste de programas OA, todas direcionadas para programas implementados em AspectJ: *JaBUTi/AJ* (Lemos, 2005) e a ferramenta de Zhou *et al.* (2004).

JaBUTi/AJ (*Jaba Bytecode UnderstandIng and Testing/ AspectJ*) é uma extensão da ferramenta *JaBUTi* também desenvolvida no ICMC/USP para permitir o uso dos critérios todos-

nós-transversais, todas-arestas-transversais e todos-usos-transversais no teste de unidade. Para tanto, a implementação do grafo Def-Uso foi alterada para gerar o grafo $\mathcal{AODU}$. A ferramenta deriva os requisitos de teste a partir dos critérios de teste selecionados durante a sessão de teste e calcula o peso dos requisitos, para indicar quais requisitos são cobertos a partir da cobertura de um requisito qualquer. Com isso, ela permite que a qualidade dos conjuntos de teste seja avaliada e, também, que novos casos de teste sejam desenvolvidos com o intuito de melhorar a cobertura de teste.

Zhou *et al.* (2004) propõem uma ferramenta para seleção de casos de teste relevantes para testar aspectos que entrecortam classes já testadas e para o cálculo de cobertura dos casos de teste selecionados. Os autores não dão muitas informações sobre o funcionamento da ferramenta, mas o processo geral de funcionamento é iniciado pela identificação de quais casos de teste sensibilizam quais aspectos. Posteriormente, a ferramenta utiliza um algoritmo que utiliza as informações a respeito da sensibilização de adendos, e consequentemente aspectos, para calcular o que os autores chamam de cobertura de teste. Com isso, é possível determinar quais casos de teste são relevantes para cada aspecto. No entanto, conforme comentado na Seção 3.5.2, pode-se notar um problema no algoritmo, uma vez que não se pode inferir que um caso de teste sensibiliza um dado aspecto pelo simples fato do caso de teste chamar um método que é afetado pelo aspecto, pois isso pode depender da entrada do caso de teste.

3.7 Considerações finais

Neste capítulo, foram apresentadas as principais características de desenvolvimento de software baseado em separação de interesses e da programação OA, bem como das abordagens de teste de software neste contexto. Como visto, as propostas seguem em diferentes direções, mas são, em sua grande maioria, evoluções de soluções propostas para o teste de programas OO, como por exemplo: reformulação do conceito de fases de teste; evolução das técnicas de teste por meio da incorporação de novos critérios de teste; e estratégias de teste que incorporam os novos conceitos de teste de unidade, de integração e de sistema.

Um resultado significativo da revisão apresentada foi a ausência de trabalhos que tratam explicitamente do problema da ordenação de classes e aspectos no teste de integração de OA, apesar da existência de indícios de que este problema também existe na programação OA. Alguns trabalhos mencionam a existência de interdependências entre aspectos e classes e discutem a necessidade de implementação de *stubs* para que a atividade de teste possa ser conduzida. Além disso, como a programação OA é uma evolução da programação OO, ela sofre com problemas semelhantes, conforme discutido anteriormente. É possível notar também que o problema da

ordenação de classes e aspectos é ainda mais complexo na programação OA porque existe um novo tipo de entidade e novos tipos de relacionamento entre entidades.

Uma proposta para o problema da ordenação de classes e aspectos no teste de integração OA é apresentada no Capítulo 4. A proposta é uma evolução de um trabalho que trata do mesmo problema na programação OO, a estratégia de ordenação proposta por Briand *et al.* (2003). O algoritmo de ordenação usado pela estratégia é adaptado para a programação OA. Para tanto, um novo modelo de dependências aspectuais que descreve os relacionamentos entre classes e aspectos é proposto no Capítulo 4. Esse modelo permite que os relacionamentos sejam modelados em uma extensão do ORD. Além disso, para que a proposta seja factível frente à imaturidade das propostas de modelagem de programas OA, é apresentado na Capítulo 5 um processo de mapeamento que descreve como um modelo de projeto OA pode ser mapeado para a extensão do ORD proposta no Capítulo 4. Como uma forma de validação da adaptação do algoritmo de Briand *et al.* (2003), do modelo de dependências aspectuais, da extensão do ORD e do processo de mapeamento, é apresentado no Capítulo 6 um estudo exploratório conduzido em três sistemas implementados em AspectJ. Durante o estudo exploratório foram coletados *stubs* e *drivers* de teste que foram utilizados recorrentemente. Essa amostra de tipos de *stubs* e *drivers* deu origem a um catálogo de casos de implementação de *stubs* e *drivers* proposto no Capítulo 7.

CAPÍTULO

4

Proposta de duas estratégias de ordenação de classes e aspectos no teste de integração OO/OA

4.1 Considerações iniciais

Não foram encontrados trabalhos na literatura especializada que visam à minimização do número de *stubs* durante o teste de integração de programas OA. Para preencher esta lacuna, neste capítulo são apresentadas duas estratégias de ordenação de classes e aspectos OO/OA propostas que têm por objetivo minimizar o número de *stubs* no teste de integração de programas OA. Para tanto, um modelo de dependências aspectuais que descreve como aspectos e classes são interdependentes é proposto. Esse modelo de dependências aspectuais determina quais mecanismos de conexão são importantes para a definição das dependências ao denotar relacionamentos entre classes e aspectos. Os mecanismos de conexão são definidos a partir de código fonte AspectJ, uma vez que pretende-se apoiar o teste de integração baseado em código. A linguagem AspectJ foi escolhida por ser a tecnologia mais madura no desenvolvimento de programas OA. Esse modelo de dependências poderia ser estendido para outras linguagens OA, tais como AspectC++ (AspectC.org, 2009) e AspectS (de Alwis e Kiczales, 2009), sem muitas dificuldades.

Com base no modelo de dependências aspectuais, propõe-se uma extensão do ORD para incluir as dependências aspectuais. Esse diagrama é chamado AORD (Aspect and Object Relation Diagram). As estratégias de ordenação utilizam o AORD para que o algoritmo de ordenação usado na estratégia de Briand *et al.* (2003), adaptado para acomodar aspectos e as dependências do modelo de dependências aspectuais, produza a ordem de teste e integração para cada uma das estratégias.

Para exemplificar como os mecanismos de conexão de programas OA geram dependências no AORD e também para mostrar como o AORD é usado para computar a ordem de teste e

integração, alguns AORDs são construídos a partir de um sistema implementado em AspectJ. Posteriormente, exemplifica-se como os AORDs são usados pelas estratégias de ordenação.

Na Seção 4.2 é apresentado o AORD e como ele estende o ORD. Ainda nessa seção, o modelo de dependências aspectuais é definido e correlacionado com o AORD. As duas estratégias de ordenação propostas são apresentas na Seção 4.3. As considerações finais são apresentadas na Seção 4.4.

4.2 Proposta de um diagrama de relacionamento de classes e aspectos

Os tipos de dependências entre aspectos e entre aspectos e classes são diferentes daqueles encontrados apenas entre classes. Para estender a estratégia proposta por Briand *et al.* (2003) foi necessário alterar o ORD original para representar adequadamente os novos tipos de dependências.

As estratégias de ordenação aqui propostas baseiam-se na utilização do AORD (Aspect and Object Relation Diagram). O AORD é uma extensão do ORD que acomoda os novos tipos de relacionamento encontrados na programação OA. Além das dependências de herança ("I"), de agregação ("Ag") e de associação ("As"), existem novas dependências de declaração intertipos ("It"), de entrecorte ("C"), de uso de conjuntos de junção ("U") e de relaxamento de exceções ("S"). Na Tabela 4.1 são listadas as formas como os relacionamentos denotam interdependências entre classes e aspectos. O AORD estende o ORD conforme definido a seguir:

Definição 1: Seja $G = (V, L, E)$ um dígrafo simples com arestas rotuladas, no qual $V = \{v_1, ..., v_n\}$ é um conjunto finito de vértices, $L = \{l_1, ..., l_k\}$ é um conjunto finito de rótulos e $E \subseteq V \times V \times L$ é um conjunto finito de arestas rotuladas.

Definição 2: Para um sistema OA SIS composto de classes e aspectos, denominados módulos, seja $CLA(SIS)$ o conjunto de todas as classes de SIS, $ASP(SIS)$ o conjunto de todos os aspectos de SIS e $V = \{m_i | m_i \in CLA(SIS) \vee m_i \in ASP(SIS)\}$ um conjunto finito de vértices representando as classes e os aspectos de SIS.

Definição 3: O AORD para SIS é um dígrafo simples com arestas rotuladas $AORD = (V, L, E)$, no qual V é o conjunto de vértices representando classes e aspectos, $L = \{I, It, Ag, As, C, U, S\}$ é o conjunto de rótulos das arestas de dependência que representam os relacionamentos e $E \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas rotuladas.

Definição 4: $E = E_I \cup E_{It} \cup E_{Ag} \cup E_{As} \cup E_C \cup E_U \cup E_S$ é o conjunto de arestas de dependência representando os relacionamentos de herança, de declaração intertipos, de agregação, de associação, de entrecorte, de uso de conjuntos de junção e de relaxamento de exceções, conforme definido no modelo de dependências aspectuais apresentado a seguir.

4.2.1 Modelo de dependências aspectuais

O conceito de dependência, no contexto de estratégias de ordenação, diz respeito à necessidade que um aspecto ou classe tem da existência de outros aspectos e classes para que seja possível efetuar os testes. Portanto, são dependências detectáveis por meio da sintaxe da linguagem de programação OA, uma vez que para conduzir o teste baseado em código o aspecto ou classe em teste deve ser compilado e executado. O modelo de dependências aspectuais aqui proposto foi desenvolvido a partir dos relacionamentos entre classes e aspectos encontrados no código fonte de programas OA implementados em AspectJ. Ele tem por finalidade descrever como ocorrem as interdependências entre classes e aspectos e, assim, determinar os tipos de arestas do AORD.

Na Tabela 4.1 é apresentado como se dá o relacionamento par a par de classes e aspectos. Os participantes do relacionamento denotam dependências no formato cliente/servidor, indicando que o módulo cliente depende do módulo servidor. Os relacionamentos de herança e de associação são estabelecidos entre qualquer par aspecto/classe, com apenas uma exceção: segundo a sintaxe de AspectJ, não há relacionamento de herança em que uma classe seja especialização de um aspecto. Existem três tipos de relacionamentos que podem ocorrer na forma aspecto/classe e aspecto/aspecto: o relacionamento de entrecorte, o relacionamento de declaração intertipos e o relacionamento de relaxamento de exceção. Isso se deve ao fato de que aspectos podem entrecortar classes e também outros aspectos, um relacionamento que, obviamente, não pode existir apenas entre classes. Da mesma forma, um aspecto pode conter declarações intertipos que afetam classes e aspectos, e um aspecto pode efetuar relaxamento de exceções em outras classes e aspectos. O relacionamento de agregação pode existir apenas entre classes e em contrapartida, o relacionamento de uso de conjunto de junção pode existir apenas entre aspectos.

Tabela 4.1: Formas de relacionamento entre classes e aspectos.

Cliente/Servidor	**Classe**	**Aspecto**
Classe	Herança, Agregação e Associação.	Associação.
Aspecto	Herança, Associação, Declaração intertipos, Entrecorte e Relaxamento de exceção	Herança, Associação, Declaração intertipos, Entrecorte, Relaxamento de exceção e Uso de conjunto de junção.

Os relacionamentos apresentados na Tabela 4.1 são representados em AspectJ por meio de elementos sintáticos presentes na linguagem, chamados de mecanismos de conexão. Com exceção do relacionamento de agregação, que correga consigo valor semântico, todos os outros podem ser rastreados diretamente no código fonte (Briand *et al.*, 2001, 2003). O importante para a definição do modelo de dependências são os mecanismos de conexão que conectam classes e aspectos, apresentados na Tabela 4.2. Existem mecanismos de conexão próprios da programação OO e mecanismos de conexão próprios da programação OA, que denotam dependências somente entre classes, somente entre aspectos e entre aspectos e classes. A primeira coluna da tabela indica se o mecanismo é proveniente da programação OO ou OA. A segunda coluna dá nome ao mecanismo de conexão enquanto que a terceira coluna mostra uma sigla referente ao mecanismo. Por fim, na quarta coluna são mostradas as dependências que podem ser mapeadas a partir do mecanismo de conexão. Note-se que, conforme descrito nas definições a seguir, mecanismos de conexão OA podem dar origem a mais de uma dependência no AORD.

Tabela 4.2: Mecanismos de conexão do modelo de dependências.

Paradigma	Mecanismo de conexão	Sigla	Dependência
OO	Declaração de herança	DH	I
OO	Declaração de atributos	DA	As
OO	Declaração de variáveis locais	DVL	As
OO	Tipo de parâmetros de retorno de métodos	TPRM	As
OO	Referência a atributos	RA	As
OO	Chamada de métodos	CM	As
OA	Declaração intertipos de alteração de hierarquia de herança	DIAHH	I, It
OA	Declaração intertipos de introdução de atributo	DIIA	It, As
OA	Declaração intertipos de introdução de métodos	DIIM	It, As
OA	Conjuntos de junção	CJ	C, As
OA	Tipo de parâmetros de retorno de adendos	TPRA	As
OA	Uso de conjuntos de junção	UCJ	U
OA	Declaração de relaxamento de exceção	DRE	S

Um ponto importante a ser observado é que o AORD tem como granularidade classes e aspectos e não modela seus membros internos[1]. Assim, as dependências ligam sempre aspectos e classes, não importando os detalhes que mostram como os membros internos se conectam. O mecanismo de conexão que denota uma dependência de herança é identificado na própria declaração da classe ou do aspecto, mostrando que a classe ou o aspecto pode ser diretamente dependente de outra classe ou aspecto. Algumas dependências podem ser identificadas porque os membros internos de uma classe ou aspecto se conectam com membros internos de outra

[1]Entende-se por membros internos: atributos, métodos, conjuntos de junção, adendos, declarações intertipos e declarações de relaxamento de exceção.

classe ou aspecto. Existem ainda mais dois casos: as dependências que podem ser identificadas porque os membros internos de uma classe ou aspecto se conectam diretamente com outra classe ou aspecto e as dependências que podem ser identificadas porque os aspectos se conectam a membros internos de outros aspectos.

São apresentadas a seguir as definições de dependências OO importantes para a programação OA e, de forma detalhada, as dependências específicas de programação OA.

Definição 5. $E_I \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando o relacionamento de herança conforme definido a seguir:

1 Para quaisquer classes $c_1, c_2 \in V$, $\langle c_2, c_1, I\rangle \in E_I$, indica que c_2 é uma especialização de c_1.
2 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_2, a_1, I\rangle \in E_I$, indica que a_2 é uma especialização de a_1.
3 Para qualquer aspecto e classe, respectivamente, $a_1, c_1 \in V$, $\langle a_1, c_1, I\rangle \in E_I$, indica que a_1 é uma especialização de c_1.

A Definição 5 descreve como o relacionamento de herança, denotado pelos mecanismos de conexão DH e DIAHH, dá origem a uma dependência do tipo "I" no AORD. Note-se que, diferentemente dos mecanismos de conexão OO, o mecanismo DIAHH denota ao mesmo tempo dois tipos de dependência, no qual uma delas é do tipo "I". Em AspectJ, uma aresta do tipo "I" é identificada no código fonte se e apenas se na declaração de classes e aspectos de SIS ou em declarações intertipos forem encontradas instruções como em um dos casos a seguir:

- c_2 `extends` c_1;
- a_2 `extends` a_1;
- a_1 `extends` c_1;
- `declare parents:` c_2 `extends` c_1;
- `declare parents:` a_2 `extends` a_1;
- `declare parents:` a_1 `extends` c_1;

Definição 6. $E_{Ag} \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando o relacionamento de agregação entre duas classes. Para quaisquer classes $c_1, c_2 \in V$, $\langle c_1, c_2, Ag\rangle \in E_{Ag}$, indica que c_1 contém um ou mais objetos da classe c_2.

Conforme mencionado, o relacionamento de agregação é semântico e não suportado como uma construção sintática própria pela grande maioria das linguagens de programação OO, senão por todas. Portanto, o rastreamento de dependências do tipo "Ag" não pode ser completamente automatizado e necessita de interferência humana (Briand *et al.*, 2001, 2003). De maneira genérica, uma dependência do tipo "Ag" é rastreada por meio da análise da declaração de atributos nas classes.

Definição 7. $E_{As} \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando o relacionamento de associação em que quaisquer classes ou aspectos $\langle m_1, m_2, As \rangle \in E_{As}$, indica que m_1 associa-se a m_2 por pelo menos uma das seguintes opções:

1 Um objeto do tipo m_2 é declarado como atributo de m_1;
2 Um objeto do tipo m_2 é declarado como variável local de m_1;
3 Um método de m_1 usa um atributo de m_2 ou faz uma chamada a um método de m_2;
4 Um objeto do tipo m_2 é usado como parâmetro ou retorno de método de um método de m_1;
5 Um método de m_1 invoca um método herdado por qualquer classe ou aspecto de SIS via alteração de hierarquia de herança feita por m_2;
6 Um método de m_1 usa um atributo herdado por qualquer classe ou aspecto de SIS via alteração de hierarquia de herança feita por m_2;
7 Um método de m_1 invoca um método introduzido por m_2 em qualquer classe ou aspecto de SIS;
8 Um método de m_1 usa um atributo introduzido por m_2 em qualquer classe ou aspecto de SIS;
9 Um adendo de m_1 usa um atributo de m_2 ou faz uma chamada a um método de m_2;
10 Um objeto do tipo m_2 é usado como parâmetro ou retorno de adendo de um adendo de m_1;
11 Um adendo de m_1 invoca um método herdado por qualquer classe ou aspecto de SIS via alteração de hierarquia de herança feita por m_2;
12 Um adendo de m_1 usa um atributo herdado por qualquer classe ou aspecto de SIS via alteração de hierarquia de herança feita por m_2;
13 Um adendo de m_1 invoca um método introduzido por m_2 em qualquer classe ou aspecto de SIS;
14 Um adendo de m_1 usa um atributo introduzido por m_2 em qualquer classe ou aspecto de SIS;

15 Uma declaração intertipos de m_1 introduz um atributo em uma classe ou aspecto de SIS e o tipo do atributo introduzido é m_2;

16 Uma declaração intertipos de m_1 introduz um método em uma classe ou aspecto de SIS e o tipo do retorno do método introduzido é m_2;

17 Uma declaração intertipos de m_1 introduz um método em uma classe ou aspecto de SIS e o tipo de um parâmetro do método introduzido é m_2;

18 Uma declaração intertipos de m_1 introduz um método em uma classe ou aspecto de SIS e existe no método introduzido uma chamada para um método de m_2 ou um uso de um atributo de m_2;

19 Uma declaração intertipos de m_1 introduz um método em uma classe ou aspecto de SIS e existe no método introduzido uma chamada para um método introduzido por m_2 ou um uso de um atributo introduzido por m_2;

20 Uma declaração intertipos de m_1 introduz um método em uma classe ou aspecto de SIS e existe no método introduzido uma chamada para um método herdado via declaração de alteração de hierarquia feita por m_2;

21 Uma declaração intertipos de m_1 introduz um método em uma classe ou aspecto de SIS e existe no método introduzido um uso de um atributo herdado via declaração de alteração de hierarquia feita por m_2;

22 Um objeto do tipo m_2 é usado como parâmetro de um conjunto de junção de m_1;

23 Um conjunto de junção de m_1 com comando condicional `If` usa um atributo de m_2 ou invoca um método de m_2;

24 Um conjunto de junção de m_1 com comando condicional `If` invoca um método herdado por qualquer classe ou aspecto de SIS via alteração de hierarquia de herança feita por m_2;

25 Um conjunto de junção de m_1 com comando condicional `If` usa um atributo herdado por qualquer classe ou aspecto de SIS via alteração de hierarquia de herança feita por m_2;

26 Um conjunto de junção de m_1 com comando condicional `If` invoca um método introduzido por m_2 em qualquer classe ou aspecto de SIS;

27 Um conjunto de junção de m_1 com comando condicional `If` usa um atributo introduzido por m_2 em qualquer classe ou aspecto de SIS;

Conforme pode-se observar na Definição 7, as dependências do tipo "As" são denotadas pelos mecanismos de conexão OO: DA, DVL, TPRM, RA e CM. Além desses mecanismos, os mecanismos de conexão OA TPRA, DIIA, DIIM e CJ também denotam dependência do tipo

"As". Estes três últimos também são exemplos de mecanismos que podem denotar simultâneamente dois tipos de dependência.

Definição 8. $E_{It} \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando relacionamentos de declarações intertipos de introdução de método, introdução de atributo e alteração de hierarquia de herança, denotados pelos mecanismos de conexão DIAHH, DIIA e DIIM, conforme definido a seguir:

1 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, It\rangle \in E_{It}$, indica que a_1 faz uma declaração intertipos cujo alvo é a_2.
2 Para qualquer aspecto e classe, respectivamente, $a_1, c_1 \in V$, $\langle a_1, c_1, It\rangle \in E_{It}$, indica que a_1 faz uma declaração intertipos cujo alvo é c_1.
3 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, It\rangle \in E_{It}$, indica que a_1 faz uma alteração de hierarquia de herança fazendo com que a_2 se torne uma especialização ou generalização de qualquer outra classe ou aspecto de SIS.
4 Para qualquer aspecto e classe, respectivamente, $a_1, c_1 \in V$, $\langle a_1, c_1, It\rangle \in E_{It}$, indica que a_1 faz uma alteração de hierarquia de herança fazendo com que c_1 se torne uma especialização ou generalização de qualquer outra classe ou aspecto de SIS.

Definição 9. $E_C \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando relacionamento de entrecorte, denotado pelo mecanismo de conexão CJ, conforme definido a seguir:

1 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, C\rangle \in E_C$, indica que a_1 possui um conjunto de junção que entrecorta a_2.
2 Para qualquer aspecto e classe, respectivamente, $a_1, c_1 \in V$, $\langle a_1, c_1, It\rangle \in E_C$, indica possui um conjunto de junção que entrecorta c_1.
3 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, C\rangle \in E_C$, indica que a_1 possui um conjunto de junção cujo ponto de junção por ele capturado é determinado a partir da avaliação do fluxo de execução de programa que passa por a_2.
4 Para qualquer aspecto e classe, respectivamente, $a_1, c_1 \in V$, $\langle a_1, c_1, It\rangle \in E_C$, indica que possui um conjunto de junção cujo ponto de junção por ele capturado é determinado a partir da avaliação do fluxo de execução de programa que passa por c_1.

Definição 10. $E_U \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando relacionamento de uso de conjuntos de junção, denotado pelo mecanismo de conexão UCJ, conforme definido a seguir:

1 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, C\rangle \in E_U$, indica que a_1 possui um adendo que usa um conjunto de junção definido em a_2.
2 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, C\rangle \in E_U$, indica que a_1 possui um conjunto de junção que usa um conjunto de junção definido em a_2.
3 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, C\rangle \in E_U$, indica que a_1 possui uma declaração de relaxamento de exceção que usa um conjunto de junção definido em a_2.
4 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_1, a_2, C\rangle \in E_U$, indica que a_1 é um aspecto não *singleton* e que o conjunto de junção usado para definir o modo de instanciação usa um conjunto de junção implementado em a_2

Definição 11. $E_S \subseteq V \times V \times L$ é o conjunto de arestas de dependência direcionadas representando relacionamento de relaxamento de exceção, denotado pelo mecanismos DRE, conforme definido a seguir:

1 Para quaisquer aspectos $a_1, a_2 \in V$, $\langle a_2, a_1, C\rangle \in E_S$, indica que a_2 possui uma exceção relaxada por a_1.
2 Para qualquer aspecto e classe, respectivamente, $c_1, a_1 \in V$, $\langle a_1, c_1, It\rangle \in E_S$, indica que c_1 possui uma exceção relaxada por c_1.
3 Para qualquer aspecto e classe, respectivamente, $a_1, c_1 \in V$, $\langle a_1, c_1, It\rangle \in E_S$, indica que c_1 é uma exceção relaxada por c_1.

4.2.2 Discussão sobre o acoplamento denotado pelas dependências do AORD

O AORD mostra as dependências entre classes e aspectos usando o modelo de dependências aspectuais proposto, que tem uma utilização específica: descrever como ocorrem as dependências entre classes e aspectos e, assim, apoiar a minimização do número de *stubs* pela remoção temporária de arestas de dependência que denotam menor acoplamento. O acoplamento é definido pelo grau de interdependência entre pares de módulos (Jin e Offutt, 1996), que é determinado pelo tipo de mecanismo de conexão e a frequência de sua utilização para conectar dois módulos (Briand *et al.*, 1999). O acoplamento entre dois módulos indica as relações de dependência entre eles, e defeitos de um podem se refletir no outro (Abdurazik e Offutt, 2006). Existem vários trabalhos que se propõem a computar o acoplamento entre classes utilizando métricas específicas.

Não obstante a existência de vários trabalhos que tratam do teste de integração frente ao problema de minimização do número de *stubs*, apenas as propostas de Briand *et al.* (2003);

Kung *et al.* (1995a) e Abdurazik e Offutt (2006) relacionam o conceito do acoplamento ao custo de implementação de *stubs*, enquanto os demais não se preocupam com essa questão. Segundo Briand *et al.* (2003), dependências de herança e agregação não devem ser removidas temporariamente porque geram *stubs* de maior complexidade. Assim, a implementação de um *stub* referente a uma remoção temporária de aresta que representa um relacionamento de baixo acoplamento é menos complexa e custosa.

Kung *et al.* (1995a) usam como argumento o fato de que não há possibilidade de haver ciclos em um sistema em que existam apenas relacionamentos de herança e agregação, e alegam que é possível provar a afirmação, mas não apresentam tal demonstração. Apesar da estratégia de Briand *et al.* (2003) usar como métrica de eficiência de sua proposta o número de *stubs* implementados, eles apresentam um estudo que usa como métrica de complexidade desses *stubs*, e consequentemente estimar o custo de sua implementação, o número de métodos e o número de atributos simulados. O custo de implementação é então usado para confirmar que relacionamentos de herança e agregação não devem ser removidos.

Usando a mesma ideia, Abdurazik e Offutt (2006) usam duas métricas de acoplamento diferentes para medir a complexidade dos *stubs*, ambas baseadas em 10 tipos de relacionamento. O cálculo da complexidade de um *stubs* é feito com base no tipo de relacionamento em função do número de atributos referenciados, de métodos invocados, de parâmetros de métodos e de valores retornados simulados. Apesar de mais elaborada, a proposta não mostra muito avanço porque segue as diretrizes propostas por Briand *et al.* (2003), nas quais os relacionamentos de herança e de agregação denotam acoplamento maior que os outros tipos.

Conforme pode ser notado, o problema a ser considerado é a medição do acoplamento dos relacionamentos para determinar quais arestas devem ser removidas do AORD. Neste trabalho, o foco são as dependências estruturais aspectuais, que ocorrem no formato classe/aspecto, aspecto/classe e aspecto/aspecto. Para tanto, uma análise do acoplamento causado pelas dependências descritas no AORD torna-se necessária.

Não existem métricas de acoplamento comprovadas ou modelos de dependências para programas OA que auxiliam na determinação do acoplamento referente aos tipos de relacionamento entre classes e aspectos, quanto mais estudos sobre o custo de implementação de *stubs* em programação OA.

Alguns trabalhos propõem métricas de acoplamento de programas OA (Bartsch e Harrison, ????; Ceccato e Tonella, 2004; Sant'anna *et al.*, 2003; Shen *et al.*, 2008; Zhao, 2003b), porém não apresentam estudos para verificar as métricas propostas. Além disso, um ponto importante a ser observado é que os trabalhos não tratam todas as dependências aspectuais do modelo aqui proposto, mostrado na Seção 4.2.1. Apesar dos relacionamentos de dependência estarem

ligados ao conceito de acoplamento (Gélinas *et al.*, 2006; Jin e Offutt, 1996), poucos apresentam e correlacionam o conceito de acoplamento a um modelo de dependências aspectuais. Apenas o trabalho de Zhao (2002) correlaciona métricas OA com um modelo de dependências, definindo informalmente o que são dependência de controle e de dados no contexto de grafos de fluxo de dados e de controle. Porém esse trabalho não mostra como computar acoplamento entre aspectos e classes e não apresenta estudos para validar as métricas.

Considerando que escolher dependências que representam menor acoplamento para serem removidas do AORD resultam em menor custo de implementação de *stubs*, são adotas as diretrizes de Briand *et al.* (2003): dependências de herança e de agregação não devem ser removidas. O problema das dependências aspectuais é resolvido com a ideia de que algumas das dependências da programação OA são semelhantes às da programação OO e, nesse caso, também são adotadas as diretrizes de Briand *et al.* (2003)

Dependências do tipo "It" ocorrem por meio dos mecanismos de conexão DIAHH, DIIA e DIIM. Quando um método é introduzido ele pode utilizar atributos e outros métodos contidos na classe alvo não importando sua visibilidade, o que denota um forte acoplamento de dados e de controle, semelhante ao acoplamento de herança. A alteração de hierarquia de herança pode fazer com que chamadas de métodos entre subclasses e superclasses sejam alteradas, o que também denota forte acoplamento de controle. A introdução de atributos demonstra apenas que existe acoplamento de dados mas é aparentemente mais fraca que as duas anteriores. Dessa forma, dependências do tipo "It" não devem ser removidas temporariamente do AORD, assim como a dependências dos tipos "I" e "Ag".

Dependências do tipo "As" são estabelecidas por meio dos mecanismos de conexão DA, DVL, TPRM, TPRA, DIIA e DIIM. O uso desses mecanismos causam dependências semelhantes. Por exemplo, o tipo de um atributo declarado em uma classe causa a mesma dependência entre a classe em que o atributo é declarado e a classe que é usada como tipo. Um atributo introduzido também pode ter uma classe usada como tipo, mas a dependência nesse caso é do aspecto que faz a introdução com a classe que é usada como tipo. É importante notar que os aspectos também podem ser usados como tipos. Outro ponto importante a ser observado é que métodos introduzidos têm tipos de retorno e de parâmetros. Assim, considera-se que o uso desses mecanismos gera dependências do tipo "As" semelhantes àquelas da programação OO.

Dependências do tipo "As" também são estabelecidas quando os mecanismos de conexão CM, RA e CJ são utilizados. Note-se que existem três casos distintos de utilização que devem ser estudados com mais atenção:

- Quando um método, um adendo, um conjunto de junção ou um método introduzido estabelece conexão com atributos e outros métodos. Nestes casos, as dependências são

semelhantes àquelas encontradas em programação OO, porque o que denota a dependência é a invocação de métodos e a referência a atributos, mesmo considerando os conjuntos de junção.

- Quando um método, um adendo ou um conjunto de junção estabelece conexão com atributos e outros métodos herdados via utilização do mecanismo DIAHH ou com atributos e outros métodos introduzidos via utilização dos mecanismos DIIA e DIIM. Estes são casos em que referências a atributos e chamadas a métodos dependem tanto do aspecto que implementa a declaração intertipos como da classe alvo das declarações intertipos. Assim, deve-se registrar no AORD apenas a dependência entre a classe ou aspecto que faz a referência ou a chamada ao método com o aspecto que implementa a declaração intertipos. Considera-se que essa é uma dependência do tipo "As" porque, com a convenção da não remoção de arestas do tipo "It", atributos e métodos introduzidos estarão sempre combinados com a classe alvo e podem ser referenciados ou invocados diretamente. Assim, a consistência do AORD é garantida, porque o aspecto que implementa o AORD sempre estará ligado à classe alvo da introdução.
- Quando um método introduzido estabelece conexão com atributos e outros métodos herdados via utilização do mecanismo DIAHH ou com atributos e outros métodos introduzidos via utilização dos mecanismos DIIA e DIIM. Este caso é semelhante ao anterior, mas com a diferença de que tanto o cliente da dependência do tipo "As" quanto o servidor possuem declarações intertipos que se relacionam.

Em dependências do tipo "C", estabelecidas pelo mecanismo CJ, o cliente é sempre um aspecto, enquanto o servidor pode ser uma classe ou outro aspecto. É importante observar que dependências desse tipo denotam dependência de controle e de dados, uma vez que um conjunto de junção pode invocar métodos que alteram parâmetros e fazer referências a atributos. Esse tipo de dependência não apresenta forte acoplamento como dependências do tipo "It", mesmo considerando que o mecanismo de quantificação seja poderoso para afetar vários pontos de junção. Assim, dependências do tipo "C" podem ser removidas do AORD da mesma maneira que dependências do tipo "As".

Em dependências do tipo "U", estabelecidas pelo mecanismo UCJ, tanto o cliente como o servidor são aspectos. Dependências do tipo "U" denotam dependência de controle e pode denotar de dados, quando adendos usam conjuntos de junção. Portanto, esses dois tipos de dependências podem ser removidos do AORD assim como dependências do tipo "As".

Dependências do tipo "S" estabelecidas pelo mecanismo DRE acontecem quando uma exceção é relaxada, fazendo com que o ponto de junção capturado dependa do aspecto que faz o relaxamento e este, por sua vez, dependa do tipo de exceção lançada. O que caracteriza essa

dependência é que um módulo cliente não pode ser compilado sem que uma exceção não tratada em qualquer de seus métodos seja relaxada. Não existem dependências na programação OO parecidas com essa. Além disso, a declaração de relaxamento de exceção é uma construção bastante específica da linguagem AspectJ. Portanto, decidiu-se que dependências do tipo "S" podem ser removidas do AORD assim como dependências do tipo "As". Com isso, nos estudos conduzidos no Capítulo 6 foi possível estudar como implementar *stubs* para simular o comportamento de aspectos com declarações de relaxamento de exceção.

4.2.3 Exemplo de construção de um AORD baseado em engenharia reversa

Nesta seção é apresentado um exemplo de como o modelo de dependências aspectuais proposto é usado na construção de um AORD. O sistema utilizado no exemplo é o sistema Telecom. A descrição do sistema Telecom é feita considerando que existam classes-base, que representam funcionalidades básicas do sistema, e interesses transversais, que descrevem requisitos funcionais e não funcionais. Usa-se, portanto, o conceito de separação de interesses (do inglês, separation of concerns) para descrever o sistema Telecom. Note-se que a implementação listada a seguir é uma das possíveis implementações que podem ser feitas a partir da descrição dos interesses transversais apresentada.

4.2.3.1 Descrição do sistema Telecom

O sistema Telecom simula o controle de chamadas telefônicas e foi adaptado do exemplo apresentado por The AspectJ Team (2002). O sistema permite que chamadas telefônicas de curta distância e de longa distância sejam feitas por clientes. Além disso, diferentes chamadas podem ser juntadas para que mais de dois clientes possam se comunicar, um recurso denominado teleconferência. Para tanto, clientes e chamadas são gerenciados de maneira que seja controlado o estado de uma chamada telefônica, que pode ser: pendente, completada ou finalizada. As classes que representam clientes, chamadas e conexões são descritas a seguir:

- a classe `Customer` possui dados como nome, número do telefone e código de área;
- a classe `Connection` representa a conexão entre dois clientes de uma chamada telefônica e é estendida pelas classes `Local` e `LongDistance`, que representam dois tipos diferentes de chamadas;
- a classe `Call` representa uma chamada telefônica e gerencia uma ou mais conexões entre clientes.

O sistema Telecom é é composto, além das classes-base, por três interesses transversais: interesse de temporização de chamadas telefônicas, interesse de tarifação de chamadas telefônicas e interesse de registro de chamadas ao contador de tempo. O interesse de temporização deve controlar o tempo de cada chamada, registrando o início e o término de cada conexão entre diferentes clientes, levando em consideração a utilização eventual do recurso de teleconferência. O interesse de tarifação deve contabilizar o valor de cada chamada a partir da medição de seu tempo e atribuir o valor computado ao cliente que fez a chamada, também levando em consideração o recurso de teleconferência. Chamadas de curta distância e de longa distância são tarifadas com valores diferentes. O interesse transversal de registro de chamadas ao medidor de tempo do mecanismo de temporização, deve mostrar na tela o início e o final da medição do tempo de cada conexão.

Os interesses de temporização, tarifação e registro de chamadas são interligados da seguinte forma: a temporização pode ser feita sem que, necessariamente, exista tarifação das ligações; a tarifação é dependente da temporização, uma vez que o valor das ligações é computado a partir do tempo das ligações; e, o registro de chamadas também necessita da temporização, porque ela é responsável por iniciar e finalizar a contagem de tempo. Portanto, pode-se implementar o sistema Telecom: apenas com o interesse de temporização; com o interesse de temporização e com o interesse de registro de chamadas; com o interesse de temporização e com o interesse de tarifação; e, com todos os três interesses transversais.

Os interesses funcionais de temporização de chamadas e tarifação de chamadas, e o interesse não funcional de registro de chamadas ao medidor de tempo são implementados pelos aspectos `Timing`, `Billing` e `TimerLog`, respectivamente. Além deles, uma classe auxiliar denominada `Timer` é utilizada para representar o medidor de tempo. O papel de cada um é descrito a seguir:

- a classe `Timer` é responsável por representar o medidor de tempo usado na temporização, e para isso implementa os métodos de início e fim de medição do tempo;
- o aspecto `Timing` é responsável pela temporização das chamadas, registrando o início e término de cada conexão entre clientes e medindo a duração com o auxilio da classe `Timer`;
- o aspecto `Billing` é responsável por calcular o valor por cada conexão entre dois clientes de acordo com o tipo de chamada, de longa ou curta duração, e associar o montante calculado ao cliente devedor;
- o aspecto `TimerLog` é responsável por registrar na tela chamadas de início e de término de medição de tempo.

Adicionalmente, o interesse de persistência também é acrescentado para que os clientes, juntamente com suas chamadas efetuadas, sejam mantidos em um banco de dados. Diferentemente dos interesses anteriores, esse interesse é implementado por um *framework* transversal proposto por Camargo e Masiero (2005). Portanto, como as regras de instanciação do *framework* devem ser seguidas, é apresentado apenas a implementação dos pontos de extensão em que o *framework* se acopla aos demais aspectos e classes. Além disso, para facilitar o entendimento do exemplo, a maneira de instanciar o *framework* é mostrada de forma simplificada.

4.2.3.2 Construção do AORD referente ao sistema Telecom

O AORD final da Figura 4.6 é produto do rastreamento do código fonte do sistema Telecom mostrado nas listagens de código a seguir. A construção do AORD por meio de engenharia reversa de código fonte AspectJ é feita em três passos: no primeiro, todas as classes e aspectos são mapeadas como vértices no AORD; posteriormente, o código fonte de classes e aspectos deve ser avaliado para que dependências do tipo "I" e "It" sejam rastreadas; por fim, as outras dependências são rastreadas e mapeadas no AORD.

Na Figura 4.1 é apresentado o AORD parcial com todas as classes e aspectos do sistema Telecom e com as dependências do tipo "I" ligando as classes `Local` e `LongDistance` a `Connection`. Os vértices em destaque representam a classe ou aspecto cujo código fonte foi rastreado. Na Listagem 4.1 e na Listagem 4.2 é possível observar, em ambas na linha 1, o mecanismo de conexão DH que mostra o relacionamento que dá origem a dependências do tipo "I".

