Avaliação da abordagem incremental no teste de integração de programas orientados a aspectos

Reginaldo Ré[1] **, Paulo Cesar Masiero**[2]

[1]Centro Universitário Barão de Mauá
Rua Ramos de Azevedo, 423 – Ribeirão Preto – SP – CEP 14090-180

[2]Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
Av. Trabalhador São-carlense, 400 – São Carlos – SP – CEP 13560-970

reginaldor@baraodemaua.br, masiero@icmc.usp.br

Abstract. *A study about ordering classes and aspects to minimize stubs in integration tests is presented. A dependence type model among classes and aspects is defined considering the syntax constructions and the semantics of AspectJ. The strategy proposed by Briand et al. [2003] is used in two cases of integration: the incremental integration, which order classes and aspects separately (often indicated as more adequate in literature), and an approach considering the combined integration of classes and aspects. The results obtained with both approaches are discussed, showing that a refined strategy is better in some cases.*

Keywords: Testing aspect-oriented programs, aspect-oriented programming, integration testing, stub minimization.

Resumo. *Um estudo sobre a ordenação de classes e aspectos para minimização de stubs em teste de integração é apresentado. Um modelo de tipos de dependência entre classes e aspectos é definido a partir das construções sintáticas e da semântica do AspectJ. A estratégia de Briand et al. [2003] é usada em dois casos de integração: a integração incremental, que ordena classes e aspectos separadamente e é freqüentemente indicada na literatura como a mais adequada; e uma abordagem de integração conjunta. Os resultados da avaliação entre a abordagem incremental e a abordagem conjunta são discutidos, mostrando que em alguns casos uma estratégia mais refinada de integração é mais adequada.*

Palavras-chave: teste de programas orientados a aspectos, programação orientada a aspectos, teste de integração, minimização de módulos pseudo-controlados.

1. Introdução

O maior problema encontrado durante o teste de integração de programas orientados a objetos 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 [Briand et al. 2003; Kung et al. 1995; Le Traon et al. 2000; Tai e Daniels 1999] foram feitas para ordenar a implementação e teste de classes com a intenção de minimizar o número de *stubs* e, conseqüentemente, minimizar o esforço.

A programação orientada a aspectos (OA) é uma proposta que tem por objetivo facilitar o desenvolvimento de sistemas por meio da separação de diferentes interesses relacionados às características funcionais e não funcionais do software [Gradecki et al. 2003; Kiczales et al. 2001, 1997]. Não obstante essa nova abordagem possua diversas vantagens [Alexander e Bieman 2005; Kiczales et al. 1997; Kienzle et al. 2003], ela também apresenta novos desafios para desenvolvimento de software [Alexander e Bieman 2005; Mortensen e Alexander 2005; Störzer e Krinke 2003]. Várias propostas para desenvolvimento de software orientados a aspectos têm sido feitas [Aldawud et al. 2003; Baniassad e Clarke 2004; Stein et al. 2002; Thompson e Odgers 1999], em um esforço para solucionar as questões decorrentes desse novo paradigma. No contexto de teste de programas OA, várias estratégias propõem o teste incremental dos programas que utilizam aspectos [Ceccato et al. 2005; Zhou et al. 2004]. Nessa abordagem, primeiramente as classes-base devem ser implementadas e testadas e posteriormente os aspectos devem ser adicionados e testados um a um para que se possa realizar as atividades do teste de integração.

Zhou et al. [2004] propõem uma abordagem para o teste de unidade, teste de integração e teste de sistema para software orientado a aspectos que consiste de quatro passos: i) desenvolvimento e teste de classes-base, para isolar e eliminar erros não relacionados a aspectos; ii) teste das classes-base com cada aspecto em separado, correspondendo ao teste de unidade de cada aspecto em particular; iii) os