Listagem 4.1: Parte do código fonte da classe `Local`.

```
1 public class Local extends Connection {
2    public Local(Customer a, Customer b) {
3       super(a, b);
4       System.out.println("[new local connection from " + a + " to " + b + "]");
5    }
6 }
```

Listagem 4.2: Parte do código fonte da classe `LongDistance`.

```
1 public class LongDistance extends Connection {
2    public LongDistance(Customer a, Customer b) {
3       super(a, b);
4       System.out.println("[new long distance connection from " + a + " to " + b + "]");
5    }
6 }
```

Na Figura 4.2 são ilustradas as dependências do tipo "I" e do tipo "It" encontradas no aspecto `MyPersistentEntities`, apresentado na Listagem 4.3. Nas linhas 2 e 3 da listagem pode-se constatar o uso do mecanismo de conexão DIAHH que dá origem a arestas

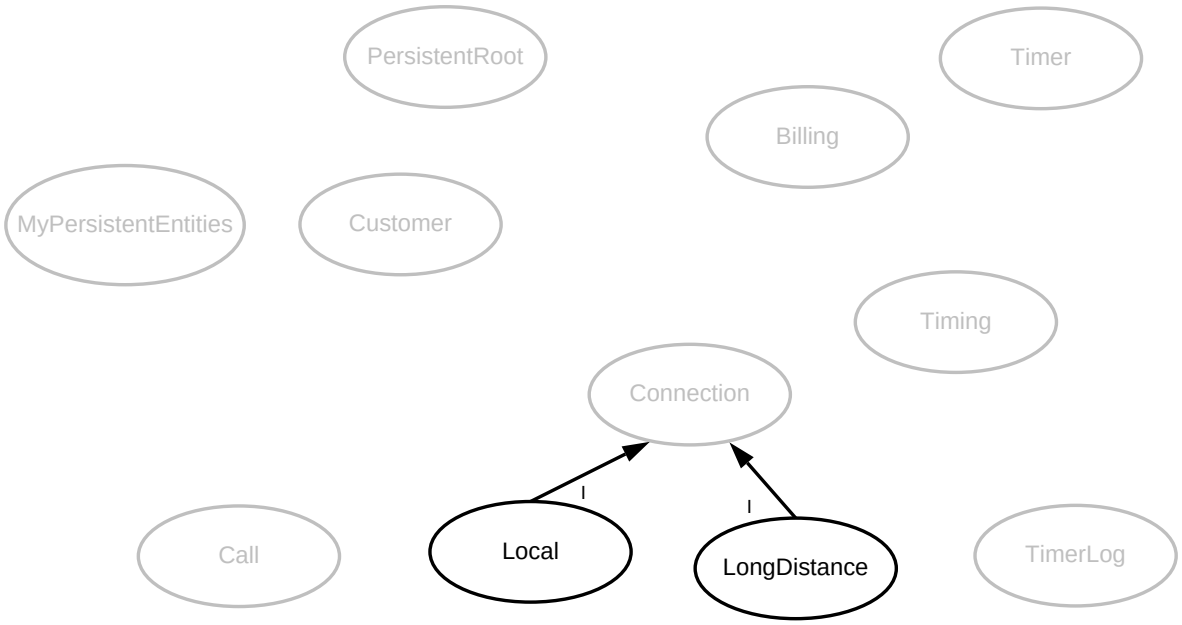


Figura 4.1: AORD parcial contendo apenas dependências do tipo "I".

do tipo "It" ligando `MyPersistentEntities` a `PersistentRoot`, a `Customer` e a `Connection`. Além dessas duas arestas o uso do mecanismo DIAHH também dá origem a uma aresta do tipo "I" ligando `Customer` a `PersistentRoot`.

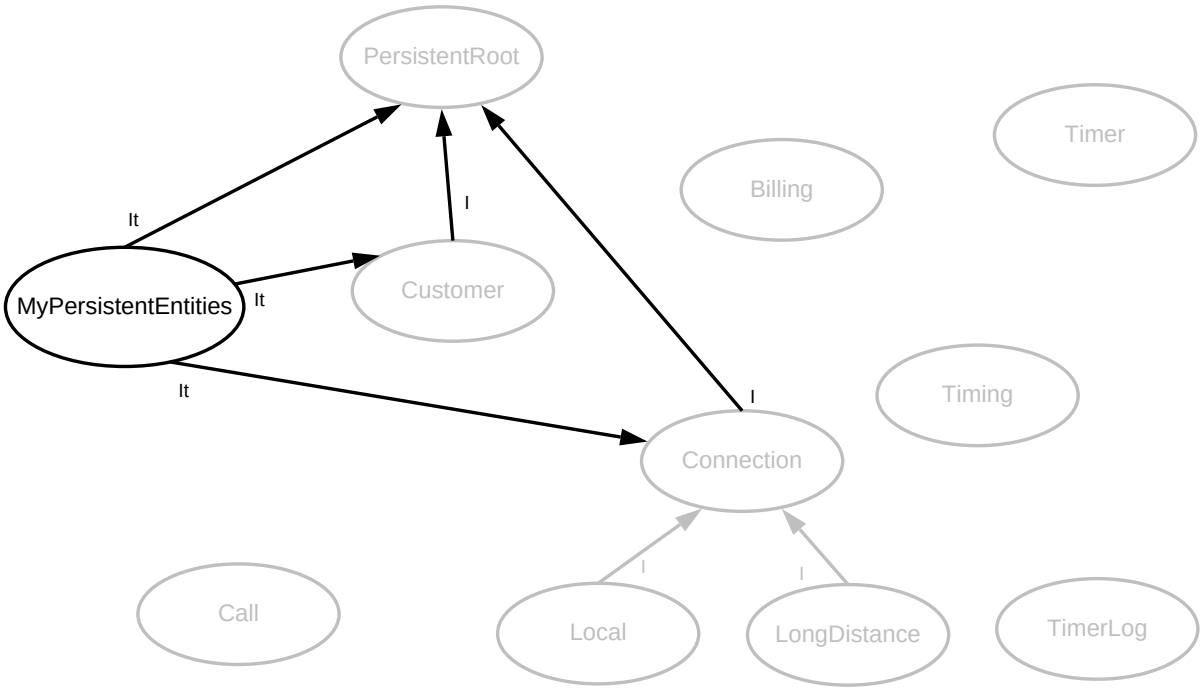


Figura 4.2: AORD parcial com a adição das dependências encontradas em `MyPersistentEntities`.

Listagem 4.3: Parte do código fonte do aspecto `MyPersistentEntities`.

```
1 public aspect MyPersistentEntities {
2    declare parents: Customer extends PersistentRoot;
3    declare parents: Connection extends PersistentRoot;
4 }
```

Na Figura 4.3 são apresentadas as dependências encontradas nos aspectos `Billing` e `Timing`, mostrados na Listagem 4.4 e na Listagem 4.5, respectivamente. As dependências de `Billing` para `Connection`, `Customer`, `LongDistance` e `Local` podem ser constatadas nas linhas 2, 3, 5 e 6 da Listagem 4.4. As dependências de `Timing` para `Customer` e `Connection` podem ser constatadas nas linhas 2 e 3 da Listagem 4.5. Conforme pode ser observado nas listagens, as dependências do tipo "It" são provenientes do uso dos mecanismos de conexão DIIA e DIIM.

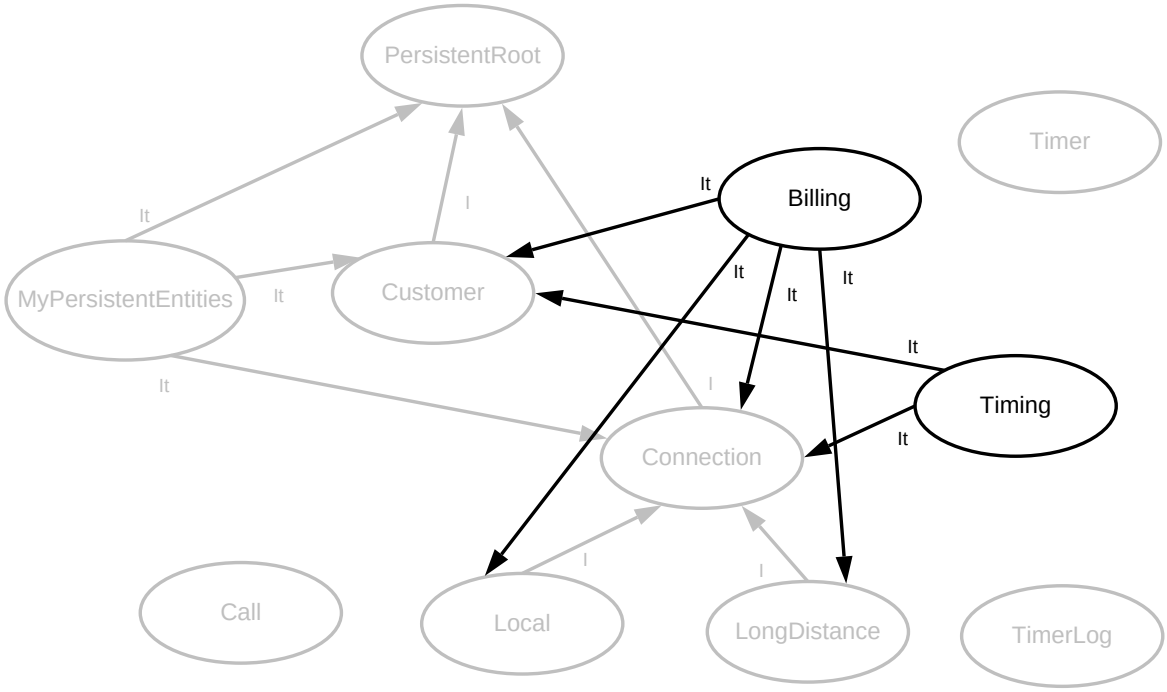


Figura 4.3: AORD parcial com a adição das dependências encontradas em `Billing` e `Timing`.

Listagem 4.4: Parte do código fonte do aspecto `Billing`.

```
1 public aspect Billing {
2    public Customer Connection.payer;
3    public long Customer.totalCharge = 0;
4    public abstract long Connection.callRate();
5    public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
6    public long Local.callRate() { return LOCAL_RATE; }
7
8    pointcut initBilling(Connection conn): target(conn) && call(void Connection.complete());
```

```
9     after(Connection conn): initBilling(conn){
10       ...
11    }
12    pointcut endBilling(Connection conn): target(conn) && call(void Connection.drop());
13    after(Connection conn): endBilling(conn) {
14       Timer t = conn.getTimer();
15       long time = t.getTime();
16       long rate = conn.callRate();
17       long cost = rate * time;
18       Customer payer = conn.getPayer();
19       payer.addCharge(cost);
20       payer.save();
21     }
22 }
```

Listagem 4.5: Parte do código fonte do aspecto `Timing`.

```
1  public aspect Timing {
2     public long Customer.totalConnectTime = 0;
3     public Timer Connection.timer = new Timer();
4     public Timer Connection.getTimer() { return timer; }
5     after (Connection conn) returning (): Billing.initBilling(conn) {
6        Timer t = conn.getTimer()
7        t.start();
8     }
9     after(Connection conn) returning () : Billing.endBilling(conn) {
10       Customer caller, receiver;
11       Timer t=conn.getTimer()
12       t.stop();
13       caller = conn.getCaller()
14       caller.setTotalConnectTime(conn.getTimer().getTime());
15       receiver = conn.getReceiver();
16       receiver.setTotalConnectTime(conn.getTimer().getTime());
17       caller.save();
18       receiver.save();
19    }
20 }
```

As dependências encontradas no terceiro passo do rastreamento do sistema Telecom são apresentadas na Figura 4.4. É interessante observar com maior atenção algumas dependências: a dependência que liga `Call` a `Customer`, rastreada pelo uso do mecanismo DM, que pode ser observado na linha 5 da Listagem 4.7; a dependência que liga `Call` a `Local`, rastreado pelo uso do mecanismo CM, que pode ser observado na linha 9 da Listagem 4.7; e a dependência que liga `Connection` a `Customer`, rastreado pelo uso do mecanismo DAVL, que pode ser observado na linha 2 da Listagem 4.8; e a dependência que liga `Connection` ao aspecto `MyPersistentEntities`, também rastreado pelo uso do mecanismo CM, que pode ser observado na linha 5 da Listagem 4.8. Apesar de existir na linha 5 uma chamada ao método `Connection.save()` ele não é implementado na classe `Connection`, mas sim na classe `PersistentRoot`. O método é herdado por `Connection` por meio de uma declaração

de alteração de hierarquia de herança feita por `MyPersistentEntities`, que faz com que `Connection` seja uma especialização de `PersistentRoot`.

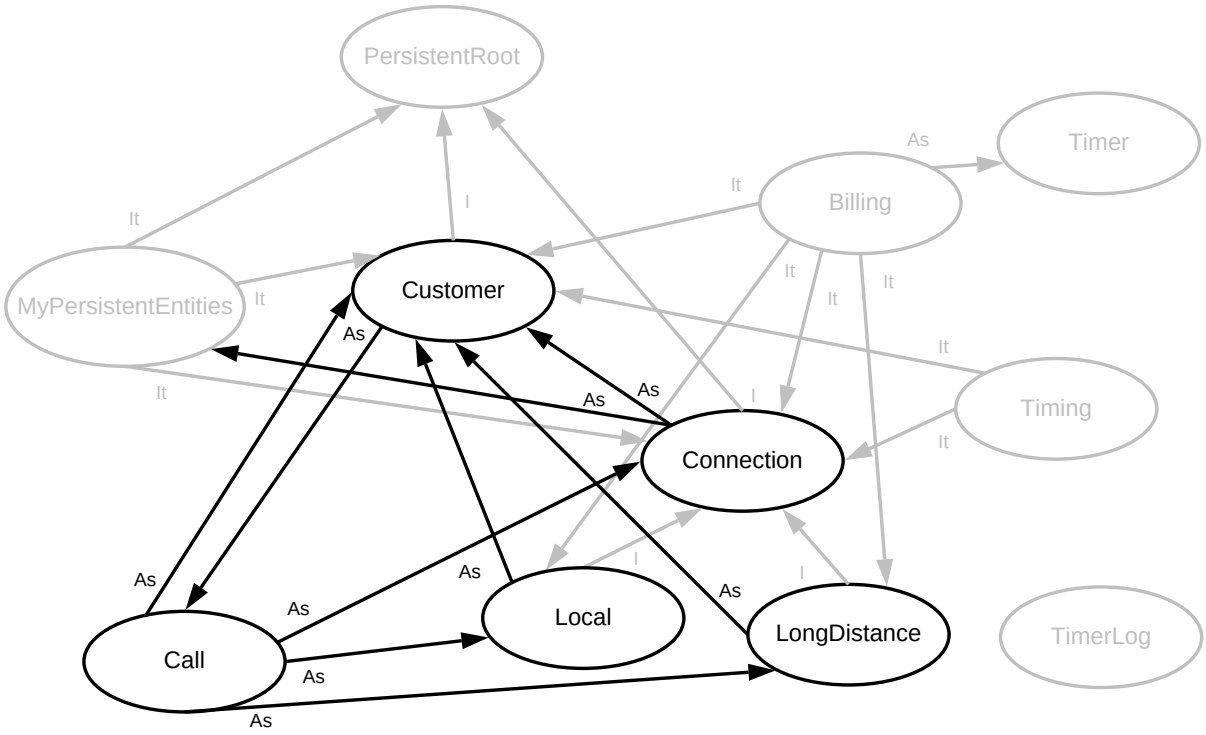


Figura 4.4: AORD parcial com a adição das dependências encontradas nas classes do sistema Telecom.

Listagem 4.6: Parte do código fonte da classe `PersistentRoot`.

```
1 public class PersistentRoot {
2    public void save() { ... }
3 }
```

Listagem 4.7: Parte do código fonte da classe `Call`.

```
1  public class Call {
2     private Customer caller, receiver;
3     private Vector connections = new Vector();
4
5     public Call(Customer caller, Customer receiver) {
6        this.caller = caller;
7        this.receiver = receiver;
8        if (receiver.localTo(caller)) {
9           connections.addElement(new Local(caller, receiver));
10       } else {
11          connections.addElement(new LongDistance(caller, receiver));
12       }
13    }
14    public void pickup() {
15       Connection connection = (Connection) connections.lastElement();
```

```
16        connection.complete();
17    }
18 }
```

Listagem 4.8: Parte do código fonte da classe `Connection`.

```
1  public abstract class Connection {
2     Customer caller, receiver;
3     public void drop() {
4        state = DROPPED;
5        this.save();
6     }
7  }
```

Listagem 4.9: Parte do código fonte da classe `Customer`.

```
1  public class Customer {
2     private String name;
3     private int areacode;
4     private Vector theCalls = new Vector();
5
6     protected void addCall(Call c){ theCalls.addElement(c); }
7     public boolean localTo(Customer other){ return areacode == other.areacode; }
8  }
```

Listagem 4.10: Parte do código fonte da classe `Timer`.

```
1  public class Timer {
2     public void start() { ... }
3     public void stop() { ... }
4     public long getTime() { ... }
5  }
```

As dependências provenientes apenas dos aspectos são mostradas na Figura 4.5. A dependência que liga `Billing` a `Timing` é rastreada pelo uso do mecanismo CM, que pode ser notado na linha 14 da Listagem 4.4. Nela, existe a chamada ao método `Connection.getTimer()`, porém o método não é implementado em `Connection`, é implementado e introduzido por `Timing`, como pode ser visto na linha 4 da Listagem 4.5. As dependências que ligam os aspectos `Billing` e `Timing` ao aspecto `MyPersistentEntities` também devem ser analisadas com mais atenção. Note-se que na linha 20 da Listagem 4.4 e na linha 17 da Listagem 4.5 existem chamadas ao método `Customer.save()`, porém o método não está implementado na classe `Customer` mas sim na classe `PersistentRoot`. O método é herdado por `Customer` por meio de uma declaração de alteração de hierarquia de herança feita por `MyPersistentEntities`, que faz com que `Customer` seja uma especialização de `PersistentRoot`.

Outras dependências interessantes são as que ligam `Billing` a `Call`, `Timing` a `Billing` e `TimerLog` a `Timing`. A dependência do tipo “C” entre `Billing` e `Call` é

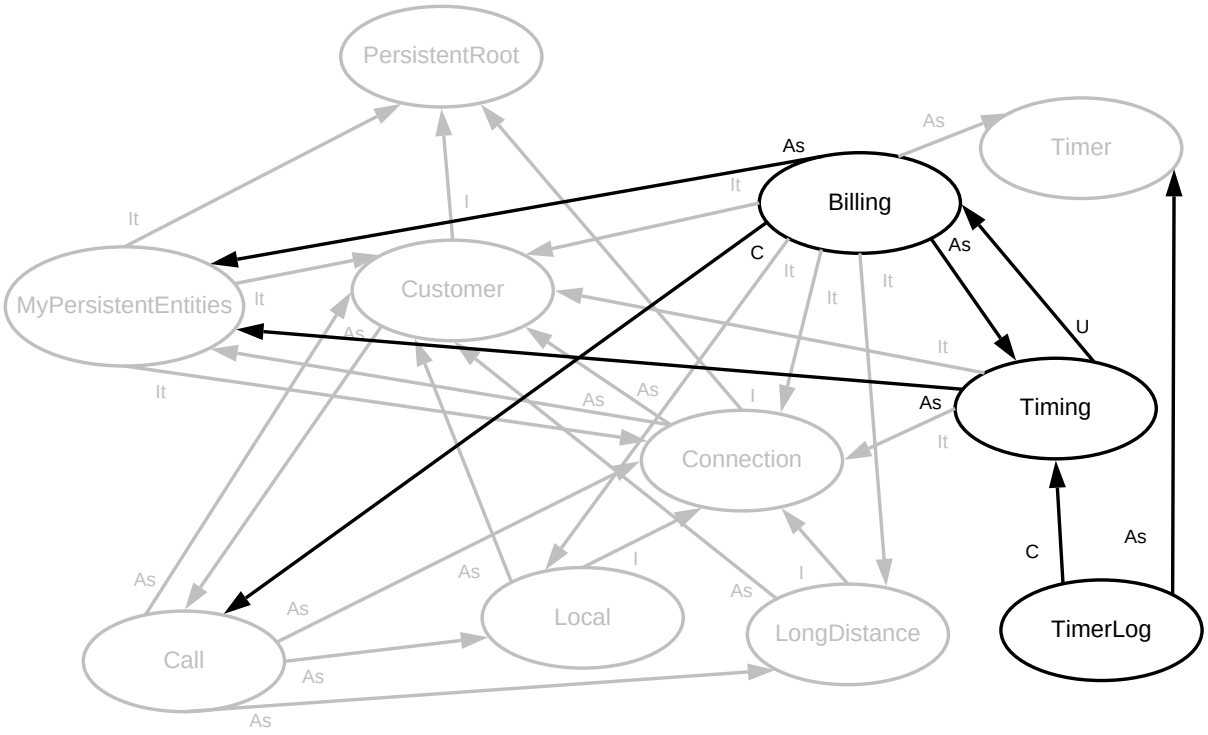


Figura 4.5: AORD parcial com a adição das dependências encontradas nos aspectos do sistema Telecom.

rastreada pelo uso do mecanismo CJ, que pode ser observado na linha 8 da Listagem 4.4. Para constatar a dependência é necessário encontrar o ponto de junção capturado pelo conjunto de junção que, como pode ser visto na linha 16 da Listagem 4.7, se encontra na classe `Call`, em que existe uma chamada ao método `Connection.Complete()`. A dependência do tipo "U" que liga os aspectos `Timing` e `Billing` é rastreada por meio do uso do mecanismo DA, que pode ser visto nas linhas 5 e 9 da Listagem 4.5. Por fim, a dependência que liga os aspectos `TimerLog` e `Timing` é rastreada pelo uso do mecanismo CJ, que pode ser visto na linha 2 da Listagem 4.11. Novamente, o ponto de junção deve ser encontrado. Neste caso, ele ocorre em um adendo do aspecto `Timing` que faz uma chamada ao método `Timer.start()`. Isso pode ser visto na linha 7 da Listagem 4.5.

Listagem 4.11: Parte do código fonte do aspecto `TimerLog`.

```
1 public aspect TimerLog {
2    after(Timer t) returning () : target(t) && call(* Timer.start()){
3       System.out.println("Timer started: " + t.startTime);
4    }
5 }
```

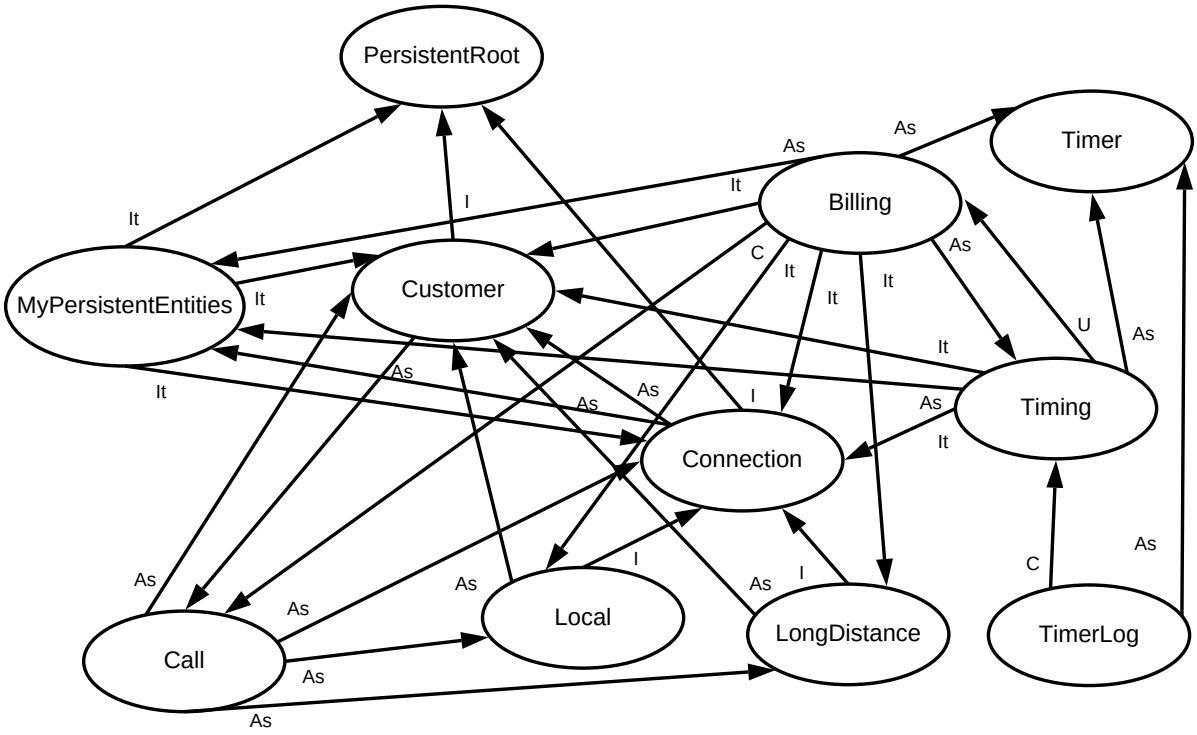


Figura 4.6: AORD final com todas as dependências encontradas no sistema Telecom.

4.3 Proposta de duas estratégias de ordenação

As estratégias apresentadas neste trabalho estendem a proposta de Briand *et al.* (2003) para ordenar aspectos e classes usando o AORD. Dessa forma, um AORD $G = (V, L, E)$ cíclico, pode ser transformado em um dígrafo acíclico $G' = (V', L', E')$, de forma que V' seja o conjunto de SCCs pertencentes a G, E' seja o conjunto de arestas entre os SCCs de G e L' seja o conjunto de rótulos das arestas de E'. Nesse caso, a ordenação topológica reversa pode ser aplicada de maneira direta. Um SCC com mais de uma classe ou aspecto ainda possui ciclos de dependência que devem ser quebrados. A ideia é remover temporariamente algumas arestas, o que resulta na implementação de *stubs* para quebrar o acoplamento e tornar o dígrafo acíclico.

4.3.1 Estratégia Incremental+

A primeira estratégia proposta é a Incremental+. Nessa estratégia primeiramente as classes são testadas e integradas e posteriormente os aspectos são testados e integrados. Conforme discutido, os trabalhos que tratam do teste de integração não são detalhistas a ponto de propor uma ordem que minimize o esforço e contorne os problemas encontrados durante essa atividade. A proposta difere das demais porque:

- Usa a ordenação topológica reversa para determinar a ordem de teste de classes e aspectos separadamente em caso de inexistência de ciclos de dependência;
- Usa o AORD para determinar a ordem de teste e integração entre classes em presença de ciclos;
- Usa o AORD para determinar a ordem de teste e integração apenas entre os aspectos em presença de ciclos de dependência;
- Minimiza, separadamente, o o número de *stubs* de classes e o número de *stubs* de aspectos durante o teste de integração.

Na estratégia Incremental+, 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 listados na Tabela 4.2, enquanto que para construir o segundo são usados todos os mecanismos de conexão. O algoritmo de Briand *et al.* (2003) é então aplicado ao primeiro AORD e a ordem de teste e integração de classes é determinada. Posteriormente, algoritmo de Briand *et al.* (2003) é aplicado ao segundo AORD e a ordem de teste e integração entre os aspectos é determinada. Note-se que, nessa estratégia, que segue o princípio da estratégia incremental otimizada pela aplicação do algoritmo de Briand *et al.* (2003), 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.

No AORD apresentado na Figura 4.7, constituído apenas de classes, pode-se observar que existem classe isoladas, que não se relacionam com outras porque elas são utilizadas apenas pelos aspectos, e um SCC formado pelas classes: `Local`, `Connection`, `LongDistance`, `Customer` e `Call`. Na prática, as classes isoladas podem ser testadas antes ou depois das classes que fazem parte do SCC.

O algoritmo de Briand *et al.* (2003), quando aplicado ao AORD indica que a aresta que liga `Customer` a `Call` deve ser removida e, consequentemente, um *stub* de `Call` deve ser implementado para o teste de `Customer`. Portanto, a ordem de teste computada é mostrada na Tabela 4.3. Note-se que ainda é necessário implementar um *stub* para simular o comportamento do aspecto `MyPersistentEntities` para testar `Connection`, uma vez que a dependência que liga a classe ao aspecto não foi considerada na construção do AORD somente com aspectos.

De acordo com AORD somente com aspectos mostrado na Figura 4.8 a ordem de implementação e teste dos aspectos do sistema Telecom pode ser visto na Tabela 4.4. Existe um ciclo

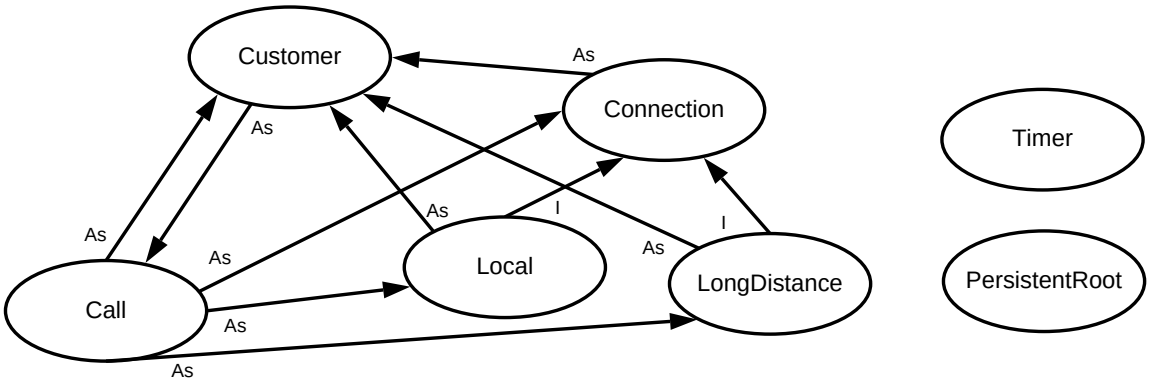


Figura 4.7: AORD somente com as classes do sistema Telecom.

Tabela 4.3: Ordem de implementação e teste das classes do sistema Telecom.

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

de dependência entre os aspectos `Timing` e `Billing` e o cálculo usado pelo algoritmo de Briand *et al.* (2003) vai computar o mesmo peso para as arestas que ligam os dois. Assim, o algoritmo escolhe para a remoção uma das duas arestas de maneira aleatória. Para efetuar o teste do aspecto `Timing` é necessário um *stub* de `Billing`. O custo final da ordenação na estratégia Incremental+ é de três *stubs*, dois de aspectos e um de classe, mas ignora o problema eventual de precisar de *stubs* de aspectos para implementar classes.

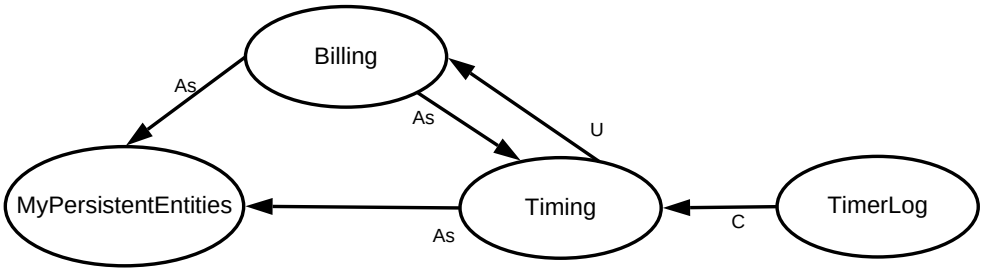


Figura 4.8: AORD somente com os aspectos do sistema Telecom.

Tabela 4.4: Ordem de implementação e teste dos aspectos do sistema Telecom.

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

4.3.2 Estratégia Conjunta

A estratégia Conjunta não divide o AORD em dois subgrafos e, ao contrário da estratégia Incremental+, considera todas as dependências representadas. Para tanto, a construção do AORD considera todos os mecanismos de conexão da Tabela 4.2. Isso faz com que classe e aspectos sejam testados e integrados conjuntamente, de acordo com a ordem determinada pela estratégia.

Na estratégia Conjunta, o AORD é construído e o algoritmo de Briand *et al.* (2003) é aplicado, resultando na ordem de teste e integração de classes e aspectos. A estratégia conjunta é diferente da estratégia incremental e da estratégia Incremental+ porque:

- Usa a ordenação topológica reversa para determinar a ordem de teste de classes e aspectos conjuntamente em caso de inexistência de ciclos de dependência;
- Usa o AORD para determinar a ordem de teste e integração entre os aspectos e classes em presença de ciclos de dependência;
- Minimiza o número de *stubs* de classes e aspectos durante o teste de integração.

No AORD com classes e aspectos são encontrados dois SCCs: o primeiro, mostrado na Figura 4.9, chamado de SCC_1 envolve `MyPersistentEntities`, `Local`, `Connection`, `LongDistance`, `Customer` e `Call`; o segundo, chamado de SCC_2, envolve `Timing` e `Billing`. A ordem de teste parcial calculada é: `PersistentRoot`, SCC_1={`MyPersistentEntities`, `Local`, `Connection`, `LongDistance`, `Customer`, `Call`}, `Timer`, SCC_2={`Timing`, `Billing`} e `TimerLog`. O algoritmo indica a remoção da aresta que liga `Customer` a `Call` para quebrar os ciclos de SCC_1 e, a aresta que liga `Timing` a `Billing` para quebrar os ciclos no SCC_2.

Ainda resta um SCC, chamado de $SCC_{1.1}$, envolvendo `MyPersistentEntities` e `Customer`, resultando na seguinte ordem parcial de teste: `PersistentRoot`, `Customer`, $SCC_{1.1}$={`MyPersistentEntities`, `Connection`}, `Local`, `LongDistance`, `Call`, `Timer`, `Timing`, `Billing` e `TimerLog`. Portanto, o algoritmo indica, aleatoriamente, a remoção da aresta que liga `Connection` a `MyPersistentEntities`, resultando na ordem

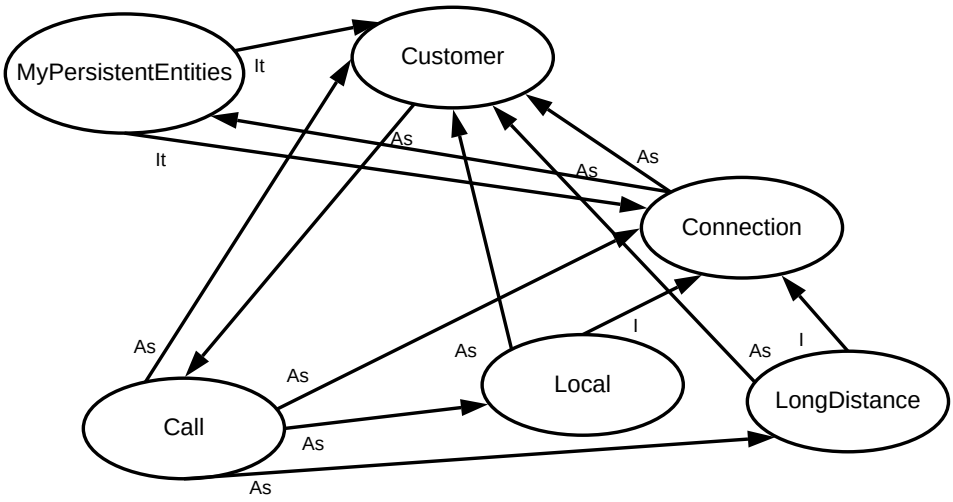


Figura 4.9: SCC_1 com classes e aspectos do sistema Telecom.

de teste final mostrada na Tabela 4.5. Por fim, conforme pode ser notado, o custo de implementação da estratégia Conjunta é o mesmo da estratégia Incremental+, isto é, um *stub* de classe e dois *stubs* de aspectos.

Tabela 4.5: Ordem de implementação e teste dos aspectos do sistema Telecom.

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`	

4.4 Considerações Finais

Conforme apresentado neste capítulo, as duas estratégias de ordenação utilizam o algoritmo de Briand *et al.* (2003) de maneira direta, graças ao AORD proposto. O AORD pode modelar de maneira completa as dependências da programação OA graças ao modelo de dependências aspectuais também proposto, além de dependências OO, já incluídas no ORD original. Na

estratégia Incremental+ devem ser construídos dois AORDs, um apenas com classes e outro apenas com aspectos. A construção de dois AORDs é uma maneira de quebrar os ciclos de dependência entre aspectos e classes sem que sejam levados em consideração os demais ciclos que o AORD eventualmente possua. No caso do exemplo utilizado para mostrar como as estratégias funcionam, a divisão entre classes e aspectos foi a mesma encontrada pela aplicação do algoritmo na estratégia Conjunta. Esse fato pode ser constatado ao se observar que foram necessários os mesmos *stubs* na aplicação das duas estratégias. Estudos exploratórios entre a abordagem usada pela estratégia Incremental+ para a quebra de ciclos entre classes e aspectos e a abordagem da estratégia Conjunta, que considera conjuntamente todas os ciclos de dependência, visando à comparação da eficiência e comportamento das duas estratégias ainda é um ponto em aberto que deve ser explorado no Capítulo 6.

No exemplo de aplicação das estratégias apresentado, percebe-se que é possível existir diferentes combinações de dependências entre classes e aspectos, conforme indicado na literatura especializada. Como consequência, *stubs* com características diferentes daqueles encontrados apenas na programação OO devem ser implementados. Isso acontece por causa dos novos mecanismos de conexão da programação OA. Por exemplo, alguns mecanismos de conexão dão origem a mais de um tipo de dependência, o que não acontece na programação OO. Outro ponto interessante é que os mecanismos de conexão podem ser usados em conjunto, como por exemplo, quando um adendo utiliza um método introduzido ou quando um conjunto de junção com o comando condicional `If` invoca um método herdado via alteração de hierarquia de herança. Ainda existem mecanismos que conectam apenas aspectos, o que leva à implementação de *stubs* de aspectos, caso a dependência seja removida em qualquer uma das estratégias. Portanto, pode-se concluir que os *stubs* são implementados de acordo com os mecanismos de conexão e, consequentemente, com os relacionamentos existentes na linguagem de programação OA.

O modelo de dependências aspectuais foi produzido para ser utilizado na construção do AORD. Assim, ele indica como ocorrem as dependências para que, quando um algoritmo de ordenação seja aplicado, o AORD permaneça consistente e o número de arestas mapeadas seja otimizado. Isso porque, quanto menos arestas desnecessárias são mapeadas no AORD menor é a ocorrência de ciclos. Além disso, os *stubs* esperados, computados a partir da ordem de teste produzida, e os realmente implementados devem ser, idealmente, os mesmos. Essa é uma propriedade que deve ser respeitada por qualquer extensão ao ORD original, caso contrário, se são implementados mais *stubs* do que o esperado, então o AORD não modela corretamente todas dependências. Em contrapartida, se são implementados menos *stubs* do que o esperado, então o AORD modela dependências desnecessárias.

Adicionalmente, o modelo de dependências aspectuais indica a existência de novas formas de acoplamento entre aspectos e entre aspectos e classes. Conforme discutido, existem poucas métricas de acoplamento para a programação OA. Além disso, não existem estudos para validar as poucas métricas propostas. Considerando a proposta de Briand *et al.* (1999), o modelo de dependências aspectuais é um artefato interessante para o desenvolvimento de métricas de acoplamento porque atende aos requisitos estabelecidos em seu arcabouço (*framework*) de métricas de acoplamento. Contudo, o desenvolvimento de métricas de acoplamento foge do escopo deste trabalho. O AORD proposto também pode ser usado para condução de testes de regressão, conforme descrito por Kung *et al.* (1996b). O modelo de dependências aspectuais mostra os possíveis pontos afetados em eventuais alterações promovidas no código fonte. Com ele é possível identificar o conjunto de classes e aspectos que devem ser retestados prioritariamente, chamado por Kung *et al.* (1996b) de *class firewall*.

Não obstante o modelo de dependências aspectuais seja completo, no sentido de que pode mapear qualquer dependência gerada por mecanismos de conexão da linguagem AspectJ, ele possui algumas restrições. Note-se que algumas dependências referentes a conjuntos de junção que usam informações disponíveis em tempo de execução não podem ser rastreadas pela análise do código fonte, apenas executando o código. Essa não é uma limitação do modelo, mas uma restrição imposta pela própria linguagem de programação. Assim, o AORD não é tão preciso na presença de conjuntos de junção desse tipo. Isto é, como ele deve ser idealmente construído na fase de projeto, essas dependências poderão surgir na fase de implementação.

CAPÍTULO 5

Como criar um AORD na fase de projeto

5.1 Considerações iniciais

O objetivo deste capítulo é mostrar como usar o que foi proposto no Capítulo 4. Na prática é importante a aplicação de uma estratégia de ordenação no fim da fase de projeto e antes do início da fase de implementação. Assim a implementação, o teste de unidade e o subsequente teste de integração segue a ordenação estabelecida pela estratégia adotada. Portanto, a construção do AORD proposto deve ser feita também nesse momento do desenvolvimento do software utilizando modelos de projeto disponíveis.

Um processo de mapeamento que usa um modelo de projeto baseado em interesses para a construção de um AORD é proposto nesse capítulo. O processo de mapeamento é importante porque promove o mapeamento automatizado, ou semiautomatizado. O mapeamento é feito primeiramente a partir de diagramas da UML e posteriormente usando os diagramas da notação MATA. Para exemplificar a utilização do processo é utilizado como exemplo a descrição do sistema Telecom, mostrada no Capítulo 4. A modelagem do sistema Telecom usado como exemplo é feita a partir da descrição do problema e não utiliza a implementação. A estratégia Conjunta é, então, aplicada ao AORD produzido para mostrar como o AORD apoia a utilização das estratégias de ordenação propostas. Além disso, o sistema é modelado baseado na separação de interesses. Conforme descrito na Seção 3.4, várias são as propostas de modelagem de sistemas orientados a aspectos, porém não existe uma notação ou técnica de modelagem amplamente utilizada e aceita como padrão pelas comunidades científica e profissional. Assim, escolheu-se a notação MATA (Whittle e Jayaraman, 2007) para modelar o sistema Telecom, por ser mais recente, preservar a notação UML e ter uma ferramenta de apoio, que em tese, poderia permitir a automação do processo proposto neste capítulo.

Este capítulo está organizado como segue. Na Seção 5.2 é apresentada a modelagem do sistema Telecom usando a notação MATA. Na Seção 5.3 é apresentado em detalhes um processo para mapear um projeto realizado com artefatos construídos com as notações UML e MATA para um AORD. A aplicação da estratégia Conjunta no AORD produzido a partir dos artefatos de projeto é mostrada em detalhes na Seção 5.4. As restrições encontradas durante o mapeamento são descritas na Seção 5.5. Por fim, na Seção 5.6 são apresentadas as considerações finais.

5.2 Modelagem baseada em separação de interesses do sistema Telecom

A modelagem do sistema Telecom foi feita em duas etapas: primeiramente identificando e separando-se os interesses-base da aplicação e depois os interesses transversais. Em seguida, cada interesse foi modelado. O modelo base usa a notação UML normal (OMG, 2007) e os modelos transversais usam a notação das regras de transformação de MATA (Whittle e Jayaraman, 2007).

5.2.1 Modelo base

Para a confecção do modelo foi usado um processo baseado no RUP (*Rational Unified Process*) (Jacobson *et al.*, 1999; Larman, 2002), mas como se trata de assunto bastante conhecido e o exemplo é simples, optou-se por não detalhar este processo e apresentar apenas o resultado final, na Figura 5.1. Nota-se que o modelo é composto de três classes: `Customer`, `Connection` e `Call`. `Connection`, por sua vez, possui duas subclasses: `Local` e `LongDistance`.

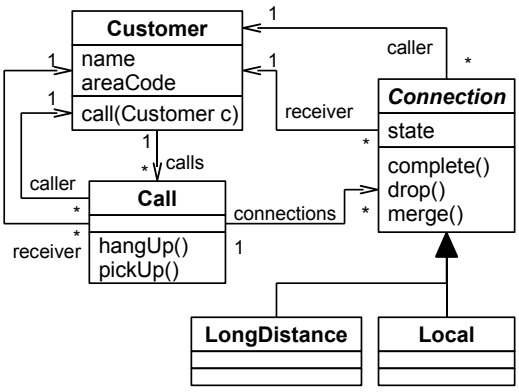
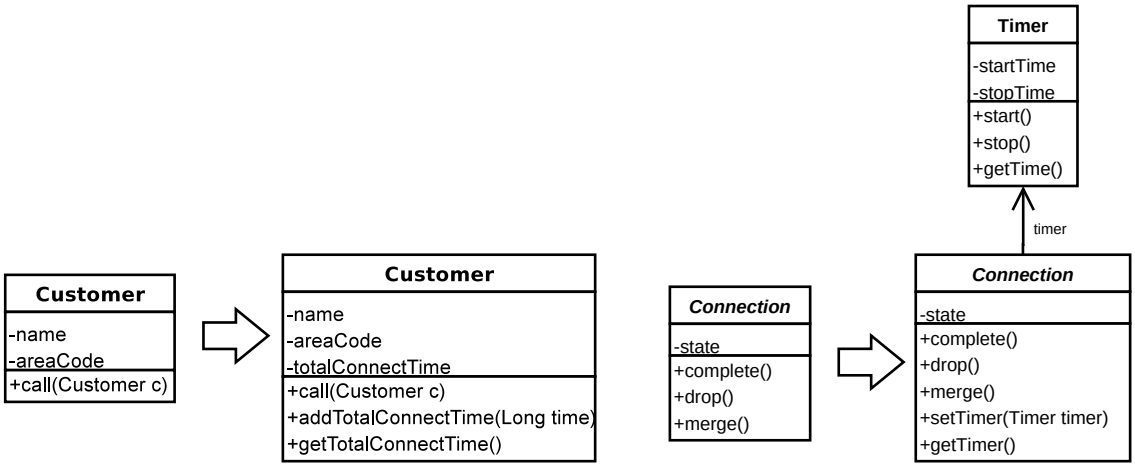


Figura 5.1: Diagrama de classes do sistema Telecom apenas com as classes-base.

5.2.2 Interesse de temporização

Para que a temporização seja feita, é necessário contabilizar o tempo total que um cliente fica conectado segundo um medidor de tempo que deve ser associado a cada conexão. Para o cliente, é importante apenas contabilizar o total de tempo de todas as conexões estabelecidas. Assim, a regra de transformação R1, que adiciona um atributo com o tempo total que cada cliente permaneceu conectado é apresentada na Figura 5.2(a). Além do atributo, são adicionados pela regra, um método para obter o valor do atributo com o tempo total e um método para adicionar ao tempo total já registrado o tempo de novas chamadas telefônicas. Na Figura 5.2(b) é mostrada a regra de transformação R2, que adiciona um atributo do tipo `Timer` na classe `Connection` para manter registrado na própria conexão seu tempo de duração. A adição é representada pelo relacionamento de associação entre as classes `Connection` e `Timer`. Ainda pela mesma regra, são adicionados métodos para definir e para obter qual é a instância da classe `Timer` associada à classe `Connection`.



(a) Regra R1 de adição de métodos e atributos na classe `Customer`.

(b) Regra R2 de adição de métodos e atributos na classe `Connection`.

Figura 5.2: Introdução de atributos e métodos necessários para o interesse de temporização.

A contagem do tempo da conexão de uma chamada telefônica é iniciada quando uma conexão entre dois clientes é estabelecida. A regra de transformação R3, contida na Figura 5.3, mostra que chamadas ao método `Connection.complete()`, feitas por qualquer método da classe `Call`, devem ser interceptadas e, após sua execução, o medidor de tempo deve ser iniciado.

O término da temporização acontece quando o método `Connection.drop()` é chamado por qualquer método da classe `Call`, conforme mostra a regra R4 da Figura 5.4. Segundo a regra de transformação R4, neste ponto de execução do sistema o temporizador encerra a

contagem de tempo, incrementa o tempo de conexão de cada cliente e persiste os dados dos clientes.

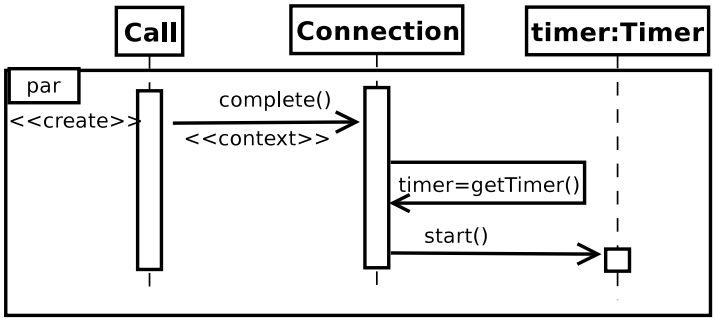


Figura 5.3: Regra R3 determina o conjunto de junção e o adendo para efetuar o início da temporização de uma chamada telefônica.

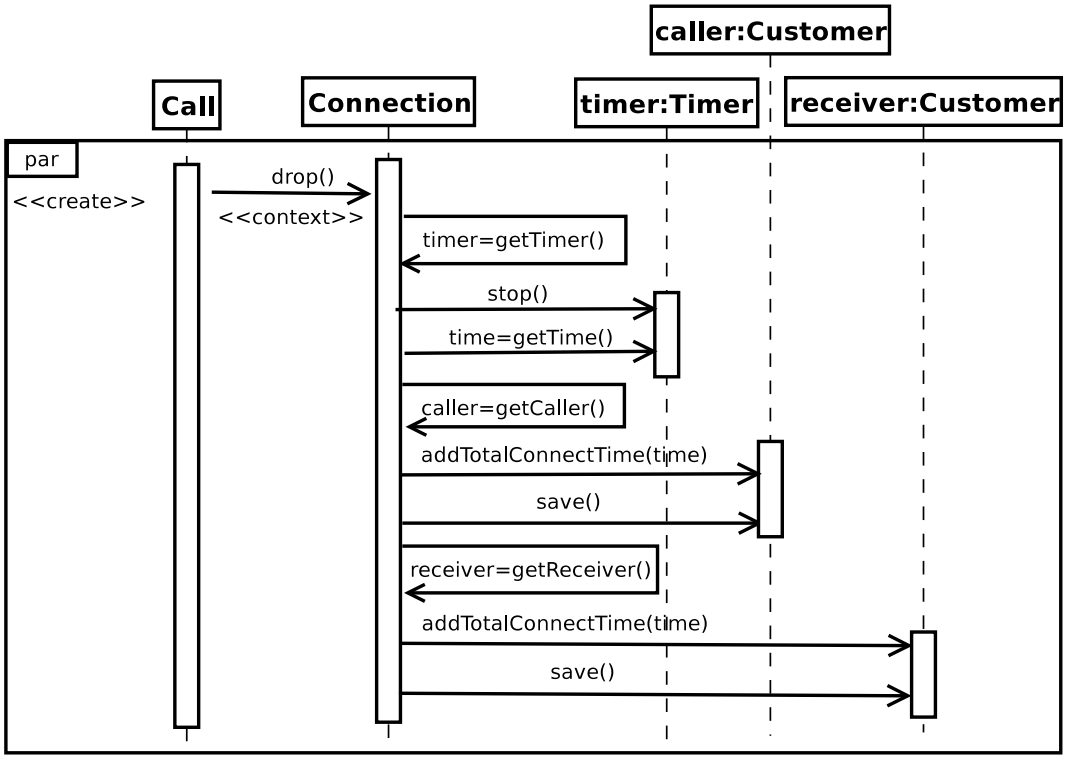
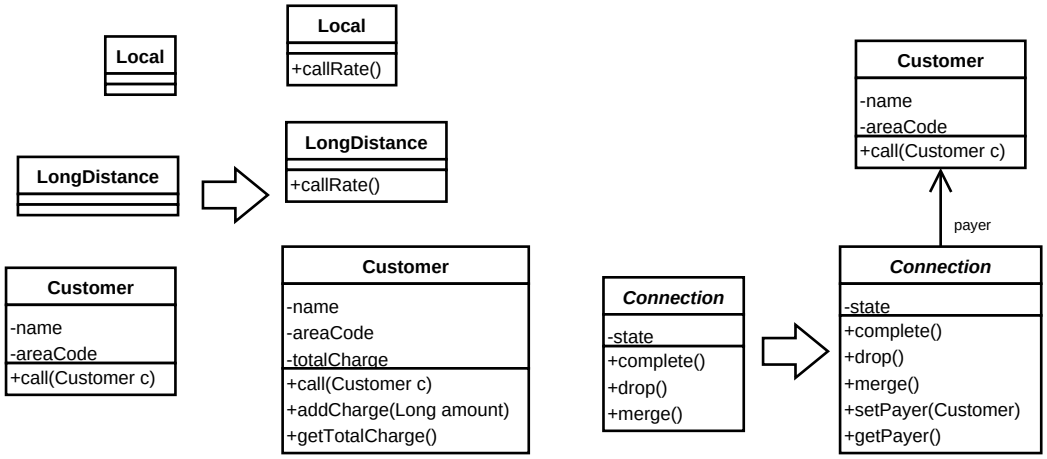


Figura 5.4: Regra R4 determina o conjunto de junção e o adendo para efetuar o término da temporização de uma chamada telefônica.

5.2.3 Interesse de tarifação

Para que o interesse de tarifação calcule e atribua o montante devido ao cliente correto, são adicionados atributos responsáveis por manipular e armazenar o valor das chamadas telefônicas

na classe `Customer`. O cálculo é feito de acordo com o tipo da chamada, local ou de longa distância, conforme pode-se observar na regra R5 ilustrada na Figura 5.5(a). A regra R5 também mostra a adição de um método para somar ao valor total que um cliente deve pagar o valor de uma nova ligação, e um método pra obter esse valor total. Além disso, a conexão que originou a tarifa também deve referenciar o cliente correto, o que é feito pela adição de um atributo do tipo `Customer` na classe `Connection`, como pode ser observado na regra R6 mostrada na Figura 5.5(b). Métodos para definir e obter a instância da classe `Customer` associada à classe `Connection` também são adicionados pela regra R6.



(a) Regra R5 de adição de métodos e atributos na classes `Local`, `LongDistance` e `Customer`.

(b) Regra R6 de adição de métodos e atributos na classe `Connection`.

Figura 5.5: Introdução de atributos e métodos necessários para o interesse de tarifação.

A tarifação é iniciada simultâneamente ao inicio da temporização, quando a chamada telefônica é estabelecida entre dois ou mais clientes pela invocação do método `Connection.complete()` por qualquer método da classe `Call`. Esse comportamento é descrito pela regra R7 mostrada na Figura 5.6. Nesse ponto, o cliente responsável pelo pagamento da tarifa é capturado e associado à conexão correta.

Da mesma forma que o final da temporização, o final da tarifação ocorre quando o método `Connection.drop()` é invocado, conforme pode-se observar na regra R8 da Figura 5.7. O interesse de tarifação usa o tempo medido por uma instância da classe `Timer`, segundo o interesse de temporização, para calcular o valor da tarifa. O valor é então faturado para o cliente associado à conexão pela regra R7, no início da tarifação. Por fim, os dados do cliente são persistidos.

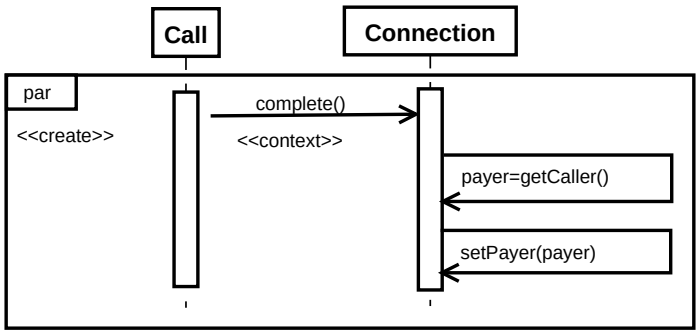


Figura 5.6: Regra R7 determina o conjunto de junção e o adendo para efetuar o início da tarifação de uma chamada telefônica.

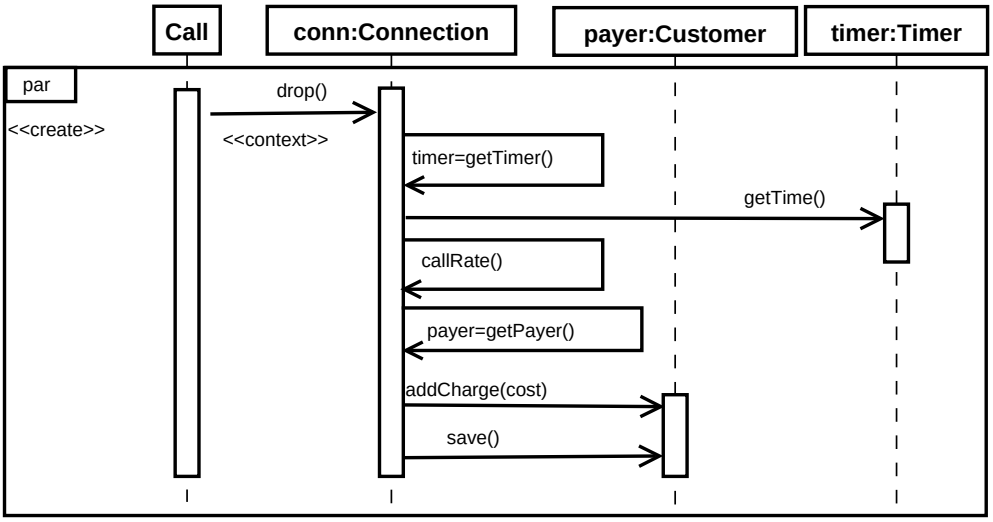


Figura 5.7: Regra R8 determina o conjunto de junção e o adendo para efetuar o término da tarifação de uma chamada telefônica.

5.2.4 Interesse de registro de chamadas ao medidor de tempo

Na Figura 5.8 e na Figura 5.9 são mostradas as regras de transformação para que chamadas aos métodos que iniciam e param o medidor de tempo sejam registradas e exibidas na tela. Note-se que as regras foram projetadas para capturar invocações aos métodos do medidor de tempo, não importando sua origem. A regra R9 e a regra R10 têm como objetivo registrar quando o interesse de temporização inicia e finaliza a contagem de tempo por meio do medidor de tempo.

Assim, após a aplicação da regra, pode-se determinar que os pontos de junção capturados estão no interesse de temporização, o que pode ser observado pelas chamadas aos métodos `Timer.start()` e `Timer.stop()` ilustradas na Figura 5.3 e na Figura 5.4.

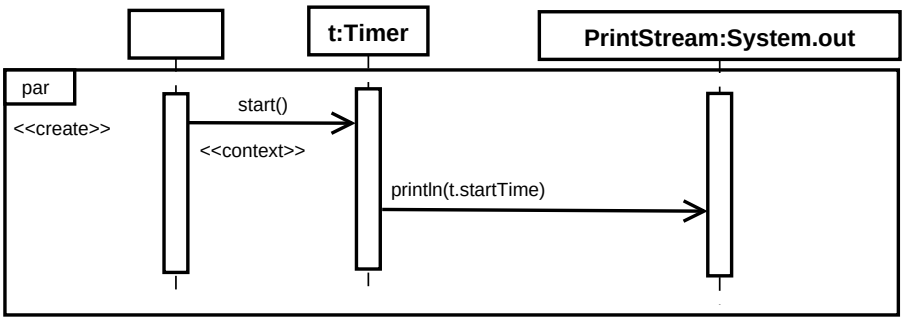


Figura 5.8: Regra R9 determina o conjunto de junção e o adendo para efetuar o registro de chamadas ao método `Timer.start()`.

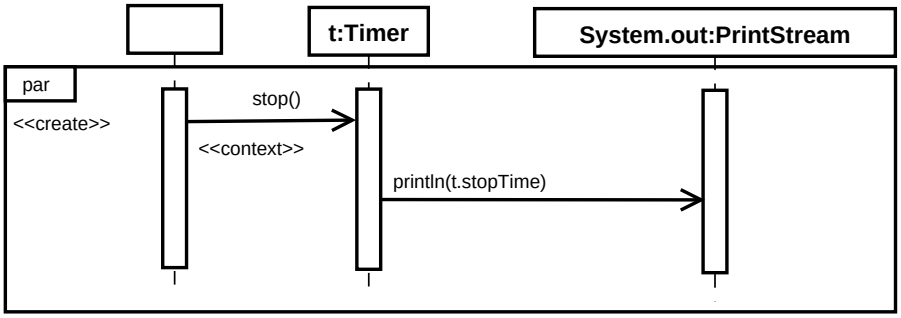


Figura 5.9: Regra R10 determina o conjunto de junção e o adendo para efetuar o registro de chamadas ao método `Timer.stop()`.

5.2.5 Interesse de persistência

Conforme descrito na Seção 4.2.3.1, sabe-se que existe um *framework* de persistência implementado que deve ser acoplado ao sistema Telecom, o que torna a descrição do interesse de persistência diferente das demais. O interesse de persistência é implementado por aspectos, enquanto que os demais interesses podem ser implementados de diferentes maneiras. Além disso, mostra-se aqui apenas aspectos e classes necessários à instanciação do *framework*. Portanto, a modelagem acompanha a implementação do *framework* para descrever como o interesse de persistência é acoplado ao projeto do sistema Telecom.

Segundo as regras de instanciação do *framework*, o interesse de persistência é acoplado por meio da classe `PersistentRoot` e pelo aspecto `MyPersistentEntities`. A classe `PersistentRoot` possui métodos responsáveis por diferentes tarefas de persistência. O aspecto `MyPersistentEntities` é responsável por definir quais classes devem ser persistidas e, por isso, implementa as alterações de hierarquia de herança da regra de transformação R11, ilustrada na Figura 5.10.

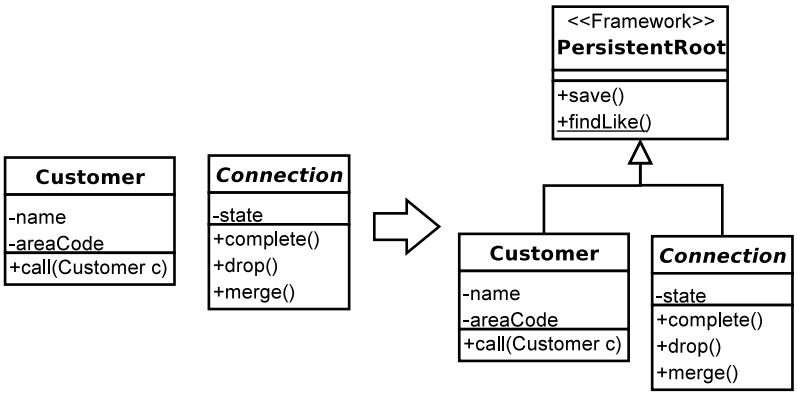


Figura 5.10: Regra R11 de alteração de hierarquia de herança das classes `Customer` e `Connection`.

5.3 O processo de mapeamento de um modelo de projeto baseado em separação de interesses para um AORD

O primeiro passo para utilizar uma das estratégias de ordenação propostas é a construção de um AORD. Apesar dos tipos de dependências que podem ser modeladas no AORD terem sido formulados a partir de código fonte AspectJ, a construção do AORD deve ser feita, idealmente, a partir de um modelo de projeto que descreve o sistema a ser implementado e testado. Muito embora existam várias propostas que utilizam o ORD, ou extensões dele, conforme descrito na Seção 2.5, como modelo que representa as dependências entre classes, apenas o trabalho de Briand *et al.* (2001) discute superficialmente como efetuar o mapeamento entre diagramas UML em fase de projeto e o ORD correspondente. A inexistência de um processo de mapeamento de modelos de projeto OO para o ORD é mais uma motivação para definir como é feito o mapeamento entre modelos de projeto AO e o AORD.

O processo de mapeamento é feito em quatro grandes passos: no primeiro, são analisados os diagramas de classes; no segundo, são analisadas regras de transformação de alteração estrutural; no terceiro, são analisados os diagramas de sequência tradicionais; e, no quarto, são analisadas regras de transformação de alteração de comportamento. O AORD é construído gradativamente conforme o detalhamento dos quatro grandes passos apresentados nas próximas seções.

Para exemplificar o processo de mapeamento, é utilizada a modelagem baseada em interesses do sistema Telecom, apresentada na Seção 5.2. É importante observar que o projeto proposto na Seção 5.2 pode ser implementado em diferentes linguagens de programação OA e,

até mesmo, em linguagens de programação OO. Como o estudo tem o foco em programação OA, considera-se preliminarmente que na fase de projeto cada interesse transversal será implementado como um aspecto de AspectJ. Futuramente, na fase de implementação, isso pode não ocorrer necessariamente, e um interesse transversal que foi inicialmente modelado como um aspecto pode dar origem a vários aspectos e classes. Nesse momento, então, o engenheiro de software poderá refinar e rodar novamente o algoritmo, para verificação da nova ordenação e as necessidades de *stubs*.

Em princípio, para executar o mapeamento entre os diagramas e o AORD, deve-se alinhar os interesses transversais e os respectivos aspectos. Para o sistema Telecom, os interesses de temporização, tarifação e registro de chamadas ao medidor de tempo são representados pelos aspectos `Timing`, `Billing` e `TimerLog`, respectivamente. Além dos três aspectos, também fazem parte do sistema Telecom os aspectos e classes necessários para o acoplamento do *framework* de persistência, já que será utilizado um *framework* já existente (Camargo e Masiero, 2005).

Para encontrar dependências da programação OO, os dois principais diagramas UML que podem ser utilizados para efetuar o mapeamento são o diagrama de classes e o diagrama de sequência – ou o diagrama de colaboração. O diagrama de classes mostra o relacionamento estrutural entre classes, no qual é possível identificar dependências de herança, agregação e associação. O diagrama de sequência pode ser usado de maneira auxiliar para revelar dependências de associação provenientes de invocação de métodos e referências a atributos. As dependências referentes à programação OA podem ser identificadas por meio das regras de transformação da notação MATA, que descrevem alterações estruturais e alterações de comportamento. As alterações estruturais são feitas por declarações intertipos enquanto que as alterações de comportamento são feitas por meio de adendos que removem ou adicionam comportamento em pontos de junção. Portanto, nos diagramas da notação MATA podem ser encontrados dependências de declarações intertipos, entrecorte, herança, agregação e associação.

Durante o processo de mapeamento são adicionados ao AORD, em forma de vértices e arestas, classes, aspectos e seus relacionamentos, descritos pelos diagramas do modelo de projeto. O processo de mapeamento apresentado nas próximas seções deve seguir duas diretrizes gerais: uma para adicionar vértices e outra para adicionar arestas. Um vértice só deve ser adicionado ao AORD caso ainda não exista, isto é, não pode haver dois vértices com o mesmo nome. Uma dependência identificada a partir do relacionamento entre aspectos e classes só é adicionada ao AORD em forma de uma nova aresta quando não existir previamente uma aresta ligando os dois vértices que representam a classe ou aspecto que se relacionam ou quando a aresta entre os dois vértices já existir, mas denotar menor acoplamento, fazendo com que seja necessário alterar o

rótulo da aresta de acordo com a nova dependência. Assim, como um AORD é um dígrafo e não um multigrafo, não deve haver mais de uma aresta ligando dois vértices.

5.3.1 Análise de diagramas de classes

Note-se que, segundo o detalhamento do passo aqui descrito, o mapeamento a partir do diagrama de classes é direto: cada classe se torna um vértice e cada relacionamento pode se tornar uma aresta anotada, de acordo com o tipo de relacionamento. Portanto, neste passo do processo de mapeamento podem ser mapeadas no AORD dependências do tipo "I", "Ag" e "As".

1 Para cada diagrama de classes:

- 1.1 Para cada classe encontrada no diagrama de classes, deve-se adicionar ao AORD um vértice com o nome da classe;
- 1.2 Para cada relacionamento de herança encontrado, deve-se adicionar ao AORD uma aresta do tipo "It" que liga o vértice que representa a subclasse ao vértice que representa a superclasse;
- 1.3 Para cada relacionamento de composição ou agregação encontrado, deve-se adicionar ao AORD uma aresta do tipo "Ag" que liga o vértice que representa a classe agregadora ao vértice que representa a classe agregada;
- 1.4 Para qualquer outro relacionamento encontrado, deve-se adicionar ao AORD uma aresta do tipo "As" de acordo com a navegabilidade do relacionamento.

O mapeamento feito a partir do diagrama de classes da Figura 5.1 resulta no AORD ilustrado na Figura 5.11. Note-se pelos relacionamentos entre `Connection` e `Customer`, e `Call` e `Customer` mostrados no diagrama de classes que o mapeamento segue a diretriz que impede que hajam duas arestas ligando os mesmos vértices. No AORD parcial existem apenas duas arestas ligando `Call` a `Customer` e `Connection` a `Customer`. Também é possível observar que cada classe deu origem a um vértice. Nos AORDs parciais mostrados nesta e nas próximas seções, estão destacados os vértices e arestas adicionados gradativamente ao AORD.

5.3.2 Análise de regras de transformação de alteração estrutural

As regras de transformação de alterações estruturais descrevem três tipos de alterações: introdução de atributos, introdução de métodos e alteração de hierarquia de herança. O detalhamento deste passo do processo de mapeamento é apresentado a seguir:

1 Para cada regra de transformação:

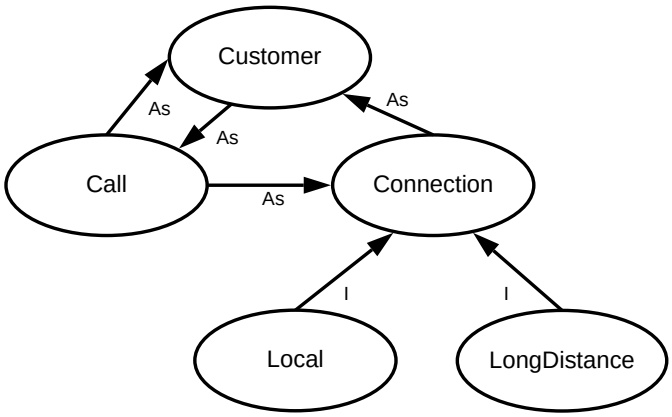


Figura 5.11: AORD parcial construído a partir do diagrama de classes do sistema Telecom.

1.1 Deve-se identificar e nomear o aspecto referente ao interesse transversal que possui a regra de transformação e adicionar ao AORD um vértice com o mesmo nome do aspecto;

1.2 Para cada nova classe, deve-se adicionar ao AORD um vértice com o mesmo nome da classe;

1.3 Para cada introdução de atributo encontrada na regra:

1.3.1 Deve-se identificar o alvo da introdução e adicionar ao AORD uma aresta do tipo “It” ligando o vértice que representa o aspecto ao vértice que representa o alvo da introdução;

1.3.2 Deve-se identificar a classe usada como tipo do atributo e adicionar ao AORD uma aresta do tipo “As” ligando o vértice que representa o aspecto ao vértice que representa a classe usada como tipo;

1.4 Para cada introdução de método encontrada na regra:

1.4.1 Deve-se identificar o alvo da introdução e adicionar ao AORD uma aresta do tipo “It” ligando o vértice que representa o aspecto ao vértice que representa o alvo da introdução;

1.4.2 Deve-se identificar as classes usadas como tipo de parâmetros e retorno dos métodos e adicionar arestas do tipo “As” ligando o vértice que representa o aspecto aos vértices que representam as classes usadas como tipo;

1.5 Para cada alteração de hierarquia de herança encontrada na regra:

1.5.1 Deve-se identificar os alvos da alteração, isto é, a subclasse e a superclasse, e adicionar ao AORD arestas do tipo “It” ligando o vértice que representa o aspecto aos vértices que representam os alvos da alteração de hierarquia de herança;

1.5.2 Deve-se adicionar ao AORD uma aresta do tipo “I” ligando a subclasse e a superclasse identificadas como alvo da introdução.

Conforme pode-se observar no detalhamento deste passo, em regras de transformação de declarações intertipos podem ser encontradas e mapeadas no AORD dependências do tipo “It”, “I” e “As”. Para definir quais tipos de dependências são mapeadas, é necessário escolher o aspecto referente ao interesse transversal que possui a regra de transformação e adicioná-lo ao AORD. Como a notação MATA não mostra explicitamente qual é esse aspecto, adotam-se as decisões tomadas durante o projeto para essa escolha. Por exemplo, analisando as regras de transformação R1 e R2, da Figura 5.2, e considerando as decisões de projeto descritas na Seção 5.3, pode-se perceber que o aspecto `Timing` faz introduções de atributos e métodos relativas ao controle de temporização das chamadas telefônicas. Assim, um vértice representado o aspecto `Timing` deve ser adicionado ao AORD, conforme ilustrado na Figura 5.12.

É necessário encontrar, também, quais e de que tipo são as dependências. Ao comparar o lado esquerdo da regra com o lado direito da regra, observando quais atributos e métodos são introduzidos, é possível identificar a classe alvo das introduções. O aspecto que faz a introdução tem uma dependência do tipo “It” com sua classe alvo. Além disso, esse aspecto possui dependências com todas as classes usadas como tipos na introdução de métodos e atributos. Assim, deve-se observar o tipo do retorno e os parâmetros dos métodos introduzidos, bem como o tipo dos atributos introduzidos para identificar e mapear dependências do tipo “As”.

Novamente, analisando as regras de transformação R1 e R2, da Figura 5.2, observa-se que, na classe `Customer` são introduzidos o atributo `totalConnectTime` e seus métodos de manipulação `addTotalConnectTime(Long time)` e `getTotalConnectTime()`, enquanto que na classe `Connection` são introduzidos o atributo `timer` e seus métodos de manipulação, `setTimer(Timer timer)` e `getTimer()`. Portanto, devem ser adicionados ao AORD uma aresta do tipo “It” ligando o vértice chamado `Timing` e os vértices que representam as classes alvo das introduções `Customer` e `Connection`, conforme pode-se observar na Figura 5.12.

Nas introduções feitas em `Connection`, o tipo do atributo, do parâmetro e do retorno dos métodos introduzidos é `Timer`. Assim, deve-se adicionar ao AORD um vértice para a classe `Timer` e uma aresta do tipo “As” ligando `Timing` a `Timer`, o que é mostrado na Figura 5.12. Ao contrário do que possa parecer, a dependência mapeada no AORD é entre `Timing` e `Timer` ao invés de ser entre `Connection` e `Timer`, porque a funcionalidade de temporização é de responsabilidade do aspecto `Timing` e, sem a existência dele no modelo, não existe o relacionamento de associação entre `Connection` e `Timer`.

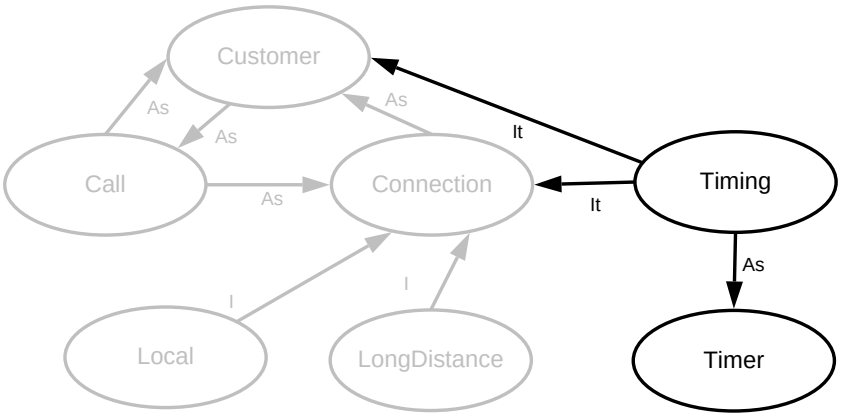


Figura 5.12: AORD parcial com as declarações intertipos feitas pelo aspecto `Timing`.

As introduções feitas pelo aspecto `Billing`, descritas pelas regras R5 e R6, mostradas na Figura 5.5, ocorrem nas classes alvo `Local`, `LongDistance`, `Customer` e `Connection`, resultando na inclusão de arestas do tipo "It" ligando `Billing` a `Local`, `LongDistance`, `Customer` e `Connection`, conforme ilustrado na Figura 5.13. A análise dos atributos e métodos introduzidos indica a inclusão de uma aresta do tipo "As" ligando `Connection` a `Customer`, porém, como já existe no AORD uma aresta desse tipo ligando os dois vértices, nada é feito.

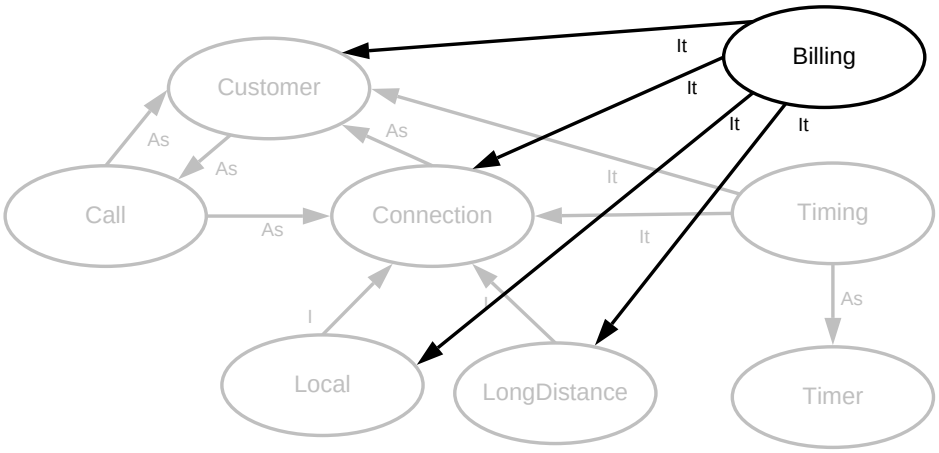


Figura 5.13: AORD parcial com as declarações intertipos feitas pelo aspecto `Billing`.

Além de depender da classe alvo e das classes usadas no retorno e nos parâmetros do método, o aspecto que faz a introdução de um método também depende de todas as classes que são usadas no corpo do método introduzido. Por exemplo, quando existem variáveis locais do método, devem ser mapeadas dependências do tipo "As" entre o aspecto que introduz o método e as classes usadas como tipos nas variáveis locais. Para detectar dependências desse tipo é

necessário que haja um diagrama de sequência descrevendo o comportamento do método introduzido.

As regras de transformação de declarações intertipos também descrevem como são feitas alterações de hierarquia de herança. Para encontrar as dependências em regras de transformação desse tipo, basta identificar o relacionamento de herança criado comparando o lado esquerdo da regra com o lado direito da regra. Assim, verifica-se que existe o aspecto com a declaração intertipo e duas classes que representam a superclasse e a subclasse do relacionamento de herança alterado. Devem ser criadas, então, duas dependências do tipo "It", entre o aspecto e as classes, e uma dependência do tipo "I" entre as duas classes.

Um exemplo pode ser ilustrado pelas regras de transformação R11, da Figura 5.10, em que pode-se observar que a classe `Customer` e a classe `Connection` se tornam especializações da classe `PersistentRoot` após a combinação. Portanto, conforme pode ser observado no AORD da Figura 5.14, existem três arestas com rótulo "It" que partem de `MyPersistentEntities` e vão para `PersistentRoot`, `Customer` e `Connection`, e duas arestas com o rótulo "I" que ligam `Customer` e `Connection` a `PersistentRoot`. Estas últimas dependências se devem ao fato de que sem a existência do aspecto `MyPersistentEntities` não existe o relacionamento de herança entre `PersistentRoot` e as classes `Customer` e `Connection`. Com essas regras de mapeamento, as arestas do AORD que unem classes e o aspecto não podem ser removidas temporariamente durante a execução do algoritmo de ordenação, garantindo que o AORD permaneça consistente.

5.3.3 Análise de diagramas de sequência

Na análise dos diagramas de sequência deve-se observar apenas as invocações de métodos presentes nos diagramas. Portanto, neste passo do processo de mapeamento podem ser mapeados para o AORD somente dependências do tipo "As". Os passos de detalhamento são mostrados a seguir:

1 Para cada diagrama de sequência:

  1.1 Para cada invocação de método deve-se adicionar ao AORD uma aresta do tipo "As" ligando o vértice que representa a classe que faz a invocação ao vértice que representa a classe que possui o método invocado;

Neste passo do processo a estrutura do sistema já está estabelecida, uma vez que as regras de alterações estruturais já foram analisadas. Portanto, é importante observar atentamente que

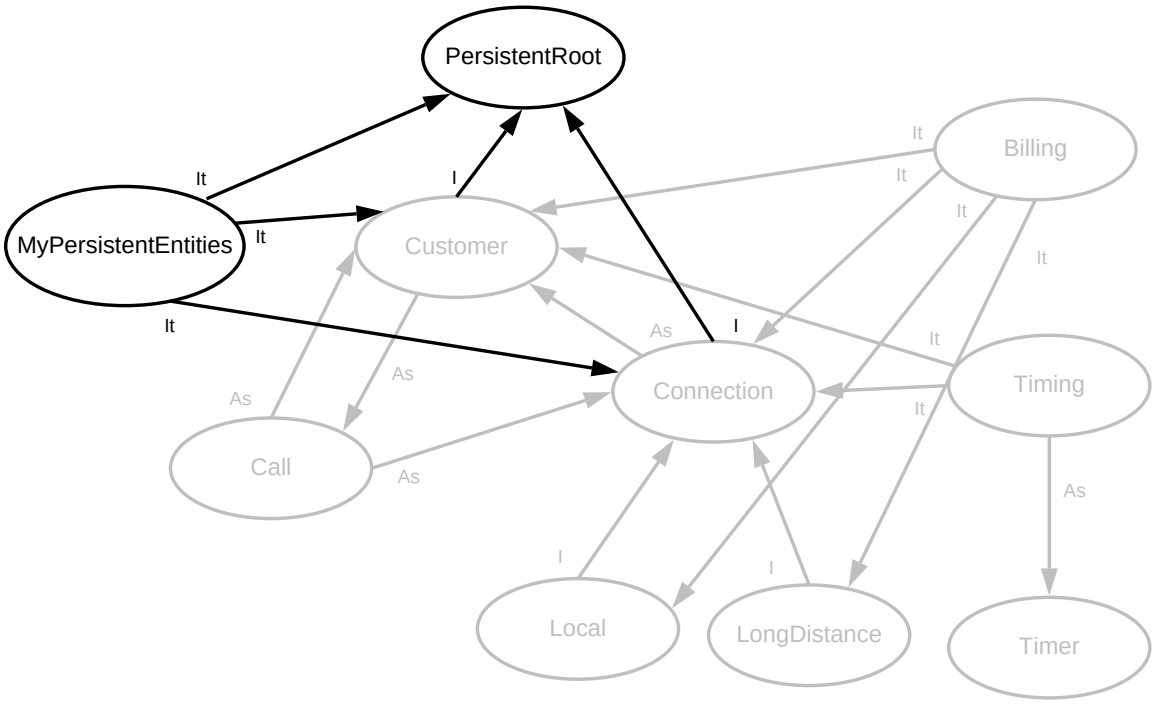


Figura 5.14: AORD parcial com as declarações intertipos feitas pelo aspecto `MyPersistentEntities`.

métodos invocados no diagrama de sequência podem não estar realmente implementados na classe, mas sim terem sido introduzidos por aspectos ou terem sido herdados via alterações de hierarquia de herança feita por aspectos. Essa situação é descrita em detalhes na Seção 5.3.5.

Algumas vezes são omitidos alguns relacionamentos com o intuito de não poluir um diagrama de classes, como por exemplo os relacionamentos de dependência[1]. Em casos como esse, diagramas de sequência são úteis porque auxiliam na identificação de dependências.

Para exemplificar como é feito o mapeamento a partir de diagramas de sequência, é apresentado na Figura 5.15 o diagrama de sequência do construtor da classe `Call` presente no diagrama de classes da Figura 5.1. O construtor recebe duas instâncias da classe `Customer` como parâmetros para que uma conexão de longa ou curta distância seja criada. Note-se que pelo diagrama de classes mostrado na Figura 5.1, em que relacionamentos de dependência são omitidos, não é possível inferir que há relacionamentos entre `Call`, `LongDistance` e `Local`. Ao invés disso, o relacionamento é estabelecido entre `Call` e `Connection`, denotado pela associação `connections`. Pelo diagrama de sequência é possível ver que existem chamadas a métodos de `LongDistance` e `Local` feitas a partir de `Call`, caracterizando os

[1]Note-se que um relacionamento de dependência da UML (OMG, 2007) não é sinônimo de dependência no AORD. Esse tipo de relacionamento dá origem a uma dependência no AORD, assim como outros tipos de relacionamentos da UML.

relacionamentos entre essas classes. Assim, deve-se mapear dependências do tipo "As" ligando `Call` a `LongDistance` e `Local`, como pode ser visto na Figura 5.16.

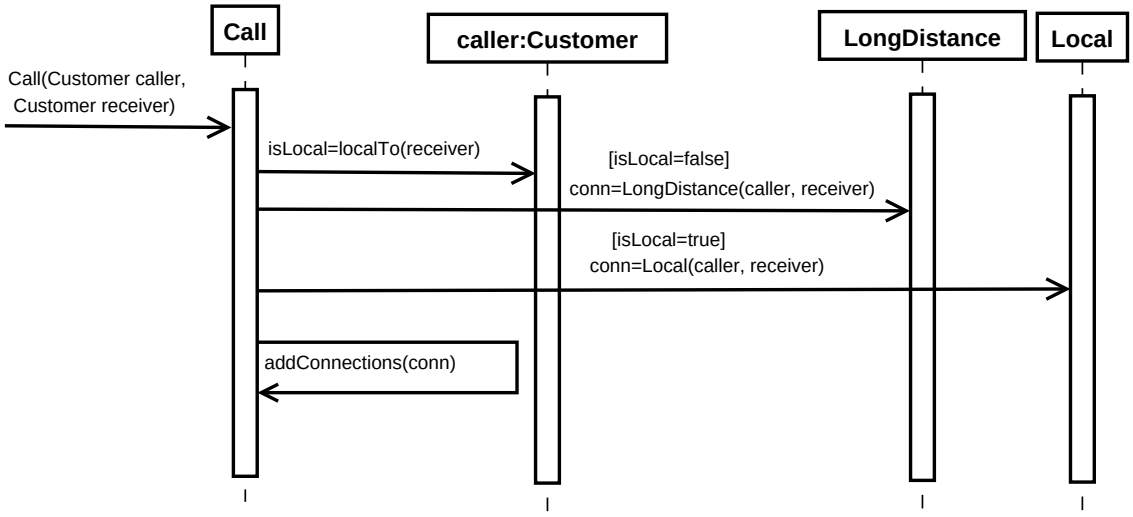


Figura 5.15: Diagrama de sequência de um construtor da classe `Call`.

.

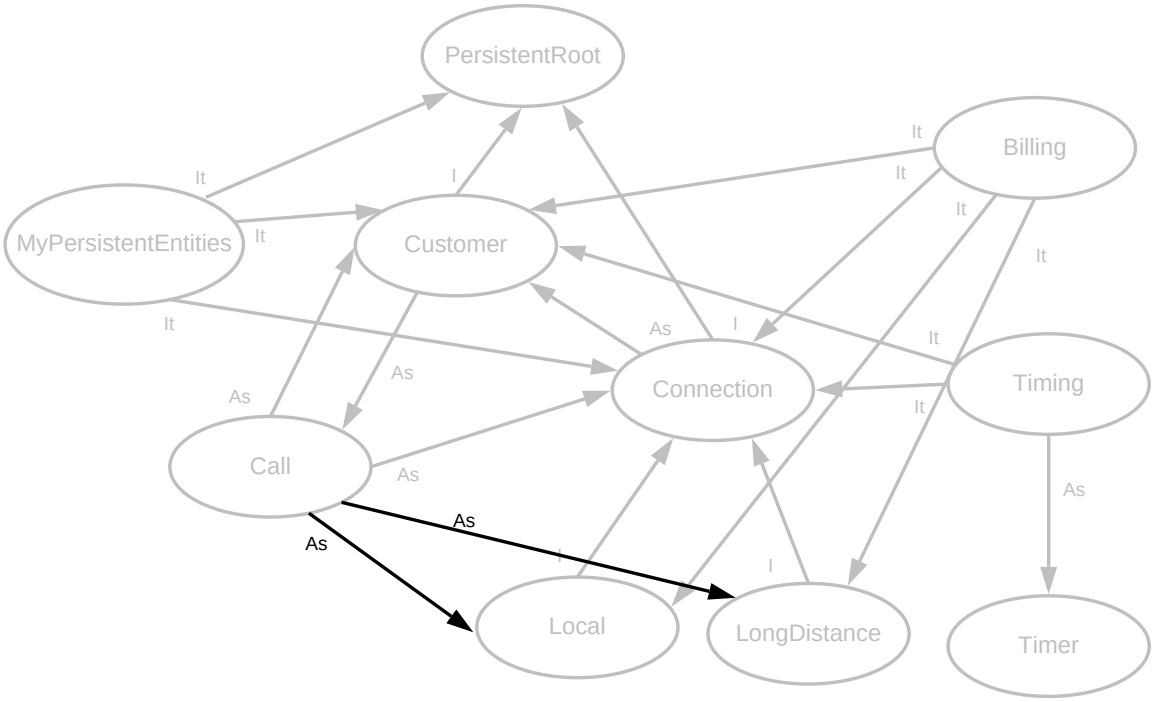


Figura 5.16: AORD construído a partir de um diagrama de sequência do sistema Telecom.

5.3.4 Análise de regras de transformação de alteração de comportamento

As regras de transformação que descrevem comportamento entrecortante contém relacionamentos que são mapeados no AORD como dependências do tipo “C” e “As”. Os passos de detalhamento são:

1 Para cada regra de transformação:

1.1 Deve-se identificar e nomear o aspecto referente ao interesse transversal que possui a regra de transformação e adicionar ao AORD um vértice com o mesmo nome do aspecto;

1.2 Para cada ponto de junção encontrado a partir da aplicação da regra, deve-se adicionar ao AORD uma aresta do tipo “C” ligando o aspecto com a regra de transformação ao aspecto ou à classe que possui o ponto de junção;

1.3 Para cada método invocado pelo comportamento adicionado pela regra, deve-se adicionar ao AORD uma aresta do tipo “As” ligando o aspecto com a regra de transformação à classe que possui o método invocado.

Da mesma forma que no grande passo anterior, é importante observar que o comportamento adicionado por um aspecto pode invocar métodos que são introduzidos por aspectos ou herdados via alterações de hierarquia de herança feita por aspectos. Além disso, um ponto de junção capturado por uma regra de transformação pode estar localizado em um método ou atributo introduzido por um aspecto. Portanto, apesar dos passos de detalhamento indicarem que as dependências acontecem entre classes, é necessário considerar essas situações. Ambas as situações são descritas em detalhes na Seção 5.3.5.

Conforme pode ser observado pelos passos de detalhamento, para identificar as dependências do aspecto é necessário encontrar um ponto de junção em que o aspecto atua usando a própria regra de transformação, porque o aspecto sempre depende do ponto de junção no qual ele age. Como visto na Seção 5.2.4, as regras de transformação algumas vezes não mostram explicitamente o ponto de junção. Em alguns casos, é necessário analisar cuidadosamente o resultado da aplicação da regra para mapear de maneira precisa as dependências no AORD. O ponto de junção é basicamente uma sequência de chamadas de métodos marcados pelo estereótipo «`context`». Algumas vezes não existe essa marcação, e deve-se procurar por fragmentos de chamadas de métodos marcados com os estereótipos «`create`» ou «`delete`». Neste caso, o ponto de junção pode ser identificado pelas invocações do método imediatamente anterior ou do método imediatamente posterior ao fragmento.

Por exemplo, a chamada ao `Connection.complete()` da regra de transformação R3, mostrado na Figura 5.3, indica qual é o ponto de junção no qual o aspecto `Timing` adiciona comportamento. Note-se que, nesse caso, não é necessário aplicar a regra para identificar a classe alvo, porque a regra descreve que a classe alvo é `Call`, uma vez que ela faz uma chamada ao método `Connection.complete()`. Apenas para ilustrar o conceito, o resultado da aplicação da regra de transformação da Figura 5.3 é apresentado na Figura 5.17. Portanto, deve-se adicionar ao AORD uma aresta do tipo "C" ligando `Timing` a `Call`. Como o aspecto `Billing` entrecorta o mesmo ponto de junção do aspecto `Timing`, que, conforme descrito pela regra de transformação R7 ilustrada na Figura 5.6, ocorre em uma chamada ao método `Connection.complete()`, também existe uma aresta do tipo "C" entre `Billing` e `Call`. Na Figura 5.7, apresentada na regra R8, pode ser visto que `Billing` faz uma chamada ao método `Timer.getTime()`, denotando uma dependência do tipo "As" entre o aspecto e a classe. O AORD mostrado na Figura 5.18 contém todas essas dependências discutidas.

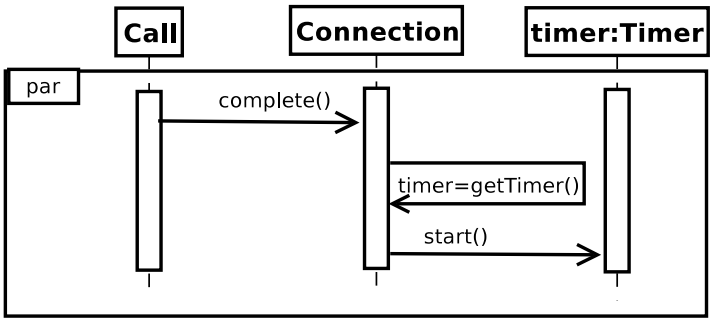


Figura 5.17: Sequência de chamadas de métodos após a aplicação da regra R4, mostrada na Figura 5.3.

Além de se relacionar com o ponto de junção, o aspecto tem adendos que invocam métodos de outras classes, ocasionando dependências do tipo "As". Para identificá-las, deve-se analisar chamadas de métodos com o estereótipo «`create`», ou chamadas de métodos dentro de fragmentos marcados com o estereótipo «`create`». Esses métodos representam o comportamento adicionado pelo aspecto. Como exemplo, pode-se observar na regra R3, da Figura 5.3, que o novo comportamento adicionado pelo fragmento `par` tem chamadas aos métodos `Connection.getTimer())` e `Timer.start()`. Dessa forma, também existe uma dependência do tipo "As" de `Timing` com a classe `Connection` e com a classe `Timer`. O primeiro ponto importante a se observar é que, conforme ilustrado na Figura 5.2(b), a regra R2 descreve que o método `Connection.getTimer()` deve ser introduzido pela próprio aspecto `Timing`, o que, via de regra, deveria fazer com que houvesse uma auto-dependência de `Timing`, que não deve ser mapeada. O segundo ponto importante é que se o AORD da

Figura 5.19 for observado, já existe uma dependência entre `Timing` e `Timer` do tipo "As", portanto, as duas dependências não são mapeadas.

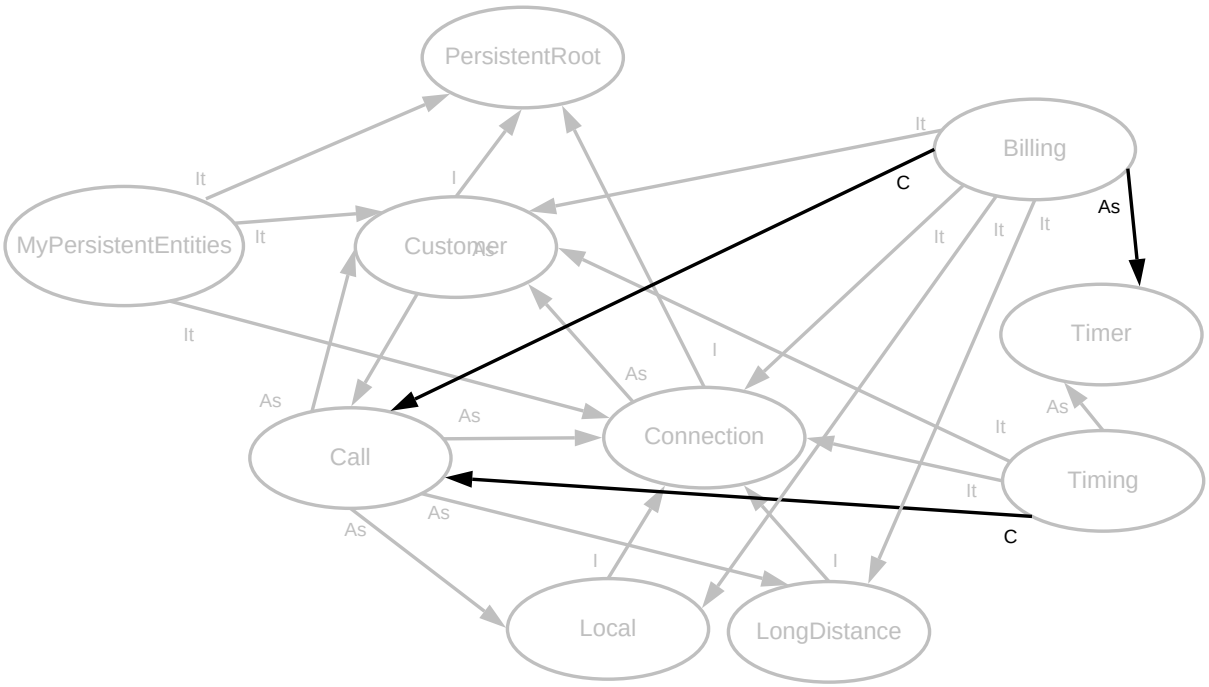


Figura 5.18: AORD parcial com dependências encontradas nas regras de transformação R3, R7 e R8.

Um caso interessante de dependência do tipo "C" pode ser observado nas regras de transformação R9 e R10, mostradas na Figura 5.8 e na Figura 5.9, em que um aspecto se relaciona a outro aspecto. A primeira regra descreve que devem ser capturadas chamadas ao método `Timer.start()`, enquanto que a segunda descreve que devem ser capturadas chamadas ao método `Timer.stop()`. Note-se que o ponto de junção para ambas as regras ocorre dentro de adendos do aspecto `Timing`, conforme pode-se observar na Figura 5.3 e na Figura 5.4, o que gera uma dependência do tipo "C" entre o aspecto `TimerLog` e o aspecto `Timing`. Nesse mesmo exemplo, existe também uma dependência do tipo "As" entre `TimerLog` e a classe `Timer`, originária da chamada aos métodos `Timer.start()` e `Timer.stop()`. O AORD da Figura 5.19 apresenta o mapeamento das duas regras de transformação.

Quando um adendo faz uma chamada a um método, existe uma dependência do tipo "As" entre o aspecto que possui o adendo e a classe que possui o método. No entanto, quando um adendo faz uma chamada a um método introduzido, a dependência na verdade acontece entre o aspecto que possui o adendo e o aspecto responsável pela introdução do método. As regras de transformação da notação MATA não mostram explicitamente quando isso acontece. É importante observar que em sequências de chamadas de métodos do adendo, deve-se rastrear

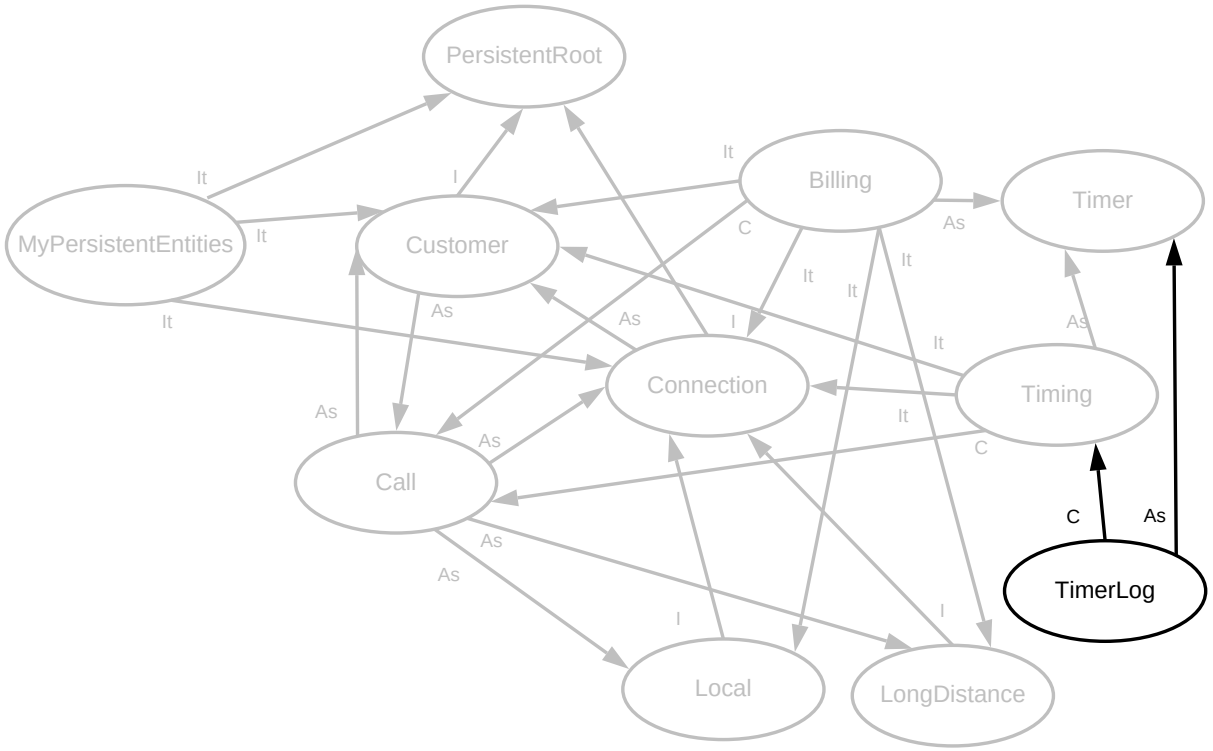


Figura 5.19: AORD parcial com dependências encontradas na regras de transformação R9 e R10.

se de fato o método invocado pertence à classe ou se foi introduzido por aspecto em uma regra anterior. Esse é um exemplo do caso complexo 2.1, descrito na Seção 5.3.5. Na regra R8, da Figura 5.7 é possível exemplificar como é feito o mapeamento quando o adendo de um aspecto utiliza um método introduzido por um outro aspecto. Note-se que na regra R8, existe uma chamada ao método `Connection.getTimer()`. No entanto, o método `getTimer()` foi introduzido na classe `Connection` pelo aspecto `Timing`, conforme pode ser observado na regra R2, da Figura 5.2(b). Assim, uma aresta do tipo "As" que liga `Billing` a `Timing` é mapeada no AORD apresentado na Figura 5.20.

O rastreamento para encontrar se um método está em uma classe ou foi introduzido por um aspecto pode ser uma tarefa custosa. A programação orientada a aspectos permite que aspectos façam alterações estruturais que afetam as classes de diferentes formas. O resultado é que uma classe pode conter métodos implementados ou usar métodos provenientes de diferentes aspectos. Por exemplo, ao invés do método ser implementado na classe ou introduzido por um aspecto, ele pode ser resultado de uma alteração de hierarquia de herança, como pode ser observado nas regras R4, R8 e R11, mostradas na Figura 5.4, Figura 5.7 e na Figura 5.10. Esse é um exemplo do caso complexo 2.2, descrito na Seção 5.3.5. Note-se que nas regras de transformação R4 e R8 existem chamadas ao método `Customer.save()`. Porém, esse método não

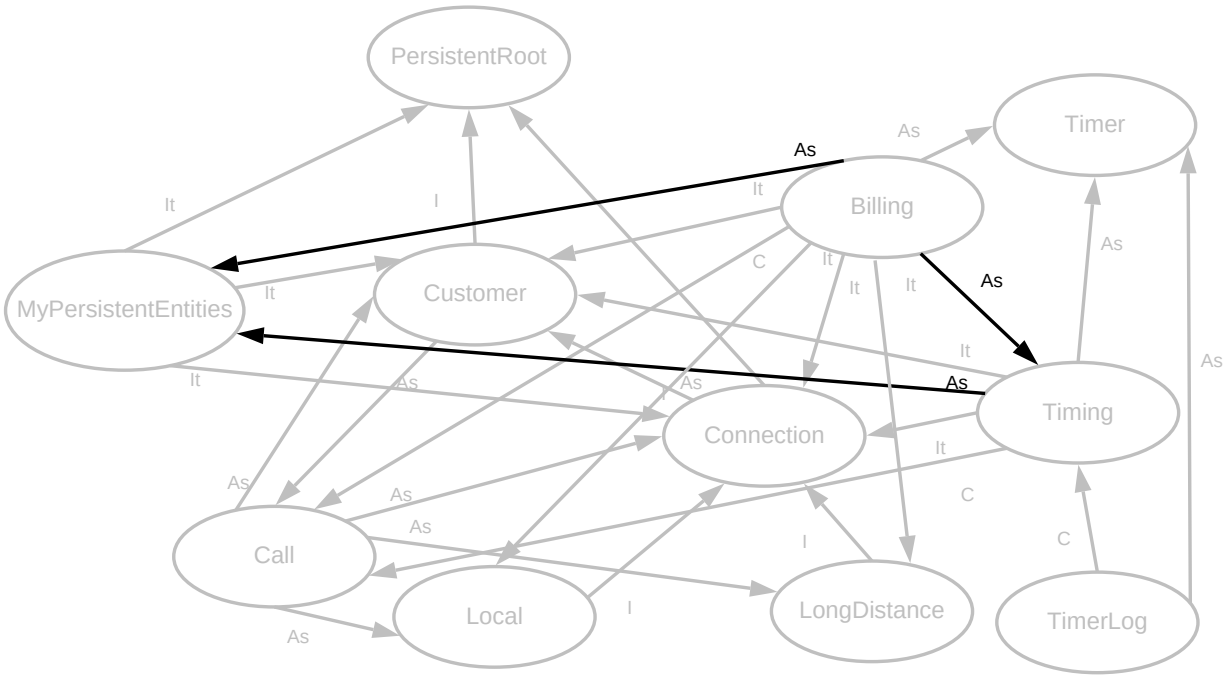


Figura 5.20: AORD parcial com as dependências encontradas nas regras de transformação R4 e R8

está definido na classe `Customer`, ele é herdado da classe `PersistentRoot` por meio da regra R11, conforme mostrado na Figura 5.10. Assim, dependências que aparentemente seriam entre `Timing` e `Customer`, e `Billing` e `Customer`, na verdade são mapeadas no AORD como uma aresta do tipo "As" ligando `Timing` e `Billing` a `MyPersistentEntities`. O AORD da Figura 5.20 destaca estas últimas dependências mapeadas, enquanto o AORD da Figura 5.21 apresenta todas as regras de transformação mapeadas.

5.3.5 Mapeamento de casos complexos

Existem alguns mapeamentos que são complexos e mais difíceis para serem identificados. Algumas dependências só podem ser mapeadas no AORD se o rastreamento levar em consideração combinações de regras de alteração estrutural e de alteração de comportamento. Existem três casos básicos que podem ser combinados entre si, dificultando significativamente o trabalho de mapeamento:

1 Quando um método adicionado por meio de uma regra de alteração estrutural faz uma invocação a:

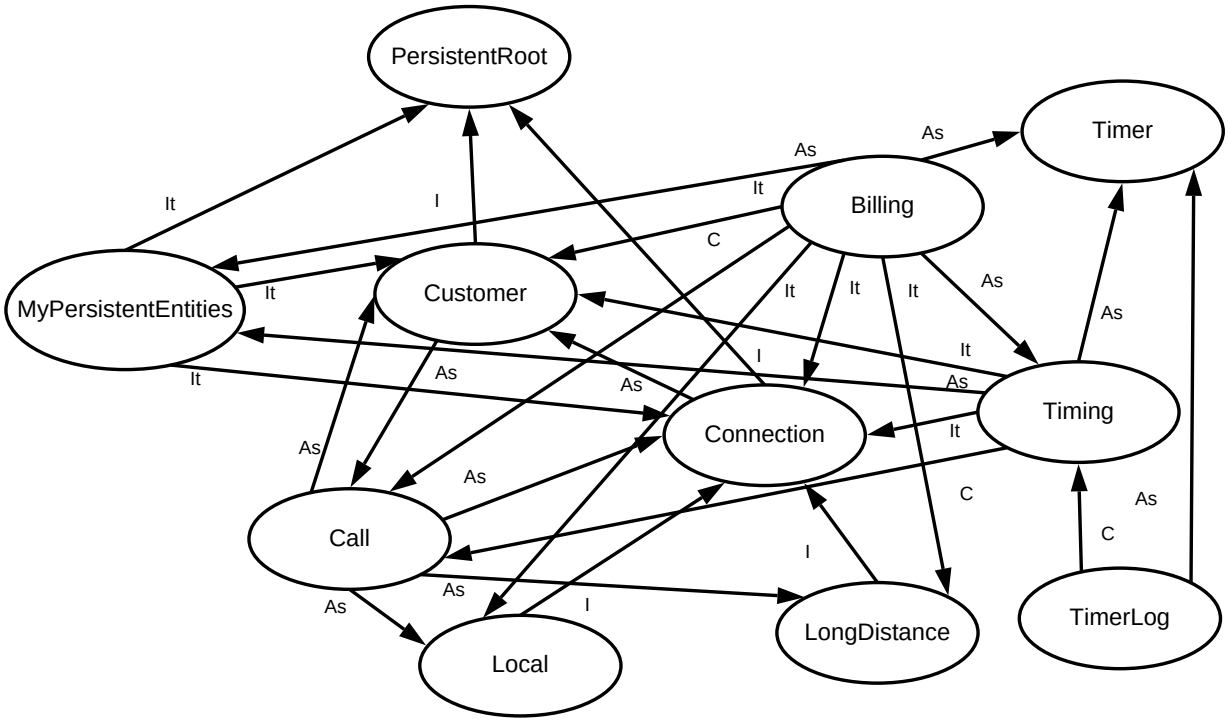


Figura 5.21: AORD final com todas as regras de transformação mapeadas.

1.1 um método também adicionado por meio de uma regra de alteração estrutural, deve-se adicionar ao AORD uma aresta do tipo “As” ligando o aspecto faz a invocação ao aspecto que possui o método invocado;

1.2 um método herdado via alteração de hierarquia de herança, deve-se adicionar ao AORD uma aresta do tipo “As” ligando o aspecto que faz a invocação ao aspecto que possui a alteração de hierarquia de herança;

2 Quando uma regra de alteração de comportamento faz uma invocação a:

2.1 um método adicionado por meio de uma regra de alteração estrutural, deve-se adicionar ao AORD uma aresta do tipo “As” ligando o aspecto que faz a invocação ao aspecto que possui o método invocado;

2.2 um método herdado via alteração de hierarquia de herança, deve-se adicionar ao AORD uma aresta do tipo “As” ligando o aspecto que faz a invocação ao aspecto que possui a alteração de hierarquia de herança;

3 Quando uma regra de alteração de comportamento entrecorta um ponto de junção localizado em um método adicionado por uma regra de alteração estrutural, deve-se adicionar ao AORD uma aresta do tipo “As” ligando o aspecto que faz o entrecorte ao aspecto que possui o ponto de junção.

Note-se os passos recém descritos referem-se a invocações de métodos, mas o uso de atributos adicionados por meio de regras de alteração de comportamento, entrecortados por regras de alteração de comportamento ou herdados via alteração de hierarquia de herança ou entrecortados, também denotam dependência do tipo "As".

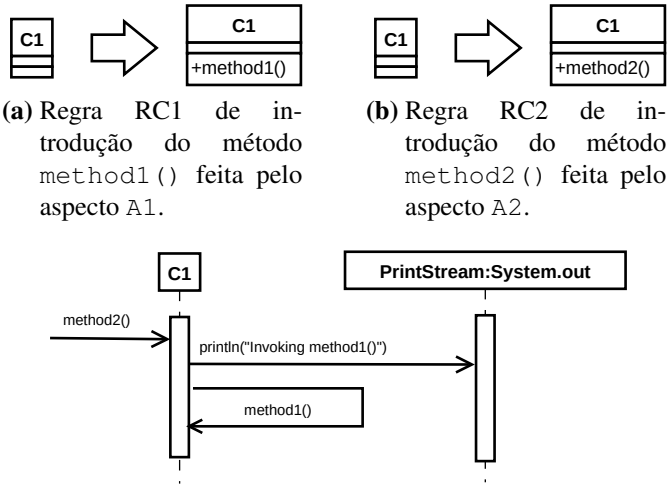
A seguir são apresentados exemplos de mapeamento dos casos complexos 1.2, 1.3 e 3, porque exemplos dos casos complexos 2.1, 2.2 são mostrados nas seções anteriores. Cada exemplo aqui apresentado deve ser compreendido separadamente, isto é, apesar das classes e aspectos terem o mesmo nome em diferentes exemplos, eles não são os mesmos. Vale ressaltar que, novamente, considera-se que os interesses transversais descritos pelas regras de transformação são implementados em aspectos.

Conforme pode-se perceber nos exemplos, nos casos complexos 1.1 e 1.2 é necessário que haja, além dos diagramas que descrevem as introduções e alterações de hierarquia de herança, diagramas de sequência descrevendo o comportamento dos métodos introduzidos para que as dependências sejam encontradas e mapeadas. Deve-se observar no diagrama de sequência do método introduzido e buscar por chamadas a outros métodos introduzidos ou herdados via alteração de hierarquia de herança.

Na Figura 5.22 é ilustrado o caso complexo 1.1. A regra RC1 da Figura 5.22(a) descreve que o aspecto `A1` introduz o método `method1()` na classe `C1`, enquanto que regra RC2 da Figura 5.22(b) descreve que o aspecto `A2` introduz o método `method2()` na classe `C1`. Na Figura 5.22(c) é apresentado um diagrama de sequência que mostra o método `C1.method2()` invocando o método `C1.method1()`. É importante observar que nenhum dos dois métodos está na classe `C1`, eles foram introduzidos por aspectos. Dessa maneira, existe uma dependência do tipo "As" entre o aspectos que fazem as introduções, `A2` e `A1`.

O caso complexo 1.2 é ilustrado na Figura 5.22(c). A regra RC3 da Figura 5.23(a) descreve que o aspecto `A1` faz uma declaração de alteração de hierarquia de herança em que a classe `C3` se torna uma especialização da classe `C2`, herdando o método `method2()`. A regra RC4 da Figura 5.23(b) descreve que o aspecto `A2` introduz o método `method1()` na classe `C1`. O diagrama de sequência apresentado na Figura 5.23(c) mostra que o método `C1.method1()` invoca o método `C3.method2()`. Novamente, como no caso anterior, os métodos não estão nas classes, e o método introduzido pelo aspecto `A2` invoca outro método herdado via alteração de hierarquia de herança feita por `A1`. Assim, existe uma dependência do tipo "As" ligando `A2` a `A1`.

O mapeamento do caso complexo 3, também encontrado durante a execução dos estudos, é feito por meio da análise do diagrama da notação MATA que descreve os adendos e os pontos de junção. Ao invés de observar a chamada de métodos feitas pelo adendo, como nos casos

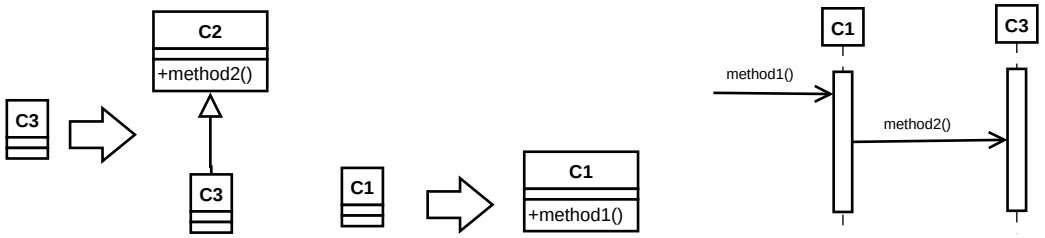


(a) Regra RC1 de introdução do método `method1()` feita pelo aspecto `A1`.

(b) Regra RC2 de introdução do método `method2()` feita pelo aspecto `A2`.

(c) Diagrama de sequência que descreve o método `method2()`.

Figura 5.22: Exemplo do caso 1.1 do mapeamento complexo.



(a) Regra RC3 de alteração de hierarquia de herança feita pelo aspecto `A1`, em que `C3` se torna uma especialização de `C2`.

(b) Regra RC4 de introdução do método `method1()` feita pelo aspecto `A2`.

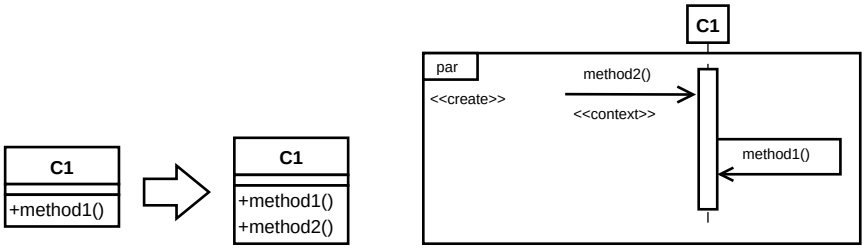
(c) Diagrama de sequência que descreve o método `method1()`.

Figura 5.23: Exemplo do caso 1.2 de mapeamento complexo.

2.1 e 2.2, deve-se observar o ponto de junção. Caso ele seja um método introduzido, então está caracterizado o caso complexo 3.

O caso complexo 3 é exemplificado na Figura 5.24. A regra RC5 da Figura 5.24(a) descreve que o aspecto `A1` introduz o método `method2()` na classe `C1`. A regra RC6 da Figura 5.24(b) descreve que o método introduzido `C1.method2()` é entrecortado pelo aspecto `A2`. Portanto, existe uma dependência do tipo "As" entre o aspecto `A2` e o aspecto `A1`.

Na Figura 5.25 é apresentado um conjunto de regras de transformação e um diagrama de sequência que mostram um exemplo de combinação de casos complexos. Na Figura 5.26(a) é apresentado um AORD parcial com dependências mapeadas a partir das regras de transformação



(a) Regra RC5 de introdução do método `method2()` feita pelo aspecto `A1`.

(b) Regra RC6 feita pelo aspecto `A2`, descrevendo que o método introduzido `method2()` é entrecortado.

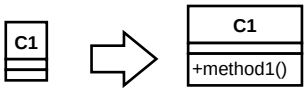
Figura 5.24: Exemplo do caso 3 de mapeamento complexo.

RC7, RC8 e RC9. O mapeamento mostra várias dependências de introdução de métodos e uma alteração de hierarquia de herança referentes a diferentes aspectos.

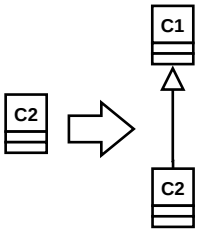
Na Figura 5.26(b) é apresentado o AORD com a adição das dependências encontradas a partir da análise do diagrama de sequência mostrado na Figura 5.25(d). O diagrama de sequência da Figura 5.25(d) mostra a invocação de um método feita pelo método `C3.method3()`. No entanto, é importante observar que o método `method3()` é introduzido em `C3` pelo aspecto `A3`. Esse método invoca o método `C2.method1()`, porém o método `method1()` é herdado por `C2` por meio de uma alteração de hierarquia de herança feita pelo aspecto `A2`, que torna `C1` uma superclasse de `C2`. Portanto, uma dependência do tipo "As" entre `A3` e `A2` é mapeada no AORD da Figura 5.26(b). Além disso, o método `method1()` também não é implementado em `C1`, ele é introduzido pelo aspecto `A1`. Dessa forma, uma dependência do tipo "As" ligando `A3` a `A1` também é mapeada no AORD da Figura 5.26(b). O mapeamento da dependência que liga `A3` a `A1` é um exemplo do caso complexo 1.1 e o mapeamento da dependência que liga `A3` a `A2` é um exemplo do caso complexo 1.2. É possível concluir que para que o método `method3()` introduzido pelo aspecto `A3` possa ser executado, ele precisa que o método `method1()` seja introduzido por `A1` em `C1`, que por sua vez deve ser herdado por `C2` por meio de uma alteração de hierarquia de herança feita por `A2`.

5.4 Aplicação da estratégia Conjunta no AORD construído

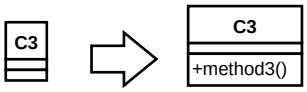
A ordem de implementação e teste de classes e aspectos segundo o AORD da Figura 5.21 e a estratégia Conjunta é mostrada na Tabela 5.1. Conforme pode ser observado na Figura 5.27, existe apenas um SCC no AORD final. Nesse exemplo é necessário implementar apenas um



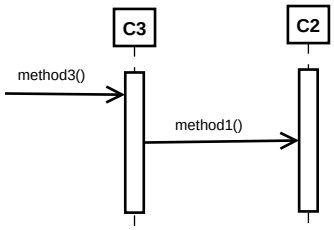
(a) Regra RC7 de introdução do método `method1()` feita pelo aspecto `A1`.



(b) Regra RC8 de alteração de hierarquia de herança feita pelo aspecto `A2`, em que `C2` se torna uma especialização de `C1`.

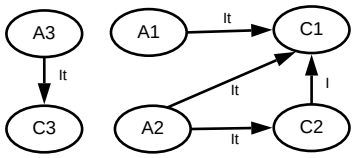


(c) Regra RC9 de introdução do método `method3()` feita pelo aspecto `A3`.

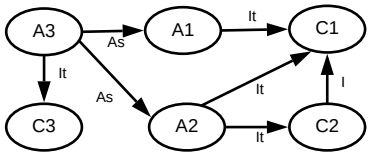


(d) Diagrama de sequência que descreve o método `method3()`.

Figura 5.25: Exemplo de combinação de casos complexos em regras de transformação e diagramas de sequência.



(a) AORD parcial com as dependências das Regras RC7, RC8 e RC9.



(b) AORD final com as dependências mapeadas a partir dos diagramas mostrados na Figura 5.25.

Figura 5.26: Construção e um AORD a partir dos diagramas mostrados na Figura 5.25.

stub, da classe `Call` para o teste da classe `Customer`, como mostra o cálculo para a remoção temporária de arestas para quebrar o SCC:

`Call-Customer`	`Call-Connection`
$1 * 1 = 1$	$1 * 1 = 1$
`Customer-Call`	`Connection-Customer`
$2 * 4 = 8$	$3 * 1 = 3$
`Call-Local`	`LongDistance-Connection`
$1 * 1 = 1$	$1 * 1 = 1$
`Call-LongDistance`	`Local-Connection`
$1 * 1 = 1$	$1 * 1 = 1$

Tabela 5.1: Ordem de implementação e teste do sistema Telecom.

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

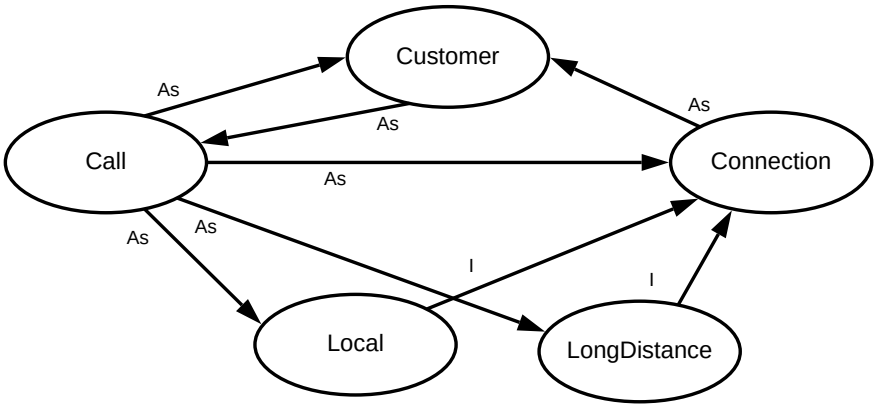


Figura 5.27: SCC encontrado no AORD final.

5.5 Restrições encontradas durante o mapeamento

As regras de mapeamento e sua aplicação mostradas neste capítulo sofrem a influência de algumas restrições. A primeira delas é que um mesmo modelo de projeto pode dar origem a diferentes implementações, pela própria abstração de detalhes inerente dessa fase do desenvolvimento de software. Assim, o AORD construído a partir de um modelo de projeto provavelmente será diferente de um AORD proveniente de engenharia reversa feita com base no código fonte, ou *bytecode* Java. Como exemplo, pode-se considerar o uso de aspectos abstratos que promovem reúso, não previstos antecipadamente em um modelo de projeto; o reúso de conjuntos de junção por diferentes aspectos; ou ainda, dependências do tipo “S”, de relaxamento de exceção, que é um problema basicamente da fase de implementação. Assim, o AORD pode ser refinado ao longo do desenvolvimento do produto de software, possibilitando que o algoritmo de ordenação possa ser reaplicado e um resultado mais fiel à implementação possa ser obtido.

Outra restrição encontrada é a própria imaturidade das técnicas de modelagem de programas OA. Existem muitas propostas nesse sentido, mas todas sofrem com problemas ainda não resolvidos pela comunidade científica. Como exemplo específico, na notação MATA os conjuntos de junção não possuem nome e não são representados explicitamente. O código fonte AspectJ referente aos conjuntos de junção é gerado de maneira automatizada por meio da aplicação das regras de transformação que capturam os pontos de junção. Com isso, não existe possibilidade de, no modelo de projeto, referenciar os conjuntos de junção. Portanto, não é possível, encontrar e mapear dependências do tipo "U" – dependência que ocorre quando um aspecto usa um conjunto de junção definido em outro aspecto – devido a impossibilidade de expressar conjuntos de junção diretamente nas regras de transformação. A notação MATA não dá suporte ao relaxamento de exceção e, assim, não se pode mapear dependências do tipo "S".

As restrições discutidas juntamente com a decisão de não mostrar alguns detalhes de persistência explica porque o AORD mostrado nesse capítulo é diferente do AORD usado na condução dos experimentos apresentados no Capítulo 6. No Capítulo 6 pode-se observar que existem várias dependências mapeadas referentes ao interesse de persistência, bem como uma dependência do tipo "U".

5.6 Considerações finais

O objetivo desse capítulo foi validar o AORD proposto, mostrando que sua construção é viável a partir de um modelo de projeto. Para isso foi utilizada a modelagem baseada em interesses do sistema Telecom usando a notação MATA. O processo de mapeamento mostra como utilizar os diagramas da notação MATA para produzir um AORD, ficando limitado em certas situações pelas restrições da notação. Um ponto importante a ser observado é que o processo de mapeamento apoia o uso de *frameworks*, OO ou transversais. A modelagem deve abranger os pontos de acoplamento para que os passos de processo identifiquem as dependências aspectuais.

O AORD construído pelo processo de mapeamento é tão preciso quanto a disponibilidade e corretude de artefatos de projeto. Além disso, ele pode ser aplicado recorrentemente ao longo do processo de desenvolvimento de software à medida que alterações são feitas nos modelos e o nível de abstração se aproxima dos detalhes de implementação.

CAPÍTULO

6

Estudos de caracterização das duas estratégias de ordenação propostas

6.1 Considerações iniciais

O principal objetivo deste estudo exploratório foi analisar o custo de criação de *stubs* para o teste de integração de três sistemas implementados com AspectJ do ponto de vista dos desenvolvedores e testadores. O estudo foi feito utilizando diferentes estratégias de ordenação e determina qual estratégia é menos custosa para cada um dos sistemas estudados. Segundo Briand *et al.* (2003) e Kung *et al.* (1995a), o número de *stubs* é um bom indicador para determinar a melhor estratégia de ordenação. Além do número de *stubs* realmente implementados, foram coletados também o número de métodos, de atributos, de conjuntos de junção, de adendos, de declarações de relaxamento de exceção e de declarações intertipos de cada *stub*.

Adicionalmente, além de determinar a estratégia de ordenação menos custosa, o estudo é usado para validar o AORD e o modelo de dependências aspectuais proposto na Seção 4.2.1. Com o estudo é possível verificar se o AORD mostra corretamente e com precisão as dependências entre classes e aspectos. O estudo ainda tem um segundo objetivo adicional, coletar informações e obter amostras da implementação de *stubs* de aspectos de *stubs* para testar aspectos, uma vez que não foram encontrados na literatura especializada diretrizes ou padrões de implementação de *stubs* desse tipo.

Na Seção 6.2 é apresentada a descrição dos estudos, que contém informações sobre as estratégias utilizadas e detalhes das etapas e tarefas executadas durante sua condução. A condução dos estudos juntamente com os resultados registrados para cada sistema utilizado são apresentados na Seção 6.3. Ainda nessa seção, os resultados obtidos em cada sistema são analisados e discutidos. As conclusões gerais envolvendo todo o estudo são apresentadas na Seção 6.4. Por fim, na Seção 6.5, são apresentadas as considerações finais deste capítulo.

6.2 Descrição dos estudos de caracterização

As duas estratégias de ordenação propostas na Seção 4.3 foram aplicadas a três sistemas para produção das ordens de teste e integração usadas nos estudos. Além delas, como controle comparativo dos estudos, foram usadas mais duas ordenações: uma produzida pela estratégia Aleatória, na qual a ordem de teste e integração das classes e aspectos foi escolhida aleatoriamente por sorteio; e uma produzida pela estratégia Reversa, em que a ordem de teste e integração de classes e aspectos é inversa àquela produzida pela estratégia conjunta. Os estudos foram conduzidos em duas etapas para cada sistema. Na primeira etapa, os AORDs foram construídos a partir de engenharia reversa de código fonte. No caso da estratégia Incremental+, foram construídos um AORD apenas com classes e um AORD apenas com aspectos, e no caso da estratégia Conjunta, foi construído um AORD com classes e aspectos. A estratégia aleatória não usa um AORD e a estratégia Reversa usa o mesmo AORD da estratégia Conjunta. Na segunda etapa, os sistemas foram testados e integrados de acordo com as estratégias de ordenação: Incremental+, Conjunta, Aleatória e Reversa.

Os estudos foram feitos com base em duas linhas de raciocínio. Na primeira linha as estratégias de ordenação Aleatória e Reversa têm custo de implementação maior que as outras duas estratégias. Na segunda linha, a estratégia Conjunta tem custo de implementação menor que a estratégia Incremental+. A primeira linha de raciocínio é trivial, uma vez que testar em ordem reversa quer dizer iniciar o teste pelos módulos que possuem o maior número de dependências. O teste na ordem aleatória, por sua vez, equivale a uma ordenação *ad hoc*, desaconselhado pela grande maioria dos trabalhos em teste de software. A segunda linha de raciocínio é mais sutil, e baseia-se na ideia de que testar classes sem considerar suas dependências pode remover indistintamente dependências e quebrar ciclos de maneira não otimizada. Por outro lado, considerar as dependências e buscar uma ordem de teste e integração que considere as dependências conjuntamente é, intuitivamente, uma solução melhor porque usa as dependências como critério para a remoção de arestas. A medição do custo nas duas linhas é feito pelo número de *stubs* implementados. É importante ressaltar que são utilizados nos estudos *stubs* específicos, conforme a definição dada por Briand *et al.* (2003). Portanto, foram coletados o número de membros internos de cada *stub* especificamente para o teste de uma classe ou aspecto.

Além de serem usadas como controle, a estratégia Reversa e a estratégia Aleatória foram incluídas nos estudos para que fosse possível analisar os *stubs* implementados. Como essas duas estratégias, em princípio, implicam na implementação de um número maior de *stubs* que as outras, amostras de implementação de *stubs* usados durante a condução dos estudos podem ser coletadas. Posteriormente, as amostras podem ser analisadas para buscar na implementação

de *stubs* de aspectos e *stubs* de classes para testar aspectos padrões de implementação que auxiliam o desenvolver e o testador.

O número de passos de teste de cada estratégia é igual à soma do número de classes e aspectos que o sistema possui. A classe ou aspecto testado e integrado em um determinado passo é denominado módulo em teste. Para cada um dos passos foram executadas seis tarefas:

1 o módulo em teste de acordo com a estratégia escolhida foi adicionado à configuração de teste, isto é, os módulos já testados e integrados, os *stubs* e os casos de teste usados em passos anteriores;

2 os *stubs* necessários foram implementados e, depois de resolvidos possíveis conflitos entre eles e aqueles presentes na configuração de teste, foram adicionados à configuração de teste;

3 os casos de teste foram implementados em JUnit;

4 o módulo em teste foi testado e integrado juntamente com seus *stubs* específicos;

5 os resultados foram analisados;

6 os dados a respeito dos *stubs* foram coletados.

6.3 Condução dos estudos nos sistemas Telecom, de comércio e MobileMedia

6.3.1 Estudo usando o sistema Telecom

Esta parte do estudo foi conduzida usando o sistema Telecom descrito na Seção 4.2.3.1. O AORD mostrado na Figura 6.1 referente ao sistema Telecom é ligeiramente diferente dos AORDs apresentados anteriormente. Isso acontece porque detalhes do *framework* de persistência omitidos naquela descrição foram considerados. Além disso, o sistema Telecom foi alterado em relação ao originalmente proposto por The AspectJ Team (2002). Foi adicionada a funcionalidade de listar chamadas telefônicas de clientes já persistidas e o aspecto `Billing`, responsável pelo interesse de tarifação, determina também quando uma chamada telefônica deve ser temporizada. A funcionalidade de listar chamadas persistidas foi incluída para que haja maior interação entre o *framework* de persistência, as classes-base e os aspectos. A alteração no aspecto `Billing` foi feita para verificar como o reúso de conjuntos de junção afeta o relacionamento entre os aspecto.

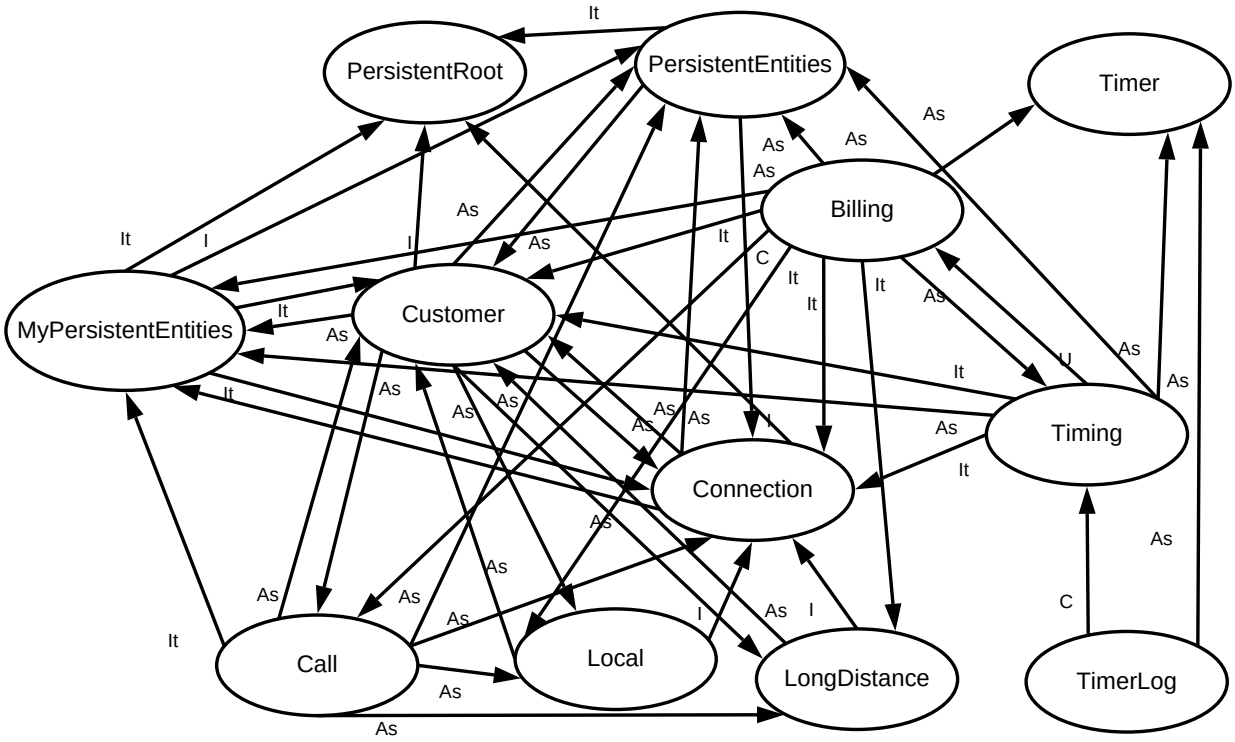


Figura 6.1: AORD com classes e aspectos construído a partir da engenharia reversa do sistema Telecom.

6.3.1.1 Aplicação da estratégia Incremental+

O AORD apresentado na Figura 6.2 mostra classes e aspectos separadamente, de acordo com a estratégia Incremental+. Na figura, os vértices em cinza indicam os aspectos, e os demais, as classes. As dependências no sentido aspecto/classe foram removidas e apenas as dependências no sentido classe/aspecto, denotadas pela seta tracejada, foram deixadas. Isso foi feito para que melhor se possa visualizar os *stubs* que devem ser implementados pela separação de aspectos e classes. Conforme discutido, a separação remove dependências e pode quebrar ciclos de dependência indistintamente. Note-se que as classes `Timer` e `PersistentRoot` estão desconectadas das demais porque se relacionam ou apenas com aspectos ou por meio de aspectos. O algoritmo inclui essas duas classes logo no início da ordenação.

Esperava-se implementar 11 *stubs*, dos quais 4 de classes e 7 de aspectos, durante os testes do sistema Telecom na estratégia Incremental+, calculados pelo algoritmo da estratégia Incremental+. O resultado da condução do estudo usando a estratégia Incremental+ para o sistema Telecom, com a lista completa dos *stubs* realmente implementados e o número de membros internos de cada um, é apresentado na Tabela 6.1. A primeira coluna mostra a ordem de teste e integração, a segunda mostra o nome do módulo em teste e a terceira informa sobre o tipo do módulo: aspecto ou classe. Na quarta coluna da tabela é mostrado o nome do *stub* imple-

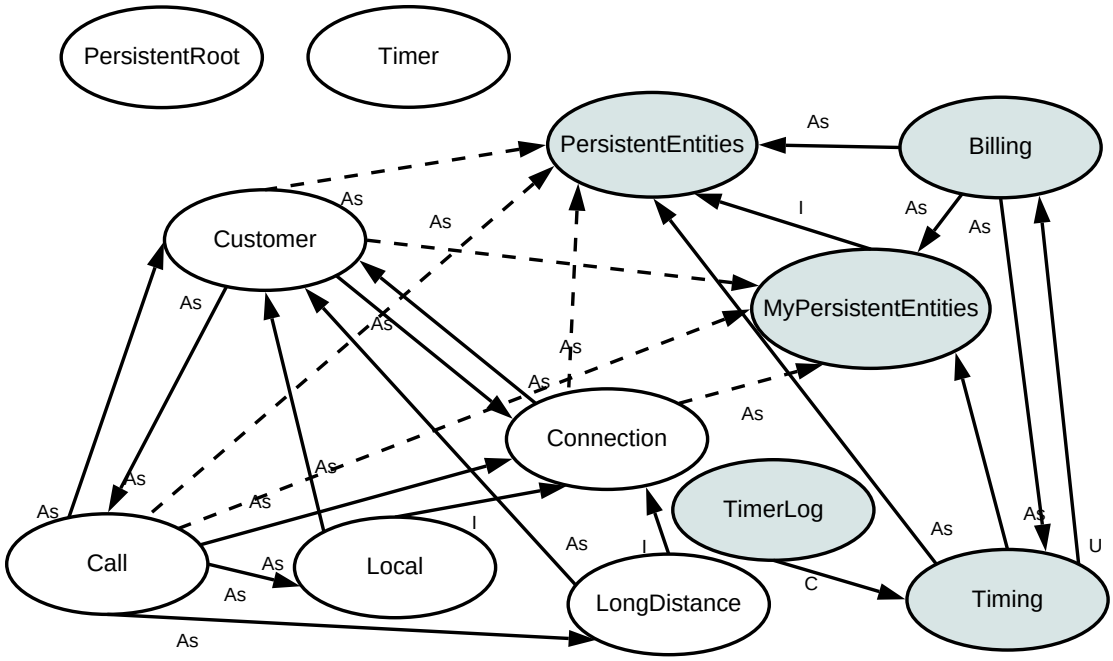


Figura 6.2: AORD referente ao sistema Telecom com classes e aspectos mostrados de maneira separada.

mentado e na quinta o tipo do *stub*: aspecto ou classe. As demais colunas mostram o total de membros internos de cada módulo em teste, mais especificamente o número de: atributos, métodos, adendos, conjuntos de junção, introduções de atributos e métodos e de alterações de hierarquia de herança. Neste ponto do estudo não houve implementação de declarações de relaxamento de exceção em *stubs* de aspectos.

Na última linha da Tabela 6.1 são apresentados o número total de *stubs* implementados e os totais de membros internos. Pode-se constatar que foram realmente implementados 12 *stubs*, sendo 4 de classes e 8 de aspectos, apesar do algoritmo ter calculado que seriam necessários 11 *stubs*. Isso acontece por que classes e aspectos abstratos não são diretamente testáveis. Assim, foi necessário implementar um *stub* não esperado para testar o aspecto abstrato `PersistentEntities`, mostrado no passo 9. Nesse caso houve, portanto, um aumento do número de *stubs* implementados no passo. O teste de um aspecto abstrato é caracterizado por problemas semelhantes ao teste de classes abstratas: não existem instâncias de aspectos abstratos; métodos abstratos não têm implementação; e, diferentemente da programação OO, conjuntos de junção abstratos não entrecortam pontos de junção, precisam ser especializados. É importante observar que o teste de módulos abstratos é um problema da programação OA assim como na programação OO.

Durante a condução do estudo, foram identificados vários padrões de implementação de *stubs* (Ré *et al.*, 2008). Esses padrões são apresentados em detalhes na Seção 7. Os padrões de

Tabela 6.1: Custo da estratégia Incremental+ em número de *stubs* e membros interno no estudo do sistema Telecom.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	PersistentRoot	C	–	–	–	–	–	–	–	–
2	Timer	C	–	–	–	–	–	–	–	–
3	Connection	C	Customer	C	1	2	–	–	–	–
			PersistentEntities	A	0	0	0	0	2	0
			MyPersistentEntities	A	0	1	0	0	0	1
4	Customer	C	Local	C	0	3	–	–	–	–
			LongDistance	C	0	3	–	–	–	–
			Call	C	2	7	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	0
			MyPersistentEntities	A	0	1	0	0	0	2
5	LongDistance	C	–	–	–	–	–	–	–	–
6	Local	C	–	–	–	–	–	–	–	–
7	Call	C	PersistentEntities	A	0	0	0	0	2	0
			MyPersistentEntities	A	0	1	0	0	0	2
8	PersistentEntities	A	MyPersistentEntities	A	0	1	0	0	0	2
9	MyPersistentEntities	A	–	–	–	–	–	–	–	–
10	Billing	A	Timing	A	0	1	0	0	0	0
11	Timing	A	–	–	–	–	–	–	–	–
12	TimerLog	A	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				12	3	20	0	0	5	7

implementação, ou casos de implementação, afetam a condução do estudo porque influenciam no número de *stubs*, uma vez que têm por objetivo facilitar sua implementação. O uso dos casos de implementação frequentemente diminui o número de *stubs* do passo de teste. Assim, são apresentados para cada estratégia os passos nos quais os casos de implementação foram utilizados e os casos de implementação que poderiam ter sido utilizados. Alguns desses casos são de uso facultativo e outros devem ser usados porque não há como conduzir o teste sem aplicá-los. Nos casos em que o uso é facultativo, escolheu-se pela não utilização, para que os resultados dos estudos não sejam perturbados por sua aplicação.

No que se refere ao possível uso de alguns casos de implementação de *stubs*, nos passos 3, 4 e 7 é possível utilizar o padrão de implementação mostrado na Seção 7.7.

6.3.1.2 Aplicação da estratégia Conjunta

Computados por meio da aplicação do algoritmo segundo a estratégia Conjunta, esperava-se implementar 9 *stubs*, sendo 5 *stubs* de classes e 4 *stubs* de aspectos. O resultado da condução do estudo para a estratégia Conjunta no sistema Telecom com os *stubs* realmente implementados são apresentados na Tabela 6.2. Foram realmente implementados nessa estratégia 10 *stubs*,

sendo 5 de classes e 5 de aspectos. Assim como na estratégia anterior, foi necessário um *stub* a mais para o teste do aspecto abstrato `PersistentEntities`, no passo 3.

Com relação ao uso de casos de implementação, nos passos 2 e 3 dessa estratégia é possível utilizar o caso de implementação de *stubs* apresentado na Seção 7.7.

Tabela 6.2: Custo da estratégia Conjunta em número de *stubs* e membros internos no estudo do sistema Telecom.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	`PersistentRoot`	C	–	–	–	–	–	–	–	–
2	`Connection`	C	`Customer`	C	1	3	–	–	–	–
			`PersistentEntities`	A	0	0	0	0	1	0
			`MyPersistentEntities`	A	0	1	0	0	0	1
3	`PersistentEntities`	A	`MyPersistentEntities`	A	0	1	0	0	0	2
			`Customer`	C	3	6	0	0	0	0
4	`Customer`	C	`Call`	C	2	7	–	–	–	–
			`Local`	C	0	1	–	–	–	–
			`LongDistance`	C	0	1	–	–	–	–
			`MyPersistentEntities`	A	0	1	0	0	0	1
5	`LongDistance`	C	–	–	–	–	–	–	–	–
6	`Local`	C	–	–	–	–	–	–	–	–
7	`MyPersistentEntities`	A	–	–	–	–	–	–	–	–
8	`Call`	C	–	–	–	–	–	–	–	–
9	`Timer`	C	–	–	–	–	–	–	–	–
10	`Billing`	A	`Timing`	A	0	1	0	0	0	0
11	`Timing`	A	–	–	–	–	–	–	–	–
12	`TimerLog`	A	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				10	6	22	0	0	1	4

Um ponto importante a observar é que os *stubs* da mesma classe `Customer` nos passos 2 e 3 são diferentes. Pode-se perceber que existe um número diferente de membros internos. Isso acontece porque os *stubs* são específicos para o teste de um módulo, ou seja, específicos para o passo de teste. Note-se que no passo 2 já existe um *stub* de `Customer` na configuração de teste. Houve uma análise para decidir como implementar métodos e atributos da classe `Customer` específicos para o passo de teste, levando em consideração a implementação do *stub* da classe `Customer` presente na configuração de teste. Isso é chamado de conflito entre *stubs* e, nesse caso, existe retrabalho para implementar o novo *stub* de maneira que seja compatível com o comportamento simulado por outros *stubs* da configuração de teste. Os conflitos aconteceram com praticamente todas as estratégias e sistemas dos estudos, com incidência notadamente maior nas estratégias Aleatória e Reversa.

6.3.1.3 Aplicação da estratégia Aleatória

A ordem de teste dos módulos na estratégia Aleatória foi escolhida por sorteio. O nome de cada módulo a ser testado foi colocado em uma urna e, posteriormente, retirados um a um. A ordem de teste foi dada pela ordem de retirada dos nomes dos módulos da urna. Eram esperados 10 *stubs* de classes e 9 *stubs* de aspectos, totalizando 19 *stubs*, calculados por meio da análise do AORD do sistema Telecom e da ordem sorteada. O resultado da condução do estudo com a estratégia Aleatória aplicada ao sistema Telecom juntamente com a ordem de teste é mostrado na Tabela 6.3. Diferemente do esperado, o número de *stubs* realmente implementados foi 20, 10 *stubs* de classes e 10 *stubs* de aspectos. Novamente, foi necessário um *stub* para o teste do aspecto abstrato `PersistentEntities`. Além disso, no passo 4 foi necessário utilizar o caso de implementação de *stubs* mostrado na Seção 7.4.

No que se refere aos possíveis usos de casos de implementação, nos passos 2, 4 e 5 pode-se utilizar o caso de implementação de *stubs* mostrado na Seção 7.7 e nos passos 4 e 6 é possível usar o caso de implementação de *stubs* apresentado na Seção 7.3.1.

Tabela 6.3: Custo da estratégia Aleatória em número de *stubs* e membros internos no estudo do sistema Telecom.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#C.I.	#Introdução	#Alt. Hierar. de Herança
1	`Timer`	C	–	–	–	–	–	–	–	–
2	`Connection`	C	`Customer`	C	1	2	–	–	–	–
			`PersistentEntities`	A	0	0	0	0	2	0
			`MyPersistentEntities`	A	0	1	0	0	0	1
			`PersistentRoot`	C	0	0	–	–	–	–
3	`Local`	C	`Customer`	C	0	0	–	–	–	–
4	`Billing`	A	`Customer`	C	1	2	–	–	–	–
			`LongDistance`	C	0	0	–	–	–	–
			`Timing`	A	0	1	0	0	0	0
			`PersistentEntities`	A	0	0	0	0	2	0
			`MyPersistentEntities`	A	0	1	0	0	0	1
			`Call`	C	0	2	0	0	0	0
5	`Customer`	C	`Call`	C	2	7	–	–	–	–
			`LongDistance`	C	0	3	–	–	–	–
			`PersistentEntities`	A	0	0	0	0	1	0
			`MyPersistentEntities`	A	0	1	0	0	0	2
			`PersistentRoot`	C	0	0	–	–	–	–
6	`TimerLog`	C	`Timing`	A	0	0	2	0	0	0
7	`PersistentRoot`	C	–	–	–	–	–	–	–	–
8	`PersistentEntities`	A	`MyPersistentEntities`	A	0	1	0	0	0	2
9	`Timing`	A	`MyPersistentEntities`	A	0	1	0	0	0	2
10	`MyPersistentEntities`	A	–	–	–	–	–	–	–	–
11	`Call`	C	`LongDistance`	C	0	2	–	–	–	–
12	`LongDistance`	C	–	–	–	–		–	–	–
Total de *stubs*/membros internos				20	4	24	2	0	5	8

6.3.1.4 Aplicação da estratégia Reversa

Na Tabela 6.4 é apresentada a ordem de teste inversa àquela da estratégia Conjunta aplicada ao sistema Telecom. Esperava-se implementar para a estratégia Reversa 36 *stubs*, sendo 26 *stubs* de classes e 10 *stubs* de aspectos, computados por meio da análise do AORD do sistema Telecom. O resultado da condução do estudo usando a estratégia Reversa aplicada ao sistema Telecom é mostrado na Tabela 6.4. O número de *stubs* realmente implementados foi 39, sendo 29 de classes e 10 de aspectos. O número de *stubs* de classes foi maior porque foi necessário implementar três *stubs* da classe `PersistentRoot` não esperados, nos passos 2, 3 e 5. Isso acontece porque o modelo de dependências aspectuais não permite que dependências referentes à declarações intertipos sejam removidas, o que foi feito na estratégia Reversa. No passo 2 foi necessário usar o caso apresentado na Seção 7.5.

Quanto à possibilidade do uso de casos de implementação de *stubs*, nos passos 2, 3 e 5 pode-se utilizar o caso de implementação mostrado na Seção 7.7 e nos passos 1 e 3 o caso de implementação mostrado na Seção 7.3.1. Nos passos 2 e 5 também é possível usar o caso de implementação mostrado Seção 7.4.

6.3.1.5 Discussão e análise dos resultados obtidos

A discussão feita aqui refere-se apenas aos resultados obtidos para o sistema Telecom. De maneira análoga, existe uma seção para cada sistema estudado que discute seus resultados específicos. Conclusões gerais sobre o estudo são apresentados na Seção 6.4.

Na Tabela 6.5, é apresentado um resumo dos resultados dos estudos. Na primeira coluna é mostrada a estratégia utilizada, na segunda o número de *stubs* de classes, na terceira o número de *stubs* de aspectos e na quarta o número total de *stubs* da estratégia. Da coluna 5 até a coluna 10 são listados os números de membros internos dos *stubs*, a saber, o número de: atributos, métodos, adendos, conjuntos de junção, introduções de atributos e métodos e alterações de hierarquia de herança. A última coluna contabiliza o número total de membros internos dos *stubs*.

Conforme pode-se observar na Tabela 6.5, a primeira linha de raciocínio foi confirmada no estudo do sistema Telecom. As estratégias Aleatória e Reversa obtiveram resultados piores que as estratégias Conjunta e Incremental+. O custo da estratégia Reversa foi bem maior que nas outras três. Esse fato pode ser explicado pela grande quantidade de dependências que os aspectos têm das classes e, eventualmente, de outros aspectos. Assim, como os aspectos foram integrados primeiro, vários *stubs* foram necessários. Dessa forma, confirma-se a intuição de que iniciar o teste de integração pelos aspectos não é uma boa solução.

Tabela 6.4: Custo da estratégia Reversa em número de *stubs* e membros internos no estudo do sistema Telecom.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	TimerLog	A	Timer	C	2	2	–	–	–	–
			Timing	A	0	0	2	0	0	0
2	Timing	A	Connection	C	2	4	–	–	–	–
			Customer	C	0	1	–	–	–	–
			Timer	C	2	3	–	–	–	–
			Billing	A	0	0	2	2	0	0
			PersistentRoot	C	0	0	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	
			MyPersistentEntities	A	0	1	0	0	0	2
3	Billing	A	Connection	C	2	4	–	–	–	–
			Customer	C	1	4	–	–	–	–
			Local	C	0	0	–	–	–	–
			LongDistance	C	0	0	–	–	–	–
			Timer	C	2	3	–	–	–	–
			PersistentRoot	A	0	0	–	–	–	–
			PersistentEntities	C	0	0	0	0	1	
			MyPersistentEntities	A	0	1	0	0	0	2
			Call	C	0	2	0	0	0	0
4	Timer	C	–	–	–	–	–	–	–	–
5	Call	C	Connection	C	3	6	–	–	–	–
			Customer	C	2	4	–	–	–	–
			Local	C	0	3	–	–	–	–
			LongDistance	C	0	3	–	–	–	–
			PersistentRoot	C	0	0	–	–	–	–
			PersistentEntities	A	0	0	0	0	2	
			MyPersistentEntities	A	0	1	0	0	0	2
6	MyPersistentEntities	A	Connection	C	0	0	–	–	–	–
			Customer	C	0	0	–	–	–	–
			PersistentRoot	C	0	1	–	–	–	–
			PersistentEntities	A	0	0	0	0	0	0
7	Local	C	Connection	C	0	1	–	–	–	–
			Customer	C	0	0	–	–	–	–
8	LongDistance	C	Connection	C	0	1	–	–	–	–
			Customer	C	0	0	–	–	–	–
9	Customer	C	PersistentRoot	C	0	0	–	–	–	–
			Connection	C	0	1	–	–	–	–
			PersistentEntities	A	0	0	0	0	0	1
10	PersistentEntities	A	Connection	C	4	8	–	–	–	–
			PersistentRoot	C	0	0	–	–	–	–
11	Connection	C	PersistentRoot	C	0	0	–	–	–	–
12	PersistentRoot	C	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				39	20	54	2	2	4	7

A segunda linha de raciocínio também foi confirmada, e a estratégia Conjunta obteve resultado melhor que estratégia Incremental+. É interessante notar que existe a tendência de balanceamento entre o número de *stubs* de aspectos e classes na estratégia Conjunta. Em contrapartida, a estratégia Incremental+ mostra uma tendência de implementação maior de número de *stubs* de aspectos do que de classes. Coincidentemente a estratégia Aleatória seguiu a tendência da estratégia Conjunta e a estratégia Reversa seguiu a tendência da estratégia Incremental+.

Um ponto importante a ser enfatizado é que o número de membros internos implementados na estratégia Conjunta foi menor do que na estratégia Incremental+. Isso acontece mesmo com as características do *framework* de persistência. O *framework* de persistência precisa de métodos de manipulação (*get* e *set*) para manipular atributos de objetos persistidos. Com isso, ao efetuar o teste de integração do *framework* com as classes, é necessário simular nos *stubs* todos os atributos e métodos de manipulação. Apesar do número de métodos e atributos simulados ser relativamente grande, sua implementação é bastante trivial e de baixo custo.

Tabela 6.5: Custo de criação dos *stubs* no estudo do sistema Telecom.

Estratégia	#*Stub* de classe	#*Stub* de aspecto	#Total de *stubs*	#Atr.	#Met.	#Ad.	#CJ.	#Intr.	#AHH.	#Membros internos
Incremental+	4	8	**12**	3	20	0	0	5	7	**35**
Conjunta	5	5	**10**	6	22	0	0	1	4	**33**
Aleatória	10	10	**20**	4	24	2	0	5	8	**43**
Reversa	26	13	**39**	20	54	2	2	4	7	**89**

6.3.2 Estudo usando o sistema de comércio

O sistema de comércio usado no estudo foi implementado com base na descrição dada por Cibrán *et al.* (2003). Ele é composto por 9 classes e 9 aspectos. A proposta de Cibrán *et al.* (2003) mostra como implementar 3 regras de negócio por meio de aspectos: uma regra de negócio para estabelecer quando um cliente é considerado cliente frequente; uma regra de negócio em que clientes frequentes recebem descontos na compra de produtos; e uma regra de negócio em que descontos são dados em compras de natal. Vale ressaltar que a arquitetura do sistema implementado para o estudo segue a proposta de Cibrán *et al.* (2003). Além disso, os aspectos utilizam código fonte listado no trabalho. Para proporcionar a funcionalidade de persistência no sistema de comércio, foi acoplado o mesmo *framework* de persistência usado no estudo do sistema Telecom.

Na Tabela 6.6 são mostradas as dependências do AORD completo. As linhas da tabela mostram classes e aspectos que dependem de classes e aspectos representados nas colunas. A representação do AORD em forma de tabela é mais compacta e torna mais fácil a compreensão das dependências do sistema, uma vez que ele possui mais classes e aspectos que o sistema Telecom.

6.3.2.1 Aplicação da estratégia Incremental+

Esperava-se implementar 15 *stubs*, sendo 2 *stubs* de classes e 13 *stubs* de aspectos, na estratégia Incremental+, computados por meio da aplicação do algoritmo segundo a estratégia. Na Tabela 6.7 é apresentado o resultado do estudo, mostrando os *stubs* realmente implementados. Ao

Tabela 6.6: Dependências do AORD do sistema de comércio.

	Product	ShoppingCart	Store	Customer	BRPriceDiscount	BRFrequentDiscount	BRFrequentCustomer	PersistentRoot	BRChristmasDiscount	PersistentEntities	MyPersistentEntities	EPricePersonalisation	ExtendCustomer	CaptureCustomer	ApplyFrequentDiscount	ApplyChristmasDiscount	ECheckout	ApplyFrequentCustomer
Product			As					I		As	As							
ShoppingCart	As							I		As	As							
Store	As			As				I		As	As							
Customer	As	As	As					I		As	As							
BRPriceDiscount		As																
BRFrequentDiscount				As	I								As					
BRFrequentCustomer				As						As	As		As					
PersistentRoot																		
BRChristmasDiscount					I													
PersistentEntities	As	As	As	As				It										
MyPersistentEntities	It	It	It	It				It		I								
EPricePersonalisation	As	C	C	As														
ExtendCustomer				It														
CaptureCustomer			C	As		As									As			
ApplyFrequentDiscount	As			As		As						U		U				
ApplyChristmasDiscount	As								As			U			As			
ECheckout			C	As														
ApplyFrequentCustomer				As			As										U	

invés de 15 *stubs*, foram implementados 19, sendo 3 de classes e 16 de aspectos. Foi necessário implementar um *stub* não esperado para o teste do aspecto abstrato `PersistentEntities`, que pode ser visto no passo 10. Da mesma forma, foi necessário implementar também um *stub* de classe não esperado para testar a classe abstrata `BRPriceDiscount`, no passo 6.

O segundo e o terceiro *stubs* de aspecto não esperados foram implementados nos passos 12 e 17. Isso acontece porque os aspectos em teste não possuem adendos, possuem apenas conjuntos de junção. Os aspectos `EPricePersonalisation` e `ECheckout` seguem a proposta de implementação de regras de negócio de Cibrán *et al.* (2003). Segundo a proposta, conjuntos de junção devem ser implementados em separado para promover seu reúso. Assim, os conjuntos de junção, e consequentemente os pontos de junção por ele capturados, ficam expostos. Percebe-se aqui o interesse em deixar clara as interfaces de entrecorte do sistema, seguindo a ideia das XPIs de Griswold *et al.* (2006). Um exemplo pode ser visto na Listagem 6.1, em que estão presentes os aspectos `ECheckout` e `ApplyFrequentCustomer`. O aspecto `ECheckout` expõe o ponto de junção a ser capturado por meio do conjunto de junção `checkout`, mostrado na linha 2, e o aspecto `ApplyFrequentCustomer` utiliza o conjunto de junção, mostrado na linha 11, para aplicar uma regra de negócio. O problema é que, para testar um conjunto de junção é necessário um adendo que o utilize. Por isso deve-se usar o caso de implementação

apresentado na Seção 7.6. Note-se que, como a granularidade do AORD deixa claro as dependências entre classes e aspectos e não entre seus membros internos, o algoritmo não detecta aspectos apenas com conjuntos de junção.

Listagem 6.1: Código fonte dos aspectos `ECheckout` e `ApplyFrequentCustomer`.

```
1  public aspect ECheckout {
2      pointcut checkout(Customer aCustomer): args(aCustomer) &&
3          execution(float Store.checkOut(Customer));
4  }
5
6  public aspect ApplyFrequentCustomer {
7      static BRFrequentCustomer businessRule;
8      static public void setBR(BRFrequentCustomer br){
9          businessRule = br;
10     }
11     after(Customer aCustomer): ECheckout.checkout(aCustomer){
12         businessRule.apply(aCustomer);
13     }
14 }
```

No passo de teste 15 foi usado o caso de implementação apresentado na Seção 7.8. O teste do aspecto `ApplyFrequentDiscount` só pode ser feito por meio da utilização do caso de implementação porque ele é um aspecto não *singleton* e *stateful*, cujo modelo de instanciação determina sua associação a um fluxo de execução de métodos. A utilização desse caso de implementação para o teste do aspecto `ApplyFrequentDiscount` é feita em todas as estratégias do estudo do sistema de comércio. Além desses casos de implementação aplicados, nos passos 2, 3, 4, 5 e 8 é possível utilizar o caso de implementação mostrado na Seção 7.7.

6.3.2.2 Aplicação da estratégia Conjunta

Na estratégia conjunta esperava-se implementar 11 *stubs*, sendo 6 *stubs* de classes e 5 *stubs* de aspectos, computados por meio da aplicação do algoritmo segundo a estratégia. Na Tabela 6.8 é apresentado o resultado do estudo, mostrando os *stubs* realmente implementados. Ao invés de 11 *stubs*, foram implementados 15, sendo 7 de classes e 8 de aspectos. O aumento do número de *stubs* realmente implementados ocorreu por causa dos mesmos casos mostrados na estratégia Incremental+: dois *stubs*, um de classe e um de aspecto, para testar um aspecto e uma classe abstrata, nos passos 8 e 2; e dois *stubs* de aspectos para testar aspectos que possuem apenas conjuntos de junção, nos passos 10 e 16.

6.3.2.3 Aplicação da estratégia Aleatória

Esperava-se implementar na estratégia Aleatória 35 *stubs*, sendo 26 *stubs* de classes e 7 *stubs* de aspectos, calculados a partir da análise da Tabela 6.6. O resultado do estudo é mostrado na

Tabela 6.7: Custo da estratégia Incremental+ em número de *stubs* e membros internos no estudo do sistema de comércio.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	PersistentRoot	C	–	–	–	–	–	–	–	–
2	Product	C	Store	C	0	1	–	–	–	–
			PersistentEntities	A	0	0	0	0	3	0
			MyPersistentEntities	A	0	1	0	0	0	1
3	ShoppingCart	C	PersistentEntities	A	0	0	0	0	1	0
			MyPersistentEntities	A	0	1	0	0	0	1
4	Store	C	Customer	C	2	10	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	0
			MyPersistentEntities	A	0	1	0	0	0	2
5	Customer	C	PersistentEntities	A	0	0	0	0	3	0
			MyPersistentEntities	A	0	1	0	0	0	1
6	BRPriceDiscount	C	BRChristmasDiscount	C	1	2	–	–	–	–
7	BRFrequentDiscount	C	ExtendCustomer	A	0	0	0	0	1	0
8	BRFrequentCustomer	C	ExtendCustomer	A	0	0	0	0	1	0
			PersistentEntities	A	0	0	0	0	1	0
			MyPersistentEntities	A	0	1	0	0	0	1
9	BRChristmasDiscount	C	–	–	–	–	–	–	–	–
10	PersistentEntities	A	MyPersistentEntities	A	0	1	0	0	0	4
11	MyPersistentEntities	A	–	–	–	–	–	–	–	–
12	EPricePersonalisation	A	TestAspect	A	0	0	2	0	0	0
13	ExtendCustomer	A	–	–	–	–	–	–	–	–
14	CaptureCustomer	A	ApplyFrequentDiscount	A	1	1	0	0	0	0
15	ApplyFrequentDiscount	A	–	–	–	–	–	–	–	–
16	ApplyChristmasDiscount	A	–	–	–	–	–	–	–	–
17	ECheckout	A	TestAspect	A	0	0	1	0	0	0
18	ApplyFrequentCustomer	A	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				19	4	20	3	0	11	10

Tabela 6.9, no qual pode-se constatar que o número de *stubs* realmente implementados foi 36, sendo 8 de aspectos e 28 de classes. Dois dos três *stubs* implementados que não eram esperados foram para o teste de uma classe abstrata, no passo 10, e para um aspecto apenas com conjuntos de junção, no passo 4. O outro *stub* implementado que não era esperado foi necessário porque o modelo de dependências aspectuais não permite que dependências referentes a declarações intertipos sejam removidas, o que foi feito nesta estratégia. Assim, um *stub* da classe `PersistentRoot` foi implementado no passo 6.

Nos passos 4 e 16 foi necessário a utilização do caso de implementação apresentado na Seção 7.3. No passo 2 foi necessário o uso do caso de implementação de *stubs* mostrado na Seção 7.4. Com relação ao possível uso de mais alguns casos de implementação e nos passos 5, 6 e 8 é possível a utilização do caso de implementação mostrado na Seção 7.7.

Tabela 6.8: Custo da estratégia Conjunta em número de *stubs* e membros internos no estudo do sistema de comércio.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	PersistentRoot	C	–	–	–	–	–	–	–	–
2	PersistentEntities	A	MyPersistentEntities	A	0	1	0	0	0	4
			Product	C	3	6	–	–	–	–
			ShoppingCart	C	2	4	–	–	–	–
			Store	C	4	8	–	–	–	–
			Customer	C	3	6	–	–	–	–
3	Product	C	Store	C	0	1	–	–	–	–
			MyPersistentEntities	A	0	1	0	0	0	1
4	ShoppingCart	C	MyPersistentEntities	A	0	1	0	0	0	1
5	Store	C	Customer	C	2	10	–	–	–	–
			MyPersistentEntities	A	0	1	0	0	0	2
6	Customer	C	MyPersistentEntities	A	0	1	0	0	0	1
7	MyPersistentEntities	A	–	–	–	–	–	–	–	–
8	BRPriceDiscount	C	BRChristmasDiscount	C	1	2	–	–	–	–
9	BRChristmasDiscount	C	–	–	–	–	–	–	–	–
10	EPricePersonalisation	A	TestAspect	A	0	0	2	0	0	0
11	ExtendCustomer	A	–	–	–	–	–	–	–	–
12	BRFrequentDiscount	A	–	–	–	–	–	–	–	–
13	CaptureCustomer	A	ApplyFrequentDiscount	A	1	1	0	0	0	0
14	ApplyFrequentDiscount	A	–	–	–	–	–	–	–	–
15	ApplyChristmasDiscount	A	–	–	–	–	–	–	–	–
16	ECheckout	A	TestAspect	A	0	0	1	0	0	0
17	BRFrequentCustomer	C	–	–	–	–	–	–	–	–
18	ApplyFrequentCustomer	A	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				15	16	43	3	0	0	9

6.3.2.4 Aplicação da estratégia Reversa

O número de *stubs* que esperava-se implementar na estratégia Reversa é 48, sendo 35 *stubs* de classes e 10 *stubs* de aspectos, calculados a partir da análise do AORD representado na Tabela 6.6. O resultado do estudo é mostrado na Tabela 6.10, no qual pode-se constatar que o número de *stubs* realmente implementados foi 50, sendo 37 de classes e 13 de aspectos. Um dos *stubs*, `PersistentRoot` no passo 2, foi necessário porque a estratégia reversa remove dependências não permitidas pelo modelo de dependências aspectuais, como em algumas estratégias anteriores. O outro *stub* não esperado é referente ao teste da classe `BRChristmasDiscount`, no passo 10, que precisa que um método de sua superclasse, a classe `BRPriceDiscount`, seja simulado.

No passo 3 foi necessário usar o caso de implementação de *stubs* mostrado na Seção 7.5. No que se refere a possíveis usos de casos de implementação de *stubs*, nos passos 3 e 9 é possível

Tabela 6.9: Custo da estratégia Aleatória em número de *stubs* e membros internos no estudo do sistema de comércio.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	ApplyFrequentDiscount	A	CaptureCustomer	A	0	0	0	1	0	0
			EPricePersonalisation	A	0	0	0	1	0	0
			Product	C	0	1	–	–	–	–
			Customer	C	0	0	–	–	–	–
			BRFrequentDiscount	C	1	1	–	–	–	–
2	ExtendCustomer	A	Customer	C	0	0	–	–	–	–
3	MyPersistentEntities	A	Customer	C	0	0	–	–	–	–
			Store	C	0	0	–	–	–	–
			ShoppingCart	C	0	0	–	–	–	–
			Product	C	0	0	–	–	–	–
			PersistentRoot	C	0	0	–	–	–	–
			PersistentEntities	A	0	0	0	0	0	0
4	ECheckout	A	Customer	C	0	0	–	–	–	–
			TestAspect	A	0	0	1	0	0	0
			Store	C	0	1	–	–	–	–
5	ShoppingCart	C	Product	C	0	5	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	0
			PersistentRoot	C	0	0	–	–	–	–
6	BRFrequentCustomer	C	Customer	C	1	3	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	0
			PersistentRoot	C	0	0	–	–	–	–
7	ApplyChristmasDiscount	A	Product	C	0	1	–	–	–	–
			EPricePersonalisation	A	0	0	0	1	0	0
			BRChristmasDiscount	C	0	1	–	–	–	–
8	Product	C	Store	C	0	1	–	–	–	–
			PersistentEntities	A	0	0	0	0	3	0
			PersistentRoot	C	0	0	–	–	–	–
9	ApplyFrequentCustomer	A	Customer	C	0	0	–	–	–	–
10	BRPriceDiscount	C	BRChristmasDiscount	C	1	2	–	–	–	–
11	PersistentEntities	A	Store	C	4	8	–	–	–	–
			PersistentRoot	C	0	0	–	–	–	–
			Customer	C	2	6	–	–	–	–
12	Customer	C	Store	C	0	1	–	–	–	–
		C	PersistentRoot	C	0	0	–	–	–	–
13	BRChristmasDiscount	C	–	–	–	–	–	–	–	–
14	PersistentRoot	C	–	–	–	–	–	–	–	–
15	BRFrequentDiscount	C	–	–	–	–	–	–	–	–
16	EPricePersonalisation	A	Store	C	0	1	–	–	–	–
			ShoppingCart	C	0	1	–	–	–	–
17	Store	C	–	–	–	–	–	–	–	–
18	CaptureCustomer	A	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				36	9	33	1	3	5	0

a utilização do caso de implementação apresentado na Seção 7.3.1 e nos passos 2, 13, 14, 15 e 16 é possível o uso do caso de implementação detalhado na Seção 7.7.

Tabela 6.10: Custo da estratégia Reversa em número de *stubs* e membros internos no estudo do sistema de comércio.

Ord.	Módulo em teste	Tipo	*Stubs* implementados	Tipo	#Atr.	#Met.	#Ad.	#CJ.	#Introdução	#Alt. Hierar. de Herança
1	ApplyFrequentCustomer	A	ECheckout	A	0	0	0	1	0	0
			BRFrequentCustomer	C	0	1	–	–	–	–
			Customer	C	0	0	–	–	–	–
2	BRFrequentCustomer	C	Customer	C	1	4	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	0
			MyPersistentEntities	A	0	1	0	0	0	1
			PersistentRoot	C	0	0				
			ExtendCustomer	A	0	0	0	0	1	0
3	ECheckout	A	Customer	C	0	0	–	–	–	–
			Store	C	0	1	–	–	–	–
4	ApplyChristmasDiscount	A	ApplyFrequentDiscount	A	1	1	0	0	0	0
			Product	C	0	1	–	–	–	–
			BRChristmasDiscount	C	0	1	–	–	–	–
			EPricePersonalisation	A	0	0	0	1	0	0
5	ApplyFrequentDiscount	A	BRFrequentDiscount	C	1	1	–	–	–	–
			Product	C	0	1	–	–	–	–
			Customer	C	0	0	–	–	–	–
			EPricePersonalisation	A	0	0	0	1	0	0
			CaptureCustomer	A	0	0	0	1	0	0
6	CaptureCustomer	A	Customer	C	0	0	–	–	–	–
			BRFrequentDiscount	C	1	2	–	–	–	–
			Store	C	0	1	–	–	–	–
7	BRFrequentDiscount	C	Customer	C	0	1	–	–	–	–
			BRPriceDiscount	C	0	2	–	–	–	–
8	ExtendCustomer	A	Customer	C	0	0	–	–	–	–
9	EPricePersonalisation	A	Product	C	0	1	–	–	–	–
			Customer	C	0	0	–	–	–	–
			Store	C	0	1	–	–	–	–
10	BRChristmasDiscount	C	BRPriceDiscount	C	0	1	–	–	–	–
11	BRPriceDiscount	C	–	–	–	–	–	–	–	–
12	MyPersistentEntities	A	Customer	C	0	0	–	–	–	–
			Store	C	0	0	–	–	–	–
			ShoppingCart	C	0	0	–	–	–	–
			Product	C	0	0	–	–	–	–
			PersistentEntities	A	0	0	0	0	0	0
			PersistentRoot	C	0	0	–	–	–	–
13	Customer	C	Store	C	0	1	–	–	–	–
			ShoppingCart	C	2	4	–	–	–	–
			Product	C	0	0	–	–	–	–
			PersistentEntities	A	0	0	0	0	3	0
			PersistentRoot	C	0	0	–	–	–	–
14	Store	C	ShoppingCart	C	2	4	–	–	–	–
			Product	C	0	6	–	–	–	–
			PersistentEntities	A	0	0	0	0	2	0
			PersistentRoot	C	0	0	–	–	–	–
15	ShoppingCart	C	Product	C	0	5	–	–	–	–
			PersistentEntities	A	0	0	0	0	1	0
			PersistentRoot	C	0	0	–	–	–	–
16	Product	C	PersistentEntities	A	0	0	0	0	3	0
			PersistentRoot	C	0	0	–	–	–	–
17	PersistentEntities	A	PersistentRoot	C	0	0	–	–	–	–
18	PersistentRoot	C	–	–	–	–	–	–	–	–
Total de *stubs*/membros internos				50	8	41	0	4	11	1

6.3.2.5 Discussão e análise dos resultados obtidos

Um resumo dos resultados da aplicação das estratégias do estudo para o sistema de comércio é apresentado na Tabela 6.11. Confirmando os resultados obtidos no estudo anterior, o custo das estratégias Aleatória e Reversa foi maior do que o custo das estratégias Conjunta e Incremental+ e, entre estas, o custo da estratégia Conjunta foi menor do que o da estratégia Incremental+. Também de forma análoga ao estudo anterior, a estratégia Conjunta tende a balancear o número de *stubs* de aspectos e classes, enquanto que na estratégia Incremental+ a tendência é contrária, com um número maior de *stubs* de classes do que de aspectos.

Um ponto importante a ser analisado é o número de membros internos implementados nos *stubs* da estratégia Conjunta. Ele é maior que o número das outras estratégias. Essa diferença tão grande deve-se, parcialmente, ao fato de que havia mais métodos de manipulação de atributos de classes que foram simulados para o acoplamento do *framework* de persistência, conforme a explicação apresentada na Seção 6.3.1.5. Outro ponto que pode ter influência no resultado é o fato de que o algoritmo de Briand *et al.* (2003) não considera membros internos no cálculo de remoção de arestas. Ainda focando o número de membros internos, pode-se perceber que a estratégias Incremental+ e Aleatória obtiveram resultados iguais. O bom resultado da estratégia Aleatória em relação a estratégia Incremental dá força à suspeita de que o algoritmo de Briand *et al.* (2003) não apresenta resultados tão bons quando o número de membros internos é considerado como parte do custo da estratégia.

Tabela 6.11: Custo de criação de *stubs* no estudo do sistema de comércio.

Estratégia	#*Stub* de classe	#*Stub* de aspecto	#Total de *stubs*	#Atr.	#Met.	#Ad.	#CJ.	#Intr.	#AHH.	#Membros internos
Incremental+	3	16	**19**	4	20	3	0	11	10	**48**
Conjunta	7	8	**15**	16	43	3	0	0	9	**71**
Aleatória	28	8	**36**	9	33	1	3	5	0	**51**
Reversa	37	13	**50**	8	41	0	4	11	1	**65**

6.3.3 Estudo usando o sistema MobileMedia

O MobileMedia (Figueiredo *et al.*, 2008) é uma linha de produtos de software para aplicações que manipulam fotos, musicas e vídeos em dispositivos móveis. No estudo foi usado um produto da linha de produtos composto por 51 classes e 21 aspectos. O produto pode mostrar arquivos de fotos e enviá-las a outros dispositivos móveis via mensagens multimídia. O produto tem a capacidade de tocar músicas e mostrar vídeos. Pode, ainda, organizar e contar arquivos de fotos, de musicas e de vídeos armazenados no dispositivo móvel, bem como manter diferentes copias do mesmo arquivo. Também é possível indicar alguns dos arquivos como favoritos. Além disso, o produto manipula exceções provenientes da utilização de arquivos.

Por causa do tamanho do sistema, e consequentemente das tabelas que mostram os *stubs* implementados, são mostrados nesta seção apenas os SCCs do AORD do sistema MobileMedia, na Tabela 6.12, e o resultado final da condução dos estudos para as quatro estratégias, na Tabela 6.13. Detalhes dos *stubs* implementados podem ser encontrados no documento de trabalho de Ré e Masiero (2008). Na Tabela 6.12, a primeira coluna identifica o SCC encontrado, a segunda mostra o número de ciclos que o SCC possui, a terceira lista quais módulos compõem o SCC e a quarta coluna mostra se o módulo é um aspecto ou uma classe. Conforme pode ser visto na tabela, foram encontrados no MobileMedia 7 SCCs com diferentes números de classes, aspectos e ciclos. Conforme pode ser observado, ainda, todos os SCCs envolvem simultâneamente aspectos e classes. O maior SCC é composto por 9 módulos e o menor por 2 módulos, entre classes e aspectos.

Tabela 6.12: Detalhes dos SCCs do AORD do MobileMedia.

Id.	#Ciclos	#Módulos no SCC	Nome do módulo	Tipo do módulo
SCC1	2	3	`DataModelAspectEH`	A
			`AlbumData`	C
			`MediaAccessor`	C
SCC2	2	3	`UtilAspectEH`	A
			`MusicMediaUtil`	C
			`MediaUtil`	C
SCC3	1	2	`SCreensAspectEH`	A
			`PhotoViewScreen`	C
SCC4	8	8	`ControllerAspectEH`	A
			`PhotoAspect`	A
			`SelectMediaController`	C
			`MusicAspect`	A
			`VideoAspect`	A
			`AlbumController`	C
			`MediaListController`	C
			`MediaController`	C
SCC5	1	2	`CopyPhotoAspect`	A
			`PhotoViewController`	C
SCC6	2	4	`SMSAspect`	A
			`SmsReceiverController`	C
			`SmsReceiverThread`	C
			`SmsOrCaptureOrPhotoOrVideo`	A
SCC7	2	3	`CaptureVideoAspect`	A
			`VideoCaptureController`	C
			`CapturePhotoAspect`	A

6.3.3.1 Discussão e análise dos resultados obtidos

No estudo do MobileMedia foram feitos vários usos de padrões de implementação de *stubs*, assim como no estudo dos sistemas anteriores. Não houve, como no sistema de comércio, casos de implementação de adendos para testar conjuntos de junção, uma vez que o MobileMedia não expõe os conjuntos de junção como interfaces de entrecorte. No entanto, vários ciclos são formados por declarações de relaxamento de exceção, ao contrário dos outros sistemas anteri-

ormente estudados. Por isso, a coluna 9 da Tabela 6.13, intitulada "#DRE", mostra o número de declarações de relaxamento de exceção simuladas. É importante salientar que dependências desse tipo são sempre bidirecionais, ou seja, o aspecto depende da classe que possui a exceção relaxada e a classe depende do aspecto que faz o relaxamento.

Tabela 6.13: Custo de criação de *stubs* no estudo do sistema MobileMedia.

Estratégia	#*Stub* de classe	#*Stub* de aspecto	#Total de *stubs*	#Atr.	#Met.	#CJ.	#Intr.	#DRE.	#Membros internos
Incremental+	5	21	**26**	0	20	15	11	19	**65**
Conjunta	13	11	**24**	0	41	4	6	3	**54**
Aleatória	9	133	**142**	32	221	8	8	8	**277**
Reversa	7	310	**317**	41	435	0	0	0	**476**

Conforme pode-se observar na Tabela 6.13 as estratégias Reversa e Aleatória têm custo de integração maior que as estratégias Incremental+ e Conjunta, conforme o esperado. A estratégia Conjunta tem custo menor que a estratégia Incremental+, de acordo com a segunda linha de raciocínio. A mesma tendência de equilíbrio entre o número de *stubs* de classes e de aspectos na estratégia Conjunta e de desequilíbrio na estratégia Incremental+ constatada nos estudos anteriores pode ser vista neste estudo. Com relação à implementação de membros internos dos *stubs*, seu número é menor na estratégia Conjunta do que na estratégia Incremental+.

6.4 Conclusões sobre os estudos

Com base nos estudos conduzidos nos sistemas Telecom, de comércio e MobileMedia, pode-se afirmar que as duas linhas de raciocínio foram confirmadas. Apesar da diferença não parecer muito grande, a estratégia Conjunta tem custo de teste de integração menor que a estratégia Incremental+. Essas duas estratégias mostram melhor resultado do que executar o teste de integração *ad hoc*, representado pela estratégia Aleatória, conforme discutido na literatura especializada. Também é confirmada a intuição de que testar primeiramente os aspectos não é uma boa solução, conforme os resultados obtidos pela estratégia Reversa. É importante ressaltar que essas afirmações são feitas com base no número de *stubs* de classe e aspectos implementados e não de seus membros internos.

Nos três estudos foi verificada a tendência de balanceamento entre o número de *stubs* de classes e de aspectos quando a estratégia Conjunta é utilizada. Em contrapartida, também nos três estudos, verifica-se a tendência de que o número de *stubs* de aspectos é maior que o número de *stubs* de classes quando a estratégia Incremental+ é aplicada. Esse é um fato importante se os membros internos de cada *stub* forem analisados com mais atenção e forem considerados para o custo de implementação de *stubs*. Se o sistema em teste possuir aspectos pequenos, com pou-

cos membros internos implementados, pode-se conjecturar que a estratégia Incremental+ terá um desempenho melhor que a estratégia Conjunta. Isso acontece porque, com a tendência da estratégia Incremental+ de gerar um número maior de *stubs* de aspectos, menor será o número de membros internos simulados nesses *stubs*. Em contrapartida, se um sistema em teste possui aspectos com número de membros internos igual ou superior ao número de membros internos de classes, então pode-se conjecturar que a estratégia Conjunta terá um resultado melhor, por causa de sua tendência ao balanceamento entre o número de *stubs* de classes e de aspectos.

Um ponto em aberto, que merece atenção em futuras pesquisas, é o desempenho do algoritmo de Briand *et al.* (2003) adaptado ao modelo de dependências aspectuais proposto na Seção 4.2.1, quando o foco é o número de membros internos implementados nos *stubs*. No estudo conduzido para o sistema de comércio, o número de membros internos implementados em *stubs* na estratégia Conjunta obteve um resultado bastante ruim, sendo maior que o da estratégia Reversa. Isso não aconteceu nos outros dois estudos.

6.5 Considerações Finais

Neste capítulo foram apresentados os estudos de caracterização das estratégias de ordenação propostas no Capítulo 4. O estudo mostra que a estratégia Conjunta tem custo de implementação menor que a estratégia Incremental+ e, assim, determina qual das duas estratégias tem melhor desempenho. O estudo mostrou que a aplicação da estratégia Conjunta minimizou o número de *stubs* para os sistemas Telecom, de comércio e MobileMedia.

Além da determinação da melhor estratégia de ordenação, o estudo também validou o AORD proposto e, consequentemente, o modelo de dependências aspectuais propostos no Capítulo 4. Pôde-se observar que o número de *stubs* realmente implementados nas estratégias Conjunta e Incremental+ foi bastante próximo do número calculado pelo algoritmo de Briand *et al.* (2003) adaptado segundo cada estratégia. O número de *stubs* realmente implementados não foi igual o número de *stubs* esperados por dois motivos: o teste de classes e aspectos abstratos faz com que *stubs* não previstos sejam implementados; e aspectos apenas com conjuntos de junção precisam de *stubs* de aspectos com adendos que utilizam tais conjuntos de junção para serem testados. O primeiro caso não afeta o desempenho das estratégias porque a diferença no número de *stubs* é a mesma nas estratégias Conjunta e Incremental+. Isso acontece porque elas seguem a diretriz de Briand *et al.* (2003) e dependências provenientes de relacionamentos de herança não são removidas do AORD. Portanto, uma superclasse abstrata sempre vai ser testada e integrada antes de suas subclasses.

O segundo caso acontece por uma decisão tomada na definição do AORD. O teste de um conjunto de junção precisa apenas de um adendo para que o entrecorte ocorra em todos os pontos de junção e não de todos os adendos que o utilizam. A decisão leva em consideração que é preferível implementar um *stub* não esperado desse tipo do que mapear no AORD dependências que ligam o aspecto que possui o conjunto de junção a todos os outros aspectos que usam o conjunto de junção. Se uma arquitetura que enfatiza o reúso é utilizada, podem existir vários aspectos que usam um mesmo conjunto de junção e, caso essas dependências fossem mapeadas no AORD, vários outros ciclos poderiam se formar. Isso poderia levar a um custo maior porque poderiam ser necessárias remoções de várias arestas e, consequentemente, implementação de vários *stubs*, ao invés de implementar apenas um *stub*.

Um ponto importante que deve ser ressaltado é que o estudo levou à identificação de alguns casos recorrentes de implementação de *stubs*, descritos no Apêndice 7. Alguns desses casos devem obrigatoriamente ser utilizados em certas situações. Isso acontece porque não se pode executar os testes sem sua utilização, como por exemplo, no caso da implementação de adendos para testar conjuntos de junção, ou no caso da implementação de um *stub* de classe para que uma introdução de método seja testada. Outros casos de implementação dos *stubs* coletados são opcionais e, se utilizados, tornam a implementação dos *stubs* para o teste de um módulo mais simples. Em alguns casos o número de *stubs* realmente implementados diminui em relação aos *stubs* esperados computados pelo algoritmo segundo cada estratégia.

CAPÍTULO 7

Casos de implementação de *Stubs* e *Drivers* de teste

7.1 Considerações iniciais

Os casos de implementação mostrados neste capítulo representam uma amostra dos tipos de *stubs* implementados durante a condução dos estudos. Os casos de implementação de *stubs* e *drivers* ocorreram, em sua maioria, quando as estratégias Reversa e Aleatória foram aplicadas, em virtude do grande número de *stubs* implementados. Contudo, eles também foram encontrados na aplicação das estratégias Conjunta e Incremental+. É importante salientar que a maioria deles ocorreu de forma recorrente nos três sistemas estudados.

Na Seção 7.2 é apresentada uma descrição geral do papel desempenhado pelos *stubs* e *drivers* durante o estudo exploratório conduzido com relação à sua aplicabilidade. As seções 7.3, 7.4, 7.5, 7.6, 7.7 e 7.8 apresentam os casos de implementação de seis *stubs* e *drivers* em questão. Por fim, na Seção 7.9, são apresentadas as considerações finais.

7.2 *Stubs* e *drivers* para o teste de programas OA

No contexto de estratégias de ordenação de classes e aspectos para o teste de integração de programas OA, um *stub* é implementado para cada remoção temporária de arestas do AORD. As arestas representam diferentes relacionamentos entre classes e aspectos, conforme definido na Seção 4.2.1. Os *stubs* simulam comportamento de acordo com tipo de relacionamento que a aresta removida representa. Portanto, são implementados *stubs* com características diferentes daqueles implementados na programação OO em virtude dos novos tipos de relacionamento. Além deles, *stubs* e *drivers* referentes aos relacionamentos tradicionais da programação OO

também são afetados pela presença dos aspectos. Diferentemente dos *stubs*, que devem simular comportamento quando solicitado, de maneira passiva, os *drivers* são necessários para controlar de maneira ativa o teste de um ou mais módulos.

Os casos mostrados nas seções 7.3, 7.4, 7.5, 7.6 e 7.8 são de uso obrigatório, sem eles não seria possível conduzir as atividades de teste de integração relatadas no Capítulo 6. Os outros casos, mostrados nas seções 7.3.1 e 7.7, são usados para diminuir o custo de implementação de *stubs* e sua aplicação é facultativa. Uma breve descrição de cada um dos casos de implementação é apresentada na Tabela 7.1.

As dependências tipo "S" e do tipo "It" referentes a declarações de alteração de herança são diferentes das demais, em termos de implementação de *stubs*. A ausência de um aspecto que faz um relaxamento de exceção causa erros de compilação no módulo alvo. Assim, uma solução deve ser adotada antes mesmo de iniciar a atividade de teste, já que não há uma versão executável. Devem ser usadas duas alternativas: na primeira, o foco é a classe alvo do relaxamento de exceção, e deve-se implementar os tratadores de exceção diretamente nas classes; na segunda, o foco é o aspecto, e a solução é semelhante àquela adotada para dependências do tipo "C", apresentada na Seção 7.3. A diferença é que o ponto de junção deve lançar uma exceção a ser relaxada pelo aspecto. A solução para dependências do tipo "It" referentes a declarações de hierarquia de herança é não efetuar o teste, uma vez que não é possível testar diretamente a existência de um relacionamento de herança.

Os exemplos usados nas casos de implementação são baseados nos três sistemas estudados. Eles não são iguais aos sistemas reais para simplificar a explicação do caso de implementação em questão.

7.3 *Stubs* para simular comportamento referente a dependências do tipo "C"

Uma dependência do tipo "C" ocorre sempre entre um conjunto de junção de um aspecto e um ponto de junção localizado em um outro módulo qualquer. O conjunto de junção pode denotar não somente dependências do tipo "C", mas também dependências do tipo "As". Isso acontece por causa dos parâmetros capturados pelo conjunto de junção por meio dos comandos `this`, `target` e `args`, e do uso do comando condicional `if`. Considerando que no aspecto existe ao menos um adendo que utiliza o conjunto de junção que denota as dependências, para efetuar o teste de um aspecto é necessário que estejam disponíveis os módulos ligados a ele pelas dependências do tipo "C" – que são os pontos de junção – e os módulos ligados a ele pelas dependências do tipo "As". Portanto, pode-se notar a necessidade de dois tipos de *stubs*, um

Tabela 7.1: Descrição resumida dos casos de implementação de *stubs* e *drivers*.

Caso	Descrição
Stubs para simular comportamento referente a dependências do tipo "C"	Para testar conjuntos de junção é necessário simular pontos de junção por meio de *stubs*. Em um caso especial, *drivers* de teste podem ser implementados para simular pontos de junção capturados por conjuntos de junção do tipo *call*, diminuindo o esforço de implementação de *stubs* e melhorando o controle sobre o módulo em teste. Esses benefícios são ainda maiores se o ponto de junção originalmente ocorrer em um adendo.
Stubs para simular comportamento referente a dependências do tipo "It"	Quando existe uma introdução de método pode ser necessário implementar um *stub* que simula o contexto em que o método introduzido deve ser testado, uma vez que ele pode usar atributos e outros métodos do alvo da introdução.
Stubs para simular comportamento referente a dependências do tipo "U"	Algumas vezes, adendos de um aspecto usam conjuntos de junção de outros aspectos para capturar pontos de junção. Nesses casos, a implementação de *stubs* de aspectos com conjuntos de junção que capturam o ponto de junção é necessária.
Stubs aspectuais para testar conjuntos de junção	Uma definição de conjunto de junção é uma construção que não pode ser diretamente executada e, por isso, não pode ser diretamente testada. Nesses casos são necessários *stubs* aspectuais para a verificação da captura dos conjuntos.
Stubs para simular comportamento referente a dependências do tipo "As"	Quando um módulo em teste precisa de atributos e métodos que são introduzidos por um outro aspecto ou são originários de alterações de hierarquias de herança, pode-se implementá-los diretamente em classes das quais o módulo em teste já tenha dependência ao invés de implementar *stubs* de aspectos com declarações inter-tipos.
Drivers para controlar o teste de aspectos *Stateful* atrelados a fluxos de execução	Aspectos que têm atributos devem ter seus estados testados. Instâncias de aspectos *stateful* cujo modelo de instanciação usa o designador `percflow` ou o designador `percflowbelow` não podem ser referenciados diretamente em um *driver* de teste convencional. Para capturar as instância nos pontos corretos dos fluxos de execução e, a partir daí, averiguar cada mudança de estado em virtude da execução de diferentes adendos, é necessário implementar um *driver* aspectual.

para cada tipo de dependência. Os *stubs* que simulam dependências do tipo "As" são triviais da programação OO, já os *stubs* que simulam dependências do tipo "C" são específicos da programação OA. Os *stubs* que simulam dependências do tipo "C" seguem a idéia de simulação de pontos e junção necessários ao teste do aspecto de forma semelhante à proposta de Yamazaki *et al.* (2005).

Um exemplo desse caso de implementação de *stubs*, ilustrado na Figura 7.1, pode ser constatado por meio da observação da Listagem 7.1, da Listagem 7.3 e da Listagem 7.4. Nas linhas 2 e 3 da Listagem 7.1 pode ser visto um conjunto de junção que captura a execução do método

`Store.checkout()`. Para testar o aspecto `ApplyFrequentCustomer` é necessário um *stub* trivial da programação OO da classe `Customer`, mostrado na Listagem 7.2, referente a uma dependência do tipo "As". Também é necessário um *stub* da classe `Store`, mostrado na Listagem 7.3, referente a uma dependência do tipo "C". No *stub* da classe `Store` está o ponto de junção simulado, o método `checkOut()`, mostrado nas linhas 2, 3 e 4. Note-se que no *driver* de teste existe uma chamada ao método `Store.checkout()`, mostrado na linha 8 da Listagem 7.4, que é capturado pelo conjunto de junção do aspecto `ApplyFrequentCustomer`, caracterizando o teste desse aspecto.

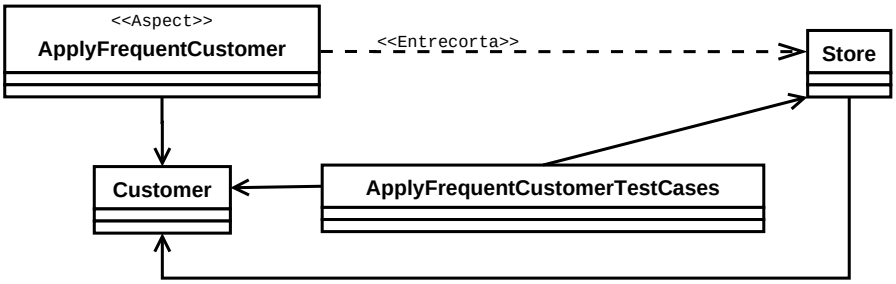


Figura 7.1: Classes e aspectos necessários ao teste de `ApplyFrequentCustomer`.

Listagem 7.1: Aspecto `ApplyFrequentCustomer`.

```
1 public aspect ApplyFrequentCustomer {
2     pointcut checkout(Customer aCustomer): args(aCustomer) &&
3         execution(float Store.checkOut(Customer));
4
5     after(Customer aCustomer): checkout(aCustomer){
6         aCustomer.incrementNumberOfPurchases();
7     }
8 }
```

Listagem 7.2: *Stub* da classe `Customer`.

```
1 public class Customer {
2     int numberOfPurchases;
3     public void incrementNumberOfPurchases(){
4         numberOfPurchases++;
5     }
6 }
```

Listagem 7.3: *Stub* da classe `Store`.

```
1 public class Store {
2     public float checkOut(Customer aCustomer){
3         return 0;
4     }
5 }
```

Listagem 7.4: *Driver* de teste do aspecto `ApplyFrequentCustomer`.

```
1  public class ApplyFrequentCustomerTestCases extends TestCase {
2      public void test1(){
3
4      Store store = new Store();
5      Customer customer = new Customer();
6      customer.numberOfPurchases=0;
7
8      store.checkOut(customer);
9      assertEquals(customer.numberOfPurchases, 1);
10     }
11 }
```

7.3.1 Transferência de dependência de um *stub* para um *driver* de teste

Um caso especial de implementação de *stubs* para simular dependências do tipo "C" ocorre quando o conjunto de junção de um aspecto em teste usa o designador `call` e não referencia o objeto entrecortado por meio dos comandos `this` e `if`. Um exemplo desse caso de implementação pode ser observado no estudo conduzido com o sistema Telecom. Durante o teste do aspecto `TimerLog` do sistema Telecom foi necessário implementar um *stub* aspectual para simular o comportamento do aspecto `Timing`. `TimerLog` é um aspecto que depende do aspecto `Timing` e da classe `Timer`, conforme pode ser observado na Listagem 7.5 e na Listagem 7.6. `TimerLog` entrecorta adendos de `Timing`, mostrado nas linhas 3 e 7 da Listagem 7.6, para registrar o início e o término de cada chamada telefônica.

Para efetuar o teste, considerando que a classe `Timer` e o aspecto `TimerLog` não estão presentes na configuração de teste, são necessários os seguintes artefatos: um *stub* de `Timing` com dois adendos e dois conjuntos de junção; um *stub* que simula o contexto para os adendos de `Timing` entrecortarem, chamado na Figura 7.2 de `StubJPTiming`; e um *driver* de teste que manipula `StubJPTiming` para cada caso de teste.

Listagem 7.5: Aspecto `TimerLog`.

```
1  public aspect TimerLog {
2      after(Timer t) returning () : target(t) && call(* Timer.start()){
3        System.out.println("Timer started: " + t.startTime);
4      }
5      after(Timer t) returning () : target(t) && call(* Timer.stop()){
6        System.out.println("Timer stopped: " + t.stopTime);
7      }
8  }
```

Listagem 7.6: Parte do aspecto `Timing`.

```
1  public aspect Timing {
2      after (Connection c) returning ():Billing.initBilling(c) {
```

```
3         getTimer(c).start();
4     }
5     after(Connection c) returning () : Billing.endBilling(c) {
6     Customer c1, c2;
7     getTimer(c).stop();
8     c1 = c.getCaller();
9     c1.totalConnectTime += getTimer(c).getTime();
10    c2 = c.getReceiver();
11    c2.totalConnectTime += getTimer(c).getTime();
12    c1.save();
13    c2.save();
14   }
15   public Timer Connection.timer = new Timer();
16    ...
17 }
```

Vários são os problemas com essa abordagem: a criação de um *stub* apenas para simular o ponto de junção, `StubJPTiming`; a necessidade de implementação de um *stub* de `Timing` com dois conjuntos de junção; a existência de dois níveis de indireção entre o *driver* de teste e o módulo a ser testado; e o esforço para manter esses artefatos de software na configuração de teste durante todo o teste de integração. Especificamente neste exemplo, os problemas foram potencializados porque o ponto de junção é um adendo que, ao contrário de métodos que são invocados, só pode ser executado em um contexto capturado por conjuntos de junção.

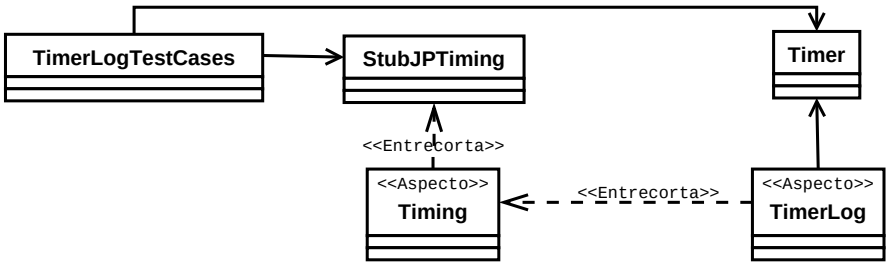


Figura 7.2: Classes e aspectos necessários ao teste de `Timing`.

O artifício utilizado para minimizar os problemas foi a transferência de dependência, que ocorre quando dependências geradas por conjuntos de junção são transferidas de um *stub* para um *driver* de teste no momento do teste. Note-se, pela Listagem 7.5, que os conjuntos de junção são do tipo `call`, conforme pode-se observar nas linhas 2 e 5. Apesar dos pontos de junção estarem localizados originalmente no aspecto `Timing` – as chamadas aos métodos `Timer.start()` e `Timer.stop()`, mostrados nas linhas 3 e 7 da Listagem 7.6 –, a dependência entre `TimerLog` e `Timing` deixa de existir no momento do teste. Isso acontece porque a dependência fica estabelecida entre o módulo em teste `TimerLog` e o *driver* de teste. Na Listagem 7.7 é mostrado parte do *driver* de teste do aspecto `TimerLog`, em que pode-se observar na linha 8, o ponto de junção capturado pelo aspecto em teste `TimerLog`. No *driver* de teste

mostrado na Listagem 7.7 é apresentado apenas o teste de um dos adendos de `TimerLog`, para simplificar a explanação do caso de implementação.

Listagem 7.7: Parte do *driver* de `TimerLogTestCases`.

```
1  public class TimerLogTestCase extends TestCase {
2     Timer t;
3     PrintStream oldps;
4     BufferedReader br;
5
6     public void test1(){
7        String resp="";
8        t.start();
9        try {
10          resp = br.readLine();
11          oldps.println(resp);
12       } catch (IOException e) {
13          e.printStackTrace();
14       }
15       assertEquals("Timer started: 1", resp);
16    }
17 }
```

Na Figura 7.3 são apresentados os artefatos utilizados no teste do aspecto `TimerLog` quando este caso de implementação é utilizado. Com a transferência da dependência do ponto de junção do *stub* para o *driver*, não é necessário implementar o *stub* que deveria simular o comportamento do aspecto `Timing` e a classe que deveria ser entrecortada por esse *stub* aspectual.

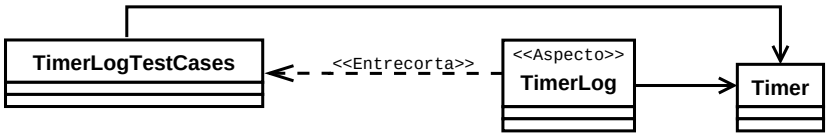


Figura 7.3: Classes e aspectos necessários ao teste de `Timing` usando a transferência de dependência.

7.4 *Stubs* para simular comportamento referente a dependências do tipo “It”

Dependências do tipo `It` ocorrem quando um aspecto possui uma declaração intertipos que introduz métodos e atributos em classes ou outros aspectos. A introdução de um método ou atributo também gera dependências do tipo “As”, uma vez que podem referenciar outras classes que não àquela alvo da introdução. Novamente, assim como no caso de implementação de *stubs* anterior, *stubs* referentes a essas dependências são tradicionais da programação OO. No

entanto, a dependência entre o método introduzido e os outros métodos e atributos da classe alvo da introdução é particular da programação OA. Dessa forma, o *stubs* que simulam relacionamentos do tipo "It" também devem conter os métodos e atributos referenciados pelo método introduzido.

Para efetuar o teste de um método introduzido em uma classe é necessário prover o contexto em que esse método é executado. Isso quer dizer que devem estar disponíveis métodos e atributos da própria classe alvo da introdução referenciados pelo método introduzido. Um exemplo desse caso de implementação de *stubs* pode ser visto na Listagem 7.8. Note-se que para testar o método `playVideoMedia` introduzido na classe `MediaController`, considerando que as outras classes necessárias já estão na configuração de teste, é necessário que existam os métodos `MediaController.getAlbumData()` e `MediaController.setNextController()`. Isso porque eles são referenciados pelo método introduzido, como pode ser visto nas linhas 4, 6 e 9 da Listagem 7.8. Parte do *stub* da classe `MediaController` é listado na Listagem 7.9 e parte do *driver* para testar a introdução feita pelo aspecto `VideoAspect` é mostrado na Listagem 7.10. No trecho de código entre as linhas 6 e 11 da Listagem 7.9, pode-se observar os dois métodos simulados pelo *stub* e, na linha 5 da Listagem 7.10, é possível observar a chamada e o teste do método introduzido, o método `playVideoMedia()`. Os relacionamentos entre as classses e os aspectos importantes para o teste de `MediaController` é ilustrado na Figura 7.4.

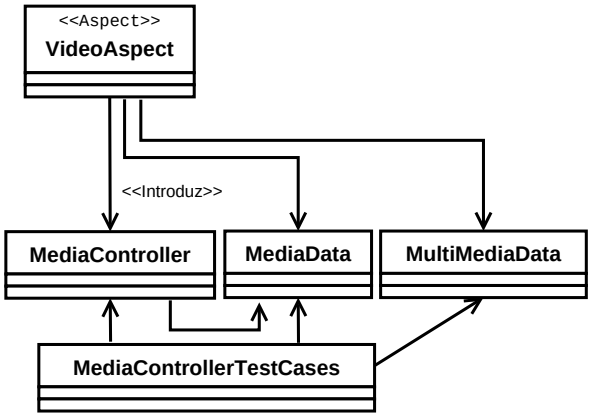


Figura 7.4: Classes e aspectos importantes para o teste de `MediaController`.

Listagem 7.8: Parte do aspecto `VideoAspect`.

```
1 public boolean MediaController.playVideoMedia(String selectedMediaName) {
2    InputStream storedMusic = null;
3    try {
4       MediaData mymedia = this.getAlbumData().getMediaInfo(selectedMediaName);
5       if (mymedia instanceof MultiMediaData){
6         storedMusic = ((VideoAlbumData) this.getAlbumData()).getVideoFromRecordStore(
```

```
7                       getCurrentStoreName(), selectedMediaName);
8           ...
9           this.setNextController(controller);
10        }
11        return true;
12    }
13    ...
14 }
```

Listagem 7.9: Parte do *stub* da classe `MediaController`.

```
1  public class MediaController {
2      MediaData media;
3      MediaController(MediaData m){
4          this.media=m;
5      }
6      public MediaData getAlbumData(){
7          return media;
8      }
9      public void setNextController(PlayVideoController controller){
10         ...
11     }
12     ...
13 }
```

Listagem 7.10: Parte do *driver* de teste do aspecto `VideoAspect`.

```
1  public class MediaControllerTestCases {
2      public void test1(){
3          MediaData md = new MultiMediaData(1, "AlbumName", "MediaLabel1", "Video1");
4          MediaController mc = new MediaController(md);
5          assertTrue(mc.playVideoMedia(selectedMediaName));
6      }
7      ...
8  }
```

7.5 *Stubs* para simular comportamento referente a dependências do tipo "U"

Dependências do tipo "U" ocorrem entre adendos e conjuntos de junção. Para que um adendo seja executado e testado é necessário que exista ao menos um ponto de junção para que o entrecorte seja feito. Portanto, *stubs* que simulam relacionamentos desse tipo devem implementar conjuntos de junção que forneçam o contexto de execução dos adendos. É importante ressaltar que os conjuntos de junção implementados em *stubs* desse tipo não precisam ser necessariamente iguais aos originais, mas devem simular o contexto de execução dos pontos de junção

originais. Os adendos também denotam dependências do tipo "As", por meio de seus parâmetros, que são triviais da programação OO.

Um exemplo deste caso de implementação de *stubs* foi utilizado no teste do aspecto `Timing` do sistema de Telecom, mostrado na Listagem 7.6. Foi necessário implementar um *stub* de `Billing`, mostrado na Listagem 7.11. A dependência é causada pelo uso que `Timing` faz de dois conjuntos de junção definidos em `Billing`, mostrados nas linhas 2 e 5 da Listagem 7.6. Para simplificar o exemplo, considera-se que as demais classes já estejam implementadas na configuração de teste. O *stub* de `Billing` deve implementar os dois conjuntos de junção usados por `Timing` e fornecer o mesmo contexto de execução para que os adendos sejam executados. Isso pode ser visto nas linhas 2 e 3 da Listagem 7.11. O *driver* de teste do aspecto `Timing` é mostrado na Listagem 7.12. Nele é possível observar os dois pontos de junção capturados pelos dois adendos de `Timing`, nas linhas 7 e 8, as chamadas aos métodos `Connection.complete()` e `Connection.Drop()`. As classes e os aspectos necessários para o teste de `Timing` são ilustrados na Figura 7.5.

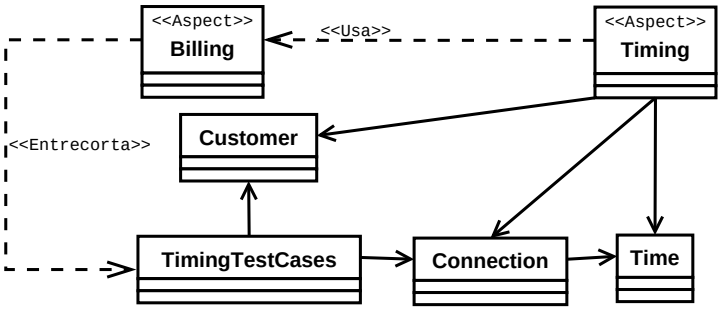


Figura 7.5: Classes e aspectos necessários para o teste de `Timing`.

Listagem 7.11: Parte do *stub* do aspecto `Billing`.

```
1 public aspect Billing {
2     pointcut initBilling(Connection conn): target(conn) && call(void Connection.complete());
3     pointcut endBilling(Connection conn): target(conn) && call(void Connection.drop());
4     ...
5 }
```

Listagem 7.12: Parte do *driver* de teste do aspecto `Timing`.

```
1 public class TimingTestCases extends TestCase {
2     public void test1(){
3         Customer cust = new Customer();
4         Connection conn = new Customer();
5         long total1, total2;
6         total1 = conn.timer;
7         conn.complete();
8         conn.drop();
9         total2=conn.getTime();
```

```
10          assertTrue(total1<total2);
11      }
12      ...
13  }
```

Um problema que surge da simulação de conjuntos de junção é que caso seja mal implementado, o conjunto de junção pode capturar pontos de junção indesejados que afetam a execução do módulo em teste. Além disso, o potencial problema continua nas próximas etapas do teste, uma vez que pode ser necessário manter o *stub* com os conjuntos de junção na configuração de teste durante o teste de módulos integrados subsequentemente.

7.6 *Stubs* aspectuais para testar conjuntos de junção

A proposta de XPIs de Griswold *et al.* (2006) enfatiza a exposição dos pontos de junção em interfaces de entrecorte. Isso faz com que conjuntos de junção sejam implementados de maneira separada dos adendos. O problema a ser contornado é como testar código fonte não executável. Isso deve ser feito sob dois pontos de vista: testar se são capturados os pontos de junção corretos e se os objetos de contexto são capturados corretamente. Este caso de implementação só pode ser aplicado por meio da análise de documentos de especificação que descrevem quais são os pontos de junção capturados pelos conjuntos de junção em teste e quais são os adendos que usam esses conjuntos de junção. A ideia é usar adendos para exercitar a captura de pontos de junção pelo conjunto de junção em teste e verificar a passagem dos parâmetros.

Essa abordagem foi usada durante o teste das regras de negócio no sistema de comércio proposta por Cibrán *et al.* (2003). O sistema de comércio implementa regras de negócio segundo uma arquitetura em que aspectos são responsáveis por capturar e aplicar regras de negócio implementadas em classes. Existem aspectos que representam XPIs, que são constituídos apenas de conjuntos de junção, e são responsáveis por capturar pontos de junção, conforme pode ser observado na Listagem 7.13. Existem outros aspectos que são responsáveis por aplicar as regras de negócio por meio de adendos e introduções de métodos e atributos nas classes-base.

Listagem 7.13: Aspecto `ECheckout`.

```
1 public aspect ECheckout {
2   pointcut checkout(Customer aCustomer):
3     args(aCustomer) &&
4     execution(float Store.checkOut(Customer));
5 }
```

O aspecto em teste `ECheckout` possui um conjunto de junção que deve entrecortar a execução do método `Store.checkOut()`. Assim, considerando que a classe `Store` já esteja

na configuração de teste, é necessário um *driver* de teste responsável por invocar o método `Store.checkOut()` a ser entrecortado por `ECheckout`, mostrado na Listagem 7.14, e um *stub* aspectual com um adendo para utilizar o ponto de junção entrecortado, mostrado na Listagem 7.15. Note-se que o *driver* e o *stub* trabalham conjuntamente para checar se o conjunto de junção captura o ponto de junção corretamente. As classes e aspectos desse exemplo de caso de implementação de *stubs* e *drivers* são ilustrados na Figura 7.6.

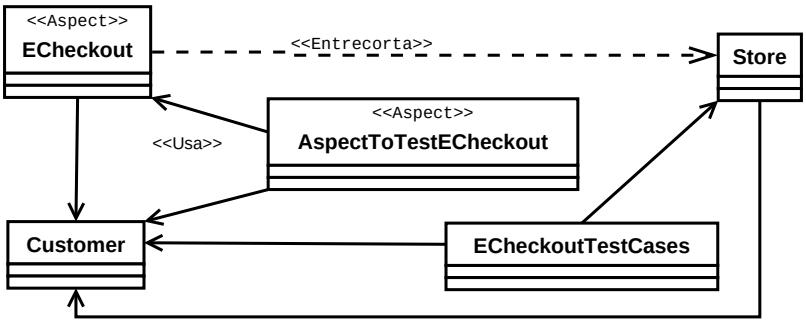


Figura 7.6: Classes e aspectos necessários para o teste de `ECheckout`.

Listagem 7.14: *Driver* de teste do aspecto `ECheckout`.

```
1  public class ECheckoutTestCases extends TestCase {
2    public void test1(){
3      Store store = new Store();
4      Customer cust = new Customer();
5      cust.checked=false;
6
7      store.checkout(cust);
8      assertTrue(cust.checked==true);
9    }
10 }
```

Listagem 7.15: *Stub* aspectual para o teste do aspecto `ECheckout`.

```
1  public aspect AspectToTestECheckout {
2     public boolean Customer.checked;
3
4     after(Customer aCustomer): ECheckout.checkout(aCustomer){
5        aCustomer.checked=true;
6     }
7  }
```

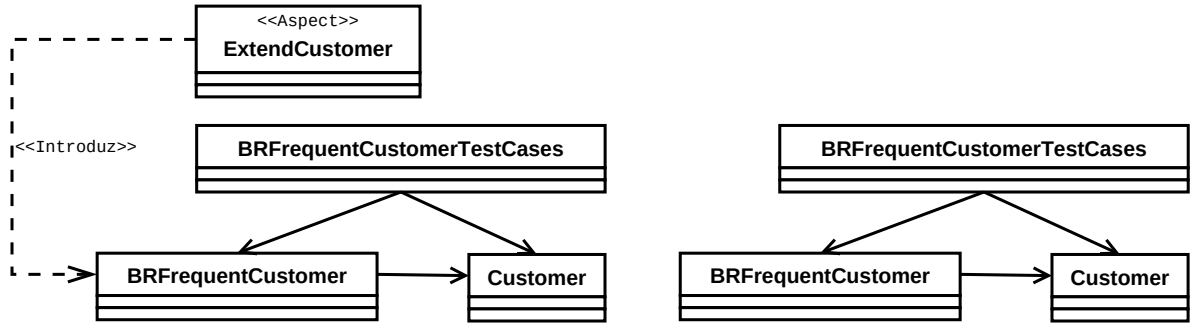
7.7 *Stubs* para simular comportamento referente a dependências do tipo "As"

Algumas dependências do tipo "As" geram *stubs* que não são triviais da programação OO porque sofrem a influência da programação OA. Isso acontece quando o relacionamento que denota uma dependência do tipo "As" é estabelecido por meio de referências a métodos e atributos introduzidos ou herdados via alteração de hierarquia de herança. Quando um método referencia um método introduzido, por exemplo, ele depende do aspecto que faz a introdução e também do alvo da introdução que fornece o contexto de execução do método introduzido. Isso é garantido pelo modelo de dependências aspectuais proposto na Seção 4.2.1 e pelas duas estratégias propostas, já que dependências do tipo "It" não são removidas do AORD. Assim, ao invés de implementar dois *stubs*, um para representar o aspecto que faz a introdução e outro para implementar o alvo da introdução, implementa-se apenas um *stub* que já contém os atributos e métodos que seriam originalmente introduzidos. Este caso de implementação de *stubs* explora o forte acoplamento entre o aspecto que faz a introdução e o alvo da introdução para simplificar a implementação de *stubs*.

Um exemplo desse caso de implementação de *stubs* pode ser visto na Listagem 7.16. A listagem mostra a classe em teste `BRFrequentCustomer`, que faz uma referência ao método `Customer.becomeFrequent()`, mostrado na linha 6. Originalmente a classe `Customer` não possui o método `becomeFrequent()`, ele é introduzido pelo aspecto `ExtendCustomer`, conforme pode ser visto na linha 6 da Listagem 7.17. Considerando que a classe `Customer` e o aspecto `ExtendCustomer` não estejam na configuração de teste, ao usar este caso de implementação de *stubs*, é necessário implementar apenas um *stub* da classe `Customer`, conforme apresentado na Listagem 7.18. Nele é possível observar a simulação do método `becomeFrequent()`, mostrado nas linhas 7, 8 e 9. Na Listagem 7.19 é mostrado o *driver* de teste da classe `BRFrequentCustomer`. Na Figura 7.7 são mostrados as classes e o aspectos necessários para o teste de `BRFrequentCustomer` de acordo com o uso ou não do caso de implementação.

Listagem 7.16: Classe `BRFrequentCustomer`.

```
1 public class BRFrequentCustomer {
2     public boolean condition(Customer aCustomer){
3         return(aCustomer.numberOfPurchases() >= 10);
4     }
5     public void action(Customer aCustomer){
6         aCustomer.becomeFrequent();
7     }
8     public void apply(Customer aCustomer){
9         if (condition(aCustomer)){
```



(a) Classes e aspectos necessários para o teste de `BRFrequentCustomer` sem o uso do caso de implementação.

(b) Classes e aspectos necessários para o teste de `BRFrequentCustomer` com o uso do caso de implementação.

Figura 7.7: Classes e aspectos necessários ao teste de `BRFrequentCustomer` de acordo com o uso ou não do caso de implementação.

```
10              action(aCustomer);
11          }
12      }
13  }
```

Listagem 7.17: Aspecto `ExtendCustomer`.

```
1  public aspect ExtendCustomer {
2      private boolean Customer.frequent = false;
3      public boolean Customer.isFrequent(){
4          return frequent;
5      }
6      public void Customer.becomeFrequent(){
7          frequent = true;
8      }
9  }
```

Listagem 7.18: *Stub* da classe `Customer`.

```
1  public class Customer {
2      int numberOfPurchases;
3
4      public Customer(int number){
5          numberOfPurchases = number;
6      }
7      public void becomeFrequent(){
8          numberOfPurchases=0;
9      }
10     public int getNumberOfPurchases(){
11         return numberOfPurchases;
12     }
13 }
```

Listagem 7.19: *Driver* de teste da classe `BRFrequentDiscountTestCases`.

```
1  public class BRFrequentCustomerTestCases extends TestCase {
2      public void test1(){
3          Customer cust = new Customer(10);
4          BRFrequentCustomer br = new BRFrequentCustomer();
5
6          br.apply(cust);
7          assertEquals(cust.numberOfPurchases, 0);
8      }
9      ...
10 }
```

Pode-se perceber que existe uma diminuição pequena do esforço de implementação de *stubs* uma vez que deixa-se de implementar o código referente a um aspecto. Além dessa diminuição, também é menor o esforço para se manter a configuração de teste consistente e evita-se que erros sejam cometidos por causa de declarações de introduções. Um ponto a ser considerado é que esse caso pode ser combinado com outras introduções e alterações de hierarquia de herança, conforme explicado nos casos complexos de mapeamento mostrados na Seção 5.3.5, tornando considerável a diminuição da complexidade de implementação dos *stubs* e de gerenciamento da configuração de teste.

7.8 *Drivers* para controlar o teste de aspectos *Stateful* atrelados a fluxos de execução

Duas características distintas da programação OA influenciam a implementação de *drivers* de teste: a possibilidade de implementação de aspectos *stateful* e o modelo de instanciação de aspectos. Um aspecto é *stateful* quando contém atributos e portanto pode possuir diferentes estados. No teste de aspectos *stateful*, além de testar os objetos afetados pelos pontos de junção entrecortados, há a necessidade de testar também os diferentes estados assumidos pelo aspecto. Embora o modelo de instanciação de aspectos adotado pelo combinador do AspectJ cria por padrão aspectos *singleton*, ou seja, existe apenas uma instância de cada aspecto em todo o sistema, pode-se alterá-lo por meio da implementação de uma condição de criação de instâncias dentro do próprio aspecto. Isso faz com que a criação de instâncias de aspectos sejam atreladas a diferentes elementos da programação OA.

O testador encontra limitações no controle do processo de criação de uma instância de um aspecto, ficando impossibilitado, por exemplo, de controlar diretamente o momento da criação de uma instância e referenciar a nova instância criada por meio de uma variável. Dentre as formas de instanciação de aspectos, uma deve ser observada com atenção especial, a criação de instâncias atreladas a fluxos de execução de métodos. Nesse modelo de instanciação,

um aspecto é criado quando um determinado ponto de execução de um método é alcançado, definido pelos designadores `percflow` e `percflowbelow`. Ao contrário de instâncias de aspectos criadas com os designadores `perthis` e `pertarget`, que são atreladas a objetos específicos e podem ser acessados por meio de referencias a esses objetos, e com o designador *pertypewithin*, que são atreladas a classes específicas e podem ser acessadas por meio das classes, instâncias criadas com os designadores `percflow` e `percflowbelow` não podem ser acessadas nos *drivers* de teste convencionais porque não há como fazer uma referência a um fluxo de execução de um trecho de código.

Para testar um aspecto com essas características, a idéia é criar um *driver* de teste tradicional e um *driver* de teste auxiliar que é um aspecto. O *driver* de teste tradicional é responsável por testar os adendos e o *driver* aspectual de teste deve testar os diferentes estados que o aspecto em teste pode assumir. O *driver* aspectual de teste deve interceptar os mesmos pontos de junção que o aspecto em teste, o que possibilita obter a instância do aspecto e coordenar o teste de acordo com cada adendo que altera o estado do aspecto.

Um exemplo deste caso de implementação pode ser observado no estudo conduzido com o sistema de comércio. O aspecto `ApplyFrequentDiscount`, mostrado na Listagem 7.20, é um aspecto *stateful* não *singleton* responsável por aplicar uma regra de negócio que possui estados que variam de acordo com o cliente para o qual uma compra é concluída. Para isso, o aspecto possui um atributo chamado `isBusinessRuleApplied` que deve indicar se a regra de negócio foi aplicada a uma determinada compra ou não, para que outra regra de negócio também não seja aplicada a uma mesma compra. Na Figura 7.8 é apresentado, usando a notação MATA, o cenário que mostra a recuperação do preço de produtos quando do momento do fechamento de uma compra. Na Figura 7.9 é apresentado o mesmo cenário, porém na presença do aspecto `ApplyFrequentDiscount`, que intercepta a chamada ao método `getPrice()` para aplicar a regra de negócio contida em `BRFrequentDiscount`. A criação de uma instância do aspecto `ApplyFrequentDiscount` acontece no momento da execução do método `Store.checkout()`, conforme pode ser observado nas chamadas compreendidas no fragmento `Mod` da Figura 7.9. A instância do aspecto é atrelada, então, ao fluxo de execução que corresponde às chamadas de métodos mostradas em `Mod`.

Listagem 7.20: Aspecto `ApplyFrequentDiscount`.

```
1 public aspect ApplyFrequentDiscount percflow(execution(float Store.checkOut(Customer))){
2   BRFrequentDiscount businessRule;
3   private boolean isBusinessRuleApplied=false;
4   public boolean getIsBusinessRuleApplied(){
5     return isBusinessRuleApplied;
6   }
7   public void setBR(BRFrequentDiscount br){
8     businessRule = br;
```

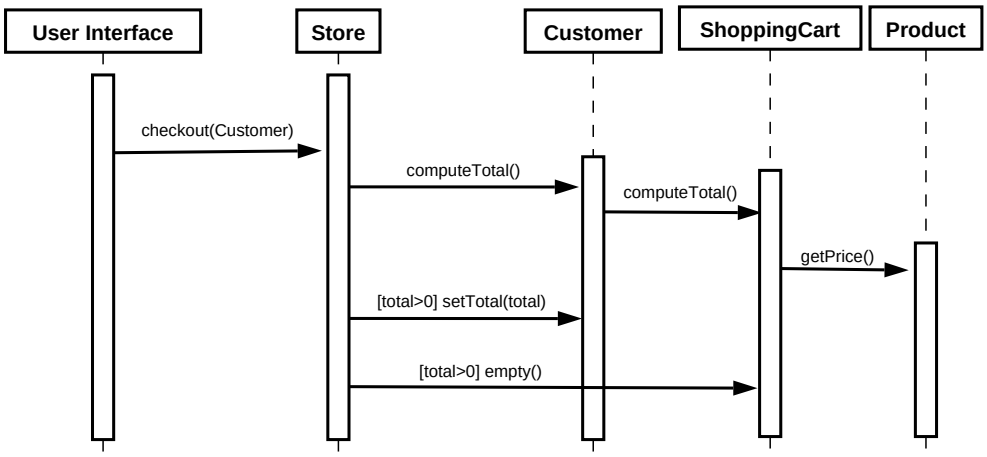


Figura 7.8: Cenário do fechamento de uma compra sem a presença do aspecto `ApplyFrequentDiscount`.

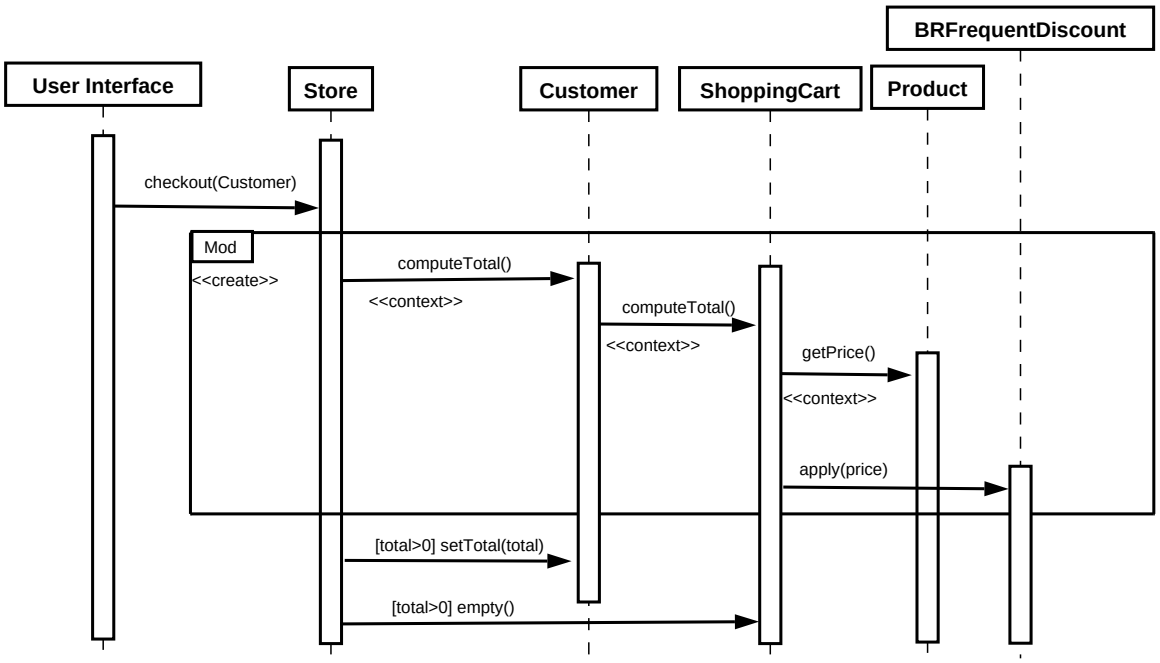


Figura 7.9: Cenário do fechamento de uma compra com a presença do aspecto `ApplyFrequentDiscount`.

```
9   }
10  float around(Product aProduct): cflow(execution(float Store.checkOut(Customer))) &&
11                                  target(aProduct) && call (float Product.getPrice()){
12    float price1=0, price2=0;
13    price1 = (Float) proceed(aProduct);
14    price2 = businessRule.apply(price1);
15    if (price1!=price2)
16      isBusinessRuleApplied=true;
17    return price2;
18  }
```

```
19 }
```

Listagem 7.21: *Driver* de teste do aspecto `ApplyFrequentDiscountTestCases`.

```
1 public class TestApplyFrequentDiscount extends TestCase {
2   public void test1(){
3     Store s = new Store(new Product(10));
4     Customer c = new Customer();
5     assertTrue(s.checkOut(c) == 5);
6   }
7 }
```

Listagem 7.22: *Driver* aspectual de teste do aspecto `ApplyFrequentDiscount`.

```
1  public aspect TestAspect extends TestCase {
2    declare precedence: TestAspect, ApplyFrequentDiscount;
3
4    after(Product aProduct): cflow(execution(float Store.checkOut(Customer))) &&
5                               target(aProduct) && call (float Product.getPrice())
6                        && cflow(execution(* TestApplyFrequentDiscount.test1())){
7      ApplyFrequentDiscount afd = ApplyFrequentDiscount.aspectOf();
8      assertTrue(afd.getIsBusinessRuleApplied()==true);
9    }
10 }
```

Considera-se que todos os módulos necessários estão presentes no teste do aspecto `ApplyFrequentDiscount`. O *driver* de teste tradicional, mostrado na Listagem 7.21, possui um caso de teste que invoca o método `Store.checkOut()` entrecortado pelo aspecto em teste, o que pode ser visto na linha 5 da Listagem 7.21. Por sua vez, o *driver* aspectual de teste mostrado na Listagem 7.22 captura as instâncias corretas do aspecto `ApplyFrequentDiscount`, permitindo a manipulação das instâncias afim de verificar sua correta mudança de estado. Note-se há uma correlação entre o conjunto de junção usado pelo adendo que provoca a mudança de estado, o caso de teste que simula o ponto de junção e o conjunto de junção usado pelo adendo que verifica a mudança de estado. Isso pode ser observado nas linhas 10 e 11 da Listagem 7.20 e nas linhas 4, 5 e 6 da Listagem 7.22. Nas listagens é possível ver que os conjuntos de junção do aspecto em teste e do *driver* aspectual afetam os mesmos pontos de junção. A pequena diferença entre eles é que o *driver* aspectual restringe a captura dos pontos de junção para que ela aconteça apenas quando o caso de teste é executado. Isso é feito para que o *driver* aspectual não interfira em outros testes enquanto está na configuração de teste.

É importante observar que a precedência entre o *driver* aspectual de teste e o aspecto em teste deve ser explicitamente estabelecida, uma vez que deve-se garantir que os adendos de verificação de mudança sejam executados na ordem correta. Outro ponto importante é que, em

alguns casos pode ser necessário implementar mais de um *driver* aspectual de teste , em virtude da existência de diferentes adendos que possam vir a utilizar um mesmo conjunto de junção.

7.9 Considerações finais

Os casos de implementação de *stubs* e *drivers* são úteis para descrever padrões de implementação de *stubs* que ocorrem em virtude dos novos relacionamentos advindos da programação OA e, também, de relacionamentos da programação que sofrem a influência dos conceitos da programação OA. Pôde-se observar pelos casos de implementação de *stubs* e *drivers*, que os relacionamentos existentes na programação OA tornam a atividade de teste de integração mais complexa, no que diz respeito à simulação de contextos de execução de unidades em teste frente às demais unidades. Outro ponto importante é que os casos de implementação de *stubs* e *drivers* aqui propostos podem ser utilizados em conjunto com outras propostas relacionadas ao teste de software, como as de Franchin *et al.* (2007), Lemos *et al.* (2007) e Lemos e Masiero (2008). Dessa forma, os casos de implementação são úteis mesmo que não utilizados com as estratégias de ordenação. Vale ressaltar ainda que, como não foram encontrados na literatura descrições de *stubs* e *drivers* de teste na programação OA ou padrões de implementação de *stubs* e *drivers* como os de Binder (1999), os casos de implementação aqui propostos mostram algumas diretrizes de implementação desses artefatos e servem como base para futuras pesquisas.

CAPÍTULO

8

Conclusão

8.1 Considerações finais

Nesta tese foram propostas duas estratégias de ordenação de classes e aspectos para apoiar o teste de integração de programas OA, denominadas Incremental+ e Conjunta. Essas estratégias têm o objetivo de tornar mais eficiente a atividade de teste por meio da minimização do número de *stubs* implementados na fase de teste de integração. As estratégias usam o AORD, que descreve interdependências entre classes e aspectos. O AORD proposto estende o ORD originalmente proposto por Kung *et al.* (1995a), que representa dependências geradas por relacionamentos da programação OO, por meio da incorporação das novas dependências geradas por relacionamentos da programação OA. Para elaborar o AORD foi desenvolvimento um modelo de dependências aspectuais que identifica as dependências por meio de mecanismos de conexão disponíveis na linguagem AspectJ. Com base no modelo de dependências aspectuais e o no AORD, as duas estratégias podem utilizar o algoritmo de Briand *et al.* (2003) diretamente, sem alterações.

Para apoiar a aplicação das duas estratégias, foi proposto também um processo de mapeamento que estabelece regras para derivar um AORD a partir de um modelo de projeto usando as notação UML e MATA. O processo de mapeamento proposto tem o objetivo de evidenciar a possibilidade de aplicação das duas estratégias em situações reais de desenvolvimento de software OA. Além disso, o processo de mapeamento mostra que o modelo de dependências aspectuais é capaz de descrever em linhas gerais os principais tipos de dependências da programação OA, apesar de ser desenvolvido a partir da linguagem AspectJ. Isso pode ser concluído porque a partir do processo é possível produzir um AORD, que é baseado no modelo de dependências aspectuais.

Para validar o modelo de dependências e o AORD um estudo com três sistemas implementados em AspectJ foi conduzido. O estudo utilizou quatro diferentes estratégias de ordenação. Os resultados obtidos mostram que a estratégia Conjunta é melhor que a estratégia Incremental+ quando o conceito de inconsciência é violado. Quando o conceito de inconsciência é mantido, o uso de uma ou outra estratégia é indiferente. Além das duas estratégias propostas foram utilizadas uma estratégia *ad hoc* e uma estratégia denominada Reversa. Essas duas estratégias foram incluídas por dois motivos: como controle comparativo do desempenho das duas estratégias propostas frente às demais, e, seguindo a intuição posteriormente comprovada, para que fosse implementado uma grande quantidade de *stubs*.

Os *stubs* implementados durante a condução do estudo foram coletados, analisados e classificados. Com isso desenvolveu-se um catálogo de casos de implementação de *stubs* e de *drivers* de teste. O catálogo tem por objetivo descrever como é a implementação de *stubs* de aspectos e de *stubs* para testar aspectos, fornecendo diretrizes úteis para a atividade de teste.

8.2 Contribuições do trabalho

A principal contribuição desta tese é o modelo de dependências aspectuais proposto, juntamente com a extensão do AORD e das duas estratégias de ordenação. Deve-se notar que as estratégias de ordenação propostas diferenciam-se pela maneira como constroem o AORD. O modelo de dependências foi elaborado a partir da sintaxe e semântica da linguagem AspectJ e é capaz de identificar de forma completa as dependências entre classes e aspectos. O estudo de caracterização conduzido mostra que o modelo de dependências e, consequentemente, o AORD faz com que as estratégias de ordenação mantenham a premissa de que o número de *stubs* implementados deve ser idealmente igual ao número de *stubs* indicados pelo algoritmo de ordenação.

O processo de mapeamento proposto também é uma contribuição relevante. O processo permite que ordenações de classes e aspectos sejam produzidas ao longo da fase de projeto e início das fases de implementação e teste. Além disso, o processo mostra que o mapeamento pode ser automatizado.

Os casos de implementação de *stubs* e *drivers* contribuem para o teste de software porque formam um catálogo com diretrizes de implementação não disponíveis na literatura. Além disso o catálogo ajuda a compreender como são as dependências provenientes dos relacionamentos de classes e aspectos.

8.3 Dificuldades e limitações

A maioria das dificuldades encontradas durante o trabalho está relacionada à recenticidade da programação OA. Se por um lado existe uma quantidade pequena de sistemas reais implementados com aspectos disponíveis para utilização em estudos, por outro não existe um método de desenvolvimento de software OA maduro. O primeiro caso dificulta a realização de estudos em condições reais. Além disso, a maioria dos sistemas disponíveis com acesso ao código fonte é bastante simples e, portanto, não apresentam as características ideias para a condução de estudos.

No segundo caso, a falta de métodos de desenvolvimento maduros implica na falta de modelos amplamente aceitos que descrevem os relacionamentos entre classes e aspectos na fase de projeto. Dessa forma, as notações atualmente disponíveis não são poderosas o suficiente para modelar de maneira fiel, adequada e sucinta os relacionamentos entre classes e aspectos, como a UML, por exemplo.

Apesar do modelo de dependências aspectuais descrever de maneira adequada as dependências entre classes e aspectos, não existem estudos que mostram o grau de acoplamento referente a essas dependências. Isso mostra claramente que, se do ponto de vista de número de *stubs* implementados as estratégias obtém bom resultado, o mesmo não acontece quando o número de membros internos é considerado. Estudos com métricas de avaliação do grau de acoplamento são importantes para estabelecer uma relação entre o tipo de dependência e o custo de implementação de um *stub*. De certa forma, essa limitação também atinge as estratégias de ordenação propostas.

Uma dificuldade encontrada durante a condução dos estudos foi a execução dos teste do sistema MobileMedia usando JUnit. Esse sistema é operacional em dispositivos móveis e o JUnit não é adequado em tal contexto. Foi necessário simular alguns pacotes específicos para aplicativos de dispositivos móveis para a condução dos estudos.

8.4 Pesquisas futuras

O algoritmo de Briand *et al.* (2001) não considera os membros internos dos módulos em teste para efetuar a ordenação, conforme pôde ser constato no estudo conduzido usando o sistema de comércio, apresentado na Seção 6.3.2. Uma nova estratégia que considere os membros internos no momento da ordenação, nos moldes dos trabalhos de Malloy *et al.* (2003) e Abdurazik e Offutt (2006), poderia ser elaborada. Para isso, o AORD deve representar de alguma forma o acoplamento que cada aresta denota.

Uma ferramenta que efetue a automatização do processo de mapeamento proposto poderia ser desenvolvida e trabalhar de maneira acoplada à ferramenta MATA por intermédio da ferramenta RSM (IBM, 2009), citada na Seção 3.4.1.

Experimentos controlados para avaliar mais formalmente as duas estratégias propostas poderiam ser conduzidos com outros sistemas e diferentes testadores. Apesar dos estudos exploratórios serem bastante controlados, a experiência do testador pode influenciar no número de membros internos de *stubs* implementados. Adicionalmente, esse trabalho poderia estender e generalizar os casos de implementação aqui propostos.

Outro trabalho que traria contribuição importante é o desenvolvimento de um processo que englobe as estratégias de ordenação e diferentes técnicas e critérios de teste OA para o teste de integração, como os trabalhos de Franchin *et al.* (2007) e de Lemos e Masiero (2008). Adicionalmente, tal trabalho poderia investigar a utilização do AORD e do modelo de dependências no teste de regressão, na linha do trabalho de Kung *et al.* (1996b).

Referências Bibliográficas

ABDURAZIK, A.; OFFUTT, J. Coupling-based class integration and test order. In: *(AST'06) Proceedings of the 2006 International Workshop on Automation of Software Test*, New York, NY, USA: ACM Press, 2006, p. 50–56.

AGRAWAL, H.; ALBERI, J.; HORGAN, J. R.; LI, J.; LONDON, S.; WONG, W. E.; GHOSH, S.; WILDE, N. Mining system tests to aid software maintenance. *IEEE Computer*, p. 64–73, 1998.

AKSIT, M.; BERGMANS, L.; VURAL, S. An object-oriented language-database integration model: the composition-filters approach. In: *(ECOOP '92) Proceedings of the European Conference on Object-Oriented Programming*, London, UK: Springer-Verlag, 1992, p. 372–395.

ALEXANDER, R. T.; BIEMAN, J. M. *Towards the systematic testing of aspect-oriented programs*. Relatório Técnico CS-04-105, Colorado State University, Fort Collins, Colorado, 2004.

ALEXANDER, R. T.; OFFUTT, A. J. Criteria for testing polymorphic relationships. In: *(ISSRE'00) Proceedings of the 11th International Symposium on Software Reliability Engineering*, Washington, DC, USA: IEEE Computer Society, 2000, p. 15–23.

DE ALWIS, B.; KICZALES, G. Apostle: Aspect programming in smalltalk. 2009. Disponível em http://www.cs.ubc.ca/labs/spl/projects/apostle.html (Acessado em 20/02/2009)

AMMANN, P.; BLACK, P. E. A specification-based coverage metric to evaluate test sets. In: *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, Washington, DC, USA: IEEE Computer Society, 1999, p. 239–248.

ANBALAGAN, P.; XIE, T. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In: *(MUTATION '06) Proceedings of the Second Workshop on Mutation Analysis*, Washington, DC, USA: IEEE Computer Society, 2006, p. 3.

ARMSTRONG, J.; MITCHELL, R. Uses and abuses of inheritance. *IEE Software Engineering Journal*, v. 9, n. 1, p. 19–26, Institution of Electrical Engineers (IEE), 1994.

ASPECTC.ORG Aspectc++. 2009.
Disponível em http://acdt.aspectc.org/ (Acessado em 20/02/2009)

BADRI, L.; BADRI, M.; BLE, V. S. A method level based approach for OO integration testing: An experimental study. In: *(SNPD/SAWN'05) Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks*, Maryland, EUA, 2005, p. 102–109.

BAEKKEN, J. S.; ALEXANDER, R. T. A candidate fault model for AspectJ pointcuts. In: *(ISSRE '06) Proceedings of the 17th International Symposium on Software Reliability Engineering*, Washington, DC, USA: IEEE Computer Society, 2006, p. 169–178.

BANIASSAD, E. L. A.; CLARKE, S. Theme: An approach for aspect-oriented analysis and design. In: *(ICSE 2004) 26th International Conference on Software Engineering*, Edinburgh, United Kingdom: IEEE Computer Society, 2004, p. 158–167.

BARBEY, S.; STROHMEIER, A. The problematics of testing object-oriented software. In: ROSS, M.; BREBBIA, C. A.; STAPLES, G.; STAPLETON, J., eds. *SQM'94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, July 26-28 1994*, 1994, p. 411–426.
Disponível em http://lgl.epfl.ch/pub/Papers/barbey-problematic_of_toos.ps (Acessado em 11/08/2005)

BARTSCH, M.; HARRISON, R. An evaluation of coupling measures for AspectJ. In: *LATE Workshop AOSD (2006*, ????

BEIZER, B. *Software testing techniques*. Second ed. Van Nostrand Reinhold, 1990.

BIEMAN, J. M.; GHOSH, S.; ALEXANDER, R. T. A technique for mutation of Java objects. In: *(ASE 2001) 16th IEEE International Conference on Automated Software Engineering*, Coronado Island, San Diego, CA: IEEE Computer Society, 2001, p. 337–340.

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

BOUJARWAH, A. S.; SALEH, K.; AL-DALLAL, J. Dynamic data flow analysis for Java programs. *Information & Software Technology*, v. 42, n. 11, p. 765–775, 2000.

BRIAND, L.; FENG, J.; LABICHE, Y. Using genetic algorithms and coupling measures to devise optimal integration test orders. In: *(SEKE '02) Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, New York, NY, USA: ACM, 2002, p. 43–50.

BRIAND, L. C.; DALY, J. W.; WÜST, J. K. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, v. 25, n. 1, p. 91–121, 1999.

BRIAND, L. C.; LABICHE, Y.; WANG, Y. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In: *(ISSRE '01) Proceedings of the 12th International Symposium on Software Reliability Engineering*, Washington, DC, USA: IEEE Computer Society, 2001, p. 287–296.

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

CAMARGO, V. V.; MASIERO, P. C. Frameworks orientados a aspectos. In: *(SBES 2005) XIX Simpósio Brasileiro de Engenharia de Software*, Uberlândia – MG, 2005, p. 200–216.

CECCATO, M.; TONELLA, P. Measuring the effects of software aspectization. In: *1st Workshop on Aspect Reverse Engineering*, Netherlands, 2004, p. 5.
Disponível em http://homepages.cwi.nl/~tourwe/ware/ceccato.pdf (Acessado em 20/02/2009)

CECCATO, M.; TONELLA, P.; RICCA, F. Is AOP code easier or harder to test than OOP code? In: *First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005)*, Chicago, Illinois – USA, 2005, p. 5.

CHAIM, M. L. *POKE-TOOL – uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados*. Dissertação de Mestrado, DCA-FEEC-UNICAMP, Campinas, SP, 1991.

CHEN, H. An approach for object-oriented cluster-level tests based on UML. In: *Proceedings of the 2003 IEEE International Conference on Systems, Man, and Cybernetics – SMC 2003*, Los Alamitos, California, USA: IEEE Computer Society Press, 2003, p. 1064–1068.

CHEN, H. Y.; TSE, T. H.; CHAN, F. T.; CHEN, T. Y. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, v. 7, n. 3, p. 250–295, 1998.

CHEN, H. Y.; TSE, T. H.; CHEN, T. Y. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, v. 10, n. 1, p. 56–109, 2001.

CHEN, M.-H.; KAO, H. M. Testing object-oriented programs - an integrated approach. In: *(ISSRE'99) Proceedings of the 10th International Symposium on Software Reliability Engineering*, Washington, DC, USA: IEEE Computer Society, 1999, p. 73–83.

CHEVALLEY, P. Applying mutation analysis for object-oriented programs using a reflective approach. In: *(APSEC 2001) 8th Asia-Pacific Software Engineering Conference*, Macau, China: IEEE Computer Society, 2001, p. 267–270.

CHOW, T. S. Testing software design modelled by finite-state machine. *IEEE Transactions on Software Engineering*, v. 4, n. 3, p. 178–187, 1978.

CIBRÁN, M.; D'HONDT, M.; JONCKERS, V. Aspect-oriented programming for connecting business rules. In: *6th International Conference on Business Information Systems*, 2003, p. 10.

CLARKE, P.; MALLOY, B. A taxonomy of classes for implementation-based testing. *Journal of Object Technology*, v. 4, n. 5, p. 95–115, 2005.
Disponível em http://www.jot.fm/issues/issue_2005_07/article2 (Acessado em 20/02/2009)

CLARKE, S.; BANIASSAD, E. *Aspect-oriented analysis and design: the theme approach*. 1 ed. Addison-Wesley, 2005.

DALAL, S. R.; JAIN, A.; KARUNANITHI, N.; LEATON, J. M.; LOTT, C. M.; PATTON, G. C.; HOROWITZ, B. M. Model-based testing in practice. In: *(ICSE'99) Proceedings of the 21st International Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, p. 285–294.

DELAMARO, M. E. *Proteum: Um ambiente de teste baseado na análise de mutantes*. Dissertação de Mestrado, ICMC/USP, São Carlos – SP, 1993.

DELAMARO, M. E. *Mutação de interface: Um critério de adequação inter-procedimental para o teste de integração*. Tese de Doutoramento, Instituto de Física de São Carlos – Universidade de São Paulo, São Carlos, SP, 1997.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M., eds. *Introdução ao teste de sofware*. Editora Campus, série Sociedade Brasileira de Computação, 2007.

DEMILLO, R. A. Mutation analysis as a tool for software quality assurance. In: *COMPSAC80*, Chicago, IL, 1980, p. 27–31.

DEMILLO, R. A.; GWIND, D. S.; KING, K. N.; MCKRAKEN, W. N.; OFFUTT, A. J. An extended overview of the mothra testing environment. In: *Software Testing, Verification and Analysis*, Banff, Canada, 1988, p. 142–151.

VAN DEURSEN, A.; MARIN, M.; MOONEN, L. *A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw*. Relatório Técnico SEN-R0507, CWI, Amsterdam, 2005.

DOMINGUES, A. *Avaliação de critérios e ferramentas de teste para programas oo*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2005.

DOONG, R.-K.; FRANKL, P. G. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, v. 3, n. 2, p. 101–130, 1994.

DOUENCE, R.; FRADET, P.; SÜDHOLT, M. Composition, reuse and interaction analysis of stateful aspects. In: MURPHY, G. C.; LIEBERHERR, K. J., eds. *WTAOP: Workshop On Testing Aspect Oriented Programs at the AOSD' 2005*, ACM, 2004, p. 141–150.

EADDY, M.; AHO, A. V.; HU, W.; MCDONALD, P.; BURGER, J. Debugging aspect-enabled programs. In: LUMPE, M.; VANDERPERREN, W., eds. *Software Composition*, Springer, 2007, p. 200–215 (*Lecture Notes in Computer Science*, v.4829).

FABBRI, S. C. P. F.; MALDONADO, J. C.; MASIERO, P. C.; DELAMARO, M. E. Mutation analysis testing for finite state machine. In: *(ISSRE'94) 5th International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, 1994, p. 220–229.

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.

FIGUEIREDO, E.; CACHO, N.; SANT'ANNA, C.; MONTEIRO, M.; KULESZA, U.; GARCIA, A.; SOARES, S.; FERRARI, F.; KHAN, S.; FILHO, F. C.; DANTAS, F. Evolving software product lines with aspects: an empirical study on design stability. In: *(ICSE'08) Proceedings of the 30th International Conference on Software Engineering*, New York, NY, USA: ACM, 2008, p. 261–270.

FILMAN, R. E.; FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In: FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M., eds. *Aspect-Oriented Software Development*, Boston: Addison-Wesley, p. 21–35, 2005.

FIRESMITH, D. G. Pattern language for testing object-oriented software. *Object Magazine*, v. 5, n. 9, p. 42–45, 1996.

FRANCHIN, I. G.; LEMOS, O. A. L.; MASIERO, P. C. Pairwise structural testing of object and aspect-oriented java programs. In: *(SBES 2007) XXI Simpósio Brasileiro de Engenharia de Software*, João Pessoa–Brazil, 2007, p. 377–393.

FRANKL, P. G.; WEYUKER, E. J. A data flow testing tool. In: *Proceedings of the 2nd Conference on Software Development Tools, Techniques, and Alternatives*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1985, p. 46–53.

FUJIWARA, S.; VON BOCHMANN, G.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, v. 17, n. 6, p. 591–603, 1991.

GAO, J. Z.; KUNG, D.; HSIA, P. An object state test model: object state diagram. In: *CASCON '95: Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, 1995, p. 23.

GÉLINAS, J.-F.; BADRI, M.; BADRI, L. A cohesion measure for aspects. *Journal of Object Technology*, v. 5, n. 7, p. 75–95, 2006.

Disponível em http://www.jot.fm/issues/issue_2006_09/article5 (Acessado em 20/02/2009)

GRADECKI, J. D.; LESIECKI, N.; GRADECKI, J. *Mastering AspectJ: Aspect-oriented programming in Java*. John Wiley & Sons, Inc., 2003.

GRISWOLD, W. G.; SULLIVAN, K.; SONG, Y.; SHONLE, M.; TEWARI, N.; CAI, Y.; RAJAN, H. Modular software design with crosscutting interfaces. *IEEE Software*, v. 23, n. 1, p. 51–60, 2006.

GÖNENÇ, G. A method for design of fault-detection experiments. *IEEE Transactions on Computers*, v. 19, n. 6, p. 551–558, 1970.

HARRISON, W.; OSSHER, H. Subject-oriented programming: a critique of pure objects. In: *OOPSLA '93: Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA: ACM Press, 1993, p. 411–428.

HARROLD, M. J.; MCGREGOR, J. D.; FITZPATRICK, K. J. Incremental testing of object-oriented class structures. In: *14th International Conference on Software Engineering*, Los Alamitos, CA: IEEE Computer Society Press, 1992, p. 68–80.

HARROLD, M. J.; ROTHERMEL, G. Performing data flow testing on classes. In: *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, New York: ACM Press, 1994, p. 154–163.

HARTMANN, J.; IMOBERDORF, C.; MEISINGER, M. UML-based integration testing. In: *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM Press, 2000, p. 60–70.

HELM, R.; HOLLAND, I.; GANGOPADHYAY, D. Contracts: Specifying behavioral compositions in object-oriented systems. In: *Proceedings of the Joint ACM European Conference on Object-Oriented Programming: Systems, Languages, and Applications – OOPSLA/ECOOP '90*, ACM Press, New, York, NY, 1990, p. 169–180.

HOFFMAN, D.; STROOPER, P. A case study in class testing. In: *CASCON '93: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, 1993, p. 472–482.

HORGAN, J. R.; MATHUR, A. P. Assessing testing tools in research and education. *IEEE Software*, v. 9, n. 3, p. 61–69, 1992.

IBM Rational Software Modeler. 2009.
Disponível em http://www.ibm.com/developerworks/rational/products/rsm/ (Acessado em 20/02/2009)

IEEE *IEEE standard glossary of software engineering terminology*. Standard 610.12, IEEE Press, 1990.

INTERNATIONAL SOFTWARE AUTOMATION Panorama C/C++. 2005a. Disponível em http://www.nsesoft.com/ (Acessado em 20/02/2009)

INTERNATIONAL SOFTWARE AUTOMATION Panorama for Java. 2005b. Disponível em http://www.nsesoft.com/ (Acessado em 20/02/2009)

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. *The unified software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

JACOBSON, I.; NG, P.-W. *Aspect-oriented software development with use cases*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2004.

JAROENPIBOONKIT, J.; SUWANNASART, T. Finding a test order using object-oriented slicing technique. In: *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, Washington, DC, USA: IEEE Computer Society, 2007, p. 49–56.

JIN, Z.; OFFUTT, A. J. Coupling-based integration testing. In: *(ICECCS '96) Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*, Washington, DC, USA: IEEE Computer Society, 1996, p. 10.

JUNIT.ORG Junit. Xerox Croporation, 2009. Disponível em http://www.junit.org (Acessado em 20/02/2009)

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of AspectJ. In: KNUDSEN, J. L., ed. *15th European Conference on Object-Oriented Programming*, Springer, 2001, p. 327–353 (*Lecture Notes in Computer Science*, v.2072).

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: AKSIT, M.; MATSUOKA, S., eds. *(ECOOP'97) 11th European Conference on Object-Oriented Programming*, Springer, 1997, p. 220–242 (*Lecture Notes in Computer Science*, v.1241).

KIENZLE, J.; GUERRAOUI, R. AOP: Does it make sense? The case of concurrency and failures. In: MAGNUSSON, B., ed. *ECOOP 2002, Malaga, Spain, June 10-14, 2002, Proceedings*, Springer, 2002, p. 37–61 (*LNCS*, v.2374).

KIENZLE, J.; YU, Y.; XIONG, J. On composition and reuse of aspects. In: LEAVENS, G. T.; CLIFTON, C., eds. *(AOSD'2003) International Conference on Aspect-Oriented Software Development – Workshop on Foundations of Aspect-Oriented Languages*, 2003.

KIM, H. *AspectC#: An AOSD implementation for C#*. Dissertação de Mestrado, University Of Dublin, 2002.

KIM, S.; CLARK, J. A.; MCDERMID, J. A. Assessing test set adequacy for object-oriented programs using class mutation. In: *(SoST'99) Symposyum on Software Techonology*, 1999, p. 72–83. Disponível em http://www-users.cs.york.ac.uk/~{}jac/papers/argentina.pdf (Acessado em 20/02/2009)

KIM, S.; CLARK, J. A.; MCDERMID, J. A. Class mutation: mutation testing for object-oriented programs. In: *(OOSS) Conference on Object-Oriented Software Systems*, 2000. Disponível em http://www.cs.york.ac.uk/~{}jac/papers/ClassMutation.pdf (Acessado em 20/02/2009)

KOBAYASHI, N.; TSUCHIYA, T.; KIKUNO, T. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters*, v. 81, n. 2, p. 85–91, 2002.

KULESZA, U.; COELHO, R.; ALVES, E.; NETO, A. C.; GARCIA, R.; LUCENA, C.; STAA, A. V.; BORBA, P. Implementing framework crosscutting extensions with EJPs and AspectJ. In: *(SBES 2006) XX Simpósio Brasileiro de Engenharia de Software*, 2006, p. 16.

KUNG, D.; LU, Y.; VEIIUGOPALAN, N.; HSIA, P.; TOYOSHIMA, Y.; C, C.; GAO, J. Object state testing and fault analysis for reliable software systems. In: *(ISSRE´96) Proceedings of the Seventh International Symposium on Software Reliability Engineering*, White Plains, NY: IEEE Computer Society Press, 1996a, p. 76–85.

KUNG, D. C.; GAO, J.; HSIA, P.; LIN, J.; TOYOSHIMA, Y. Class firewall, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Program*, v. 8, n. 2, p. 51–65, 1995a.

KUNG, D. C.; GAO, J.; HSIA, P.; TOYOSHIMA, Y.; CHEN, C. A test strategy for object-oriented programs. In: *(COMPSAC'95) Proceedings of the 19th International Computer Software and Applications Conference*, 1995b, p. 239–244.

KUNG, D. C.; GAO, J.; HSIA, P.; TOYOSHIMA, Y.; CHEN, C. On regression testing of object-oriented programs. *Journal of Systems and Software*, v. 32, n. 1, p. 21–40, 1996b.

KUNG, D. C.; HSIA, P.; TOYOSHIMA, Y.; CHEN, C.; GAO, J. Object-oriented software testing: Some research and development. In: *(HASE'98) The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, IEEE Computer Society, 1998, p. 158–165.

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 Press, 2000, p. 136–145.

LADDAD, R. *AspectJ in action: Practical aspect-oriented programming*. 1 ed. Greenwich, Connecticut – USA: Manning Publications Company, 512 p., 2003.

LARMAN, C. *Applying UML and patterns: An introduction to object-oriented analysis and design and the unified process*. Prentice Hall PTR, 2002.

LE HANH, V.; AKIF, K.; TRAON, Y. L.; JÉZÉQUEL, J.-M. Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. In: KNUDSEN, J. L., ed. *(ECOOP 2001) Proceedings of 15th European Conference on Object-Oriented Programming*, Budapest, Hungary: Springer, 2001, p. 381–401 (*Lecture Notes in Computer Science*, v.2072).

LE TRAON, Y.; JÉRON, T.; JÉZÉQUEL, J.; MOREL, P. Efficient OO integration and regression testing. *IEEE Transactions on Reliability*, v. 49, n. 1, p. 12–25, 2000.

LEAVENS, G. T. Modular specification and verification of object-oriented programs. *IEEE Software*, v. 8, n. 4, p. 72–80, 1991.

LEMOS, O. A. L. *Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2005.

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

LEMOS, O. A. L.; MASIERO, P. C. Integration testing of aspect-oriented programs: a structural pointcut-based approach. In: *(SBES 2008) XXII Simpósio Brasileiro de Engenharia de Software*, Campina, Brasil, 2008, p. 49–64.

LEMOS, O. A. L.; VINCENZI, A. M. R.; MALDONADO, J. C.; MASIERO, P. C. Teste de unidade de programas orientados a aspectos. In: *(SBES 2004) XVIII Simpósio Brasileiro de Engenharia de Software*, Brasília, DF, Brasil, 2004, p. 55–70.

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

LIEBERHERR, K. J. *Adaptive object-oriented software: The demeter method with propagation patterns*. Boston, MA, USA: PWS Publishing Co., 1995.

LIMA, G. M. P.; TRAVASSOS, G. H. Testes de integração aplicados a software orientado a objetos: Heurísticas para ordenação de classes. In: *(SBQS 2004) III Simposio Brasileiro de Qualidade de Software*, Sociedade Brasileira de Computação, 2004.

LOPES, C. V.; NGO, T. C. Unit-testing aspectual behavior. In: *Workshop on Testing Aspect-Oriented Programs (WTAOP), held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, Illinois, 2005, p. 6.

LOUGHRAN, N.; RASHID, A. Framed aspects: Supporting variability and configurability for AOP. In: *(ICSR 2004) Software Reuse: Methods, Techniques and Tools: Proceedings of 8th International Conference on Software Reuse*, Springer, 2004, p. 127–140 (*LNCS*, v.3107).

MA, Y.-S.; KWON, Y. R.; OFFUTT, J. Inter-class mutation operators for Java. In: *(ISSRE 2002) 13th International Symposium on Software Reliability Engineering*, IEEE Computer Society, 2002, p. 352–366.

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

MALDONADO, J. C.; BARBOSA, E. F. Teste de software: teoria e prática. In: *XVII Simpósio Brasileiro de Engenharia de Software*, 2003.

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais-usos. In: *XXII Congresso Nacional de Informática*, São Paulo, SP, 1989.

MALDONADO, J. C.; FABBRI, S. C. P. F. Teste de software. In: DA ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C., eds. *Qualidade de Software – Teoria e Prática*, 1 ed, São Paulo: Prentice Hall, p. 73–84, 2001.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S.; DELAMARO, M. E. *Aspectos teóricos e empíricos de teste de cobertura de software*. Notas didáticas 31, ICMC/USP, 1998.

MALLOY, B. A.; CLARKE, P. J.; LLOYD, E. L. A parameterized cost model to order classes for class-based testing of C++ applications. In: *(ISSRE 2003) 14th International Symposium on Software Reliability Engineering*, Denver, CO, USA: IEEE Computer Society, 2003, p. 353–364.

MAO, C.; LU, Y. Aicto: An improved algorithm for planning inter-class test order. In: *CIT '05: Proceedings of the The Fifth International Conference on Computer and Information Technology*, Washington, DC, USA: IEEE Computer Society, 2005, p. 927–931.

MARTENA, V.; ORSO, A.; PEZZÈ, M. Interclass testing of object oriented software. In: *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002)*, Greenbelt, MD, USA, 2002, p. 145–154.

MASSICOTTE, P.; BADRI, L.; BADRI, M. Aspects-classes integration testing strategy: An incremental approach. In: GUELFI, N.; SAVIDIS, A., eds. *RISE*, Springer, 2005, p. 158–173 (*Lecture Notes in Computer Science*, v.3943).

MCDANIEL, R.; MCGREGOR, J. D. Testing the polymorphic interactions between classes. *TR94-103, Clemson University*, p. 56, 1994.

MCGREGOR, J. D.; KORSON, T. D. Integrated object-oriented testing and development processes. *Commun. ACM*, v. 37, n. 9, p. 59–77, 1994.

MCGREGOR, J. D.; SYKES, D. A. *A practical guide to testing object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

MYERS, G. J. *The art of software testing*. John Wiley & Sons, Inc., revised and Updated by Tom Badgett, Todd M. Thomas with Corey Sandler, 2004.

OMG *Unified modeling language: Superstructure, version 2.1.2*. Relatório Técnico OMG document formal/2007-02-05, Object Modeling Group, 2007.
Disponível em http://www.omg.org/docs/formal/07-11-02.pdf (Acessado em 20/02/2009)

OSSHER, H.; KAPLAN, M.; HARRISON, W.; KATZ, A.; KRUSKAL, V. Subject-oriented composition rules. In: *(OOPSLA'95) Proceedings of the tenth annual conference on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA: ACM Press, 1995, p. 235–250.

OVERBECK, J. *Integration testing for object-oriented software*. Tese de Doutoramento, Vienna University of Technology, Vienna, Austria, 1994.

OVERBECK, J. Testing object-oriented software and reusability – contradiction or key to success? In: *Proceedings, 8th Annual Software Quality Week*, San Francisco: Software Research Inc, 1995.

PARADKAR, A. M.; TAI, K.-C.; VOUK, M. A. Specification-based testing using cause-effect graphs. *Annals of Software Engineering*, v. 4, p. 133–157, 1997.

PARASOFT CORPORATION Automated software error prevention tools and products. 2005. Disponível em http://www.parasoft.com/ (Acessado em 20/02/2009)

PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. *Journal Object Oriented Program.*, v. 2, n. 5, p. 13–19, 1990.

PEZZÈ, M.; YOUNG, M. Testing object oriented software. In: *(ICSE'04) Proceedings of the 26th International Conference on Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2004, p. 739–740.

PFLEEGER, S. L. *Software engineering: theory and practice*. 2 ed. Prentice Hall, 2001.

PRESSMAN, R. S. *Software engineering: A practitioner's approach*. McGraw Hill College Division, 2004.

RASHID, A.; CHITCHYAN, R. Persistence as an aspect. In: *(AOSD'03) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, New York, NY, USA: ACM Press, 2003, p. 120–129.

RÉ, R.; MASIERO, P. C. Estudos de caraterização das propostas de estratégias de teste de integração OO/OA. Documento de trabalho., 2008.

RÉ, R.; DOS SANDOS DOMINGUES, A. L.; MASIERO, P. C. Um catálogo de stubs para apoiar o teste de integração de programas orientados a aspectos. In: *(SBES 2008) XXII SBES - Simpósio Brasileiro de Engenharia de Software*, Campinas – SP, 2008, p. 65–80.

SABNANI, K.; DAHBURA, A. A protocol test generation procedure. *Computer Network and ISDN Systems*, v. 15, n. 4, p. 285–297, 1988.

SANT'ANNA, C.; GARCIA, A.; CHAVEZ, C.; LUCENA, C.; V. VON STAA, A. On the reuse and maintenance of aspect-oriented software: An assessment framework. In: *(SBES 2003) XVII Simpósio Brasileiro de Engenharia de Software*, 2003, p. 16.

SCHAUERHUBER, A.; SCHWINGER, W.; KAPSAMMER, E.; RETSCHITZEGGER, W.; WIMMER, M.; KAPPEL, G. *A survey on aspect-oriented modeling approaches*. Relatório Técnico, Vienna University of Technology, 2007.
Disponível em http://www.wit.at/people/schauerhuber/publications/AOMSurvey_Schauerhuber_Oct2007.pdf (Acessado em 20/02/2009)

SHEN, H.; ZHANG, S.; ZHAO, J. An empirical study of maintainability in aspect-oriented system evolution using coupling metrics. In: *(TASE'08) Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2008, p. 233–236.

SINHA, S.; HARROLD, M. J. Criteria for testing exception-handling constructs in Java programs. In: *(ICSM'99) Proceedings of the IEEE International Conference on Software Maintenance*, Washington, DC, USA: IEEE Computer Society, 1999, p. 265–274.

SOARES, S.; LAUREANO, E.; BORBA, P. Implementing distribution and persistence aspects with AspectJ. In: *(OOPSLA'2002) Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, New York, NY, USA: ACM Press, 2002, p. 174–90.

QUEST SOFTWARE JProbe Developer Suite. 2005.
Disponível em http://www.quest.com/jprobe/ (Acessado em 20/02/2009)

SOFTWARE RESEARCH, INC. Tcat C/C++. 2001a.
Disponível em http://www.soft.com/ (Acessado em 20/02/2009)

SOFTWARE RESEARCH, INC. Tcat for Java. 2001b.
Disponível em http://www.soft.com/ (Acessado em 11/08/2005)

SOMMERVILLE, I. *Software engineering*. 7 ed. Pearson Addison Wesley, 2004.

SOUTER, A. L.; POLLOCK, L. L. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, v. 29, n. 11, p. 1005–1018, 2003.

SZE, L. Atacobol: A cobol test coverage analysis tool and its applications. In: *ISSRE'2000 – International Symposium on Software Reliability Engineering*, San Jose, CA, 2000.

TAENTZER, G. Agg: A graph transformation environment for modeling and validation of software. In: PFALTZ, J. L.; NAGL, M.; BÖHLEN, B., eds. *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, Springer, 2003, p. 446–453 (*Lecture Notes in Computer Science*, v.3062).

TAI, K.-C.; DANIELS, F. J. Interclass test order for object-oriented software. *Journal of Object-Oriented Programming*, v. 12, n. 4, p. 18–25,35, 1999.

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

TARR, P.; OSSHER, H.; HARRISON, W.; STANLEY M. SUTTON, J. N degrees of separation: multi-dimensional separation of concerns. In: *(ICSE'99) Proceedings of the 21st International Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, p. 107–119.

TELCORDIA TECHNOLOGIES, INC. Telcordia Software Visualization and Analysis Toolsuite (xSuds). 1998.
Disponível em http://xsuds.argreenhouse.com/ (Acessado em 20/02/2009)

TELCORDIA TECHNOLOGIES, INC. Telcordia Software Visualization and Analysis Toolsuite (xSuds). 2005.
Disponível em http://xsuds.argreenhouse.com/ (Acessado em 20/02/2009)

THE ASPECTJ TEAM The AspectJ programming guide. Xerox Croporation, 2002.
Disponível em http://www.eclipse.org/aspectj/doc/released/progguide/index.html (Acessado em 20/02/2009)

TONELLA, P. Evolutionary testing of classes. In: *(ISSTA'04) Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM Press, 2004, p. 119–128.

TSAI, B.-Y.; STOBART, S.; PARRINGTON, N.; MITCHELL, I. Automated class testing using threaded multi-way trees to represent the behaviour of state machines. *Annals of Software Engineering*, v. 8, n. 1-4, p. 203–221, 1999a.

TSAI, B.-Y.; STOBART, S.; PARRINGTON, N.; MITCHELL, I. A state-based testing approach providing data flow coverage in object-oriented class testing. In: *12th International Software Quality Week*, California, USA: Software Research Institute, 1999b, p. 18.
Disponível em http://www.stickyminds.com/getfile.asp?ot=XML&id=1461&fn=XML0614%2Edoc (Acessado em 20/02/2009)

TURNER, C. D.; ROBSON, D. J. The state-based testing of object-oriented programs. In: *(ICSM'93) Proceedings of the Conference on Software Maintenance*, Washington, DC, USA: IEEE Computer Society, 1993, p. 302–310.

TURNER, C. D.; ROBSON, D. J. A state-based approach to the testing of class-based programs. *Software - Concepts and Tools*, v. 16, n. 3, p. 106–112, 1995.

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

W3C Extensible markup language (XML). World Wide Web Consortium, 2009.
Disponível em http://www.w3.org/XML/

WHITTLE, J.; JAYARAMAN, P. K. Mata: A tool for aspect-oriented modeling based on graph transformation. In: GIESE, H., ed. *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, Springer, 2007, p. 16–27 (*Lecture Notes in Computer Science*, v.5002).

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

YAMAZAKI, Y.; MASUHARA, H.; SAKURAI, K.; HASHIURA, H.; MATSUURA, S.; KOMIYA, S. A unit testing framework for aspects without weaving. In: *(AOSD'2005) Fourth International Conference on Aspect-Oriented Software Development – Workshop On Testing Aspect Oriented Programs*, Chicago, Illinois – USA, 2005.

ZHANG, S.; ZHAO, J. On identifying bug patterns in aspect-oriented programs. In: *(COMPSAC'07) Proceedings of the 31st Annual International Computer Software and Applications Conference*, Washington, DC, USA: IEEE Computer Society, 2007, p. 431–438.

ZHAO, J. *Towards a metrics suite for aspect-oriented software*. Relatório Técnico SE-136-25, Information Processing Society of Japan, 2002.
Disponível em http://cse.sjtu.edu.cn/~zhao/pub/pdf/ipsj-tr-se-136-25.pdf (Acessado em 20/02/2009)

ZHAO, J. Data-flow-based unit testing of aspect-oriented programs. In: *(COMPSAC'03) Proceedings of the 27th Annual International Computer Software and Applications Conference*, IEEE Computer Society, 2003a, p. 188–197.

ZHAO, J. *Measuring coupling in aspect-oriented systems*. Relatório Técnico SE-142-6, Information Processing Society of Japan, 2003b.
Disponível em http://cse.sjtu.edu.cn/~zhao/pub/pdf/metrics04lbp.pdf (Acessado em 20/02/2009)

ZHOU, Y.; RICHARDSON, D.; ZIV, H. Towards a practical approach to test aspect-oriented software. In: BEYDEDA, S.; GRUHN, V.; MAYER, J.; REUSSNER, R.; SCHWEIGGERT, F., eds. *(TECOS'2004) Testing of Component-Based Systems and Software Quality – Workshop Testing Component-Based Systems*, Erfurt – Germany, 2004, p. 1–16.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. *ACM Comput. Surv.*, v. 29, n. 4, p. 366–427, 1997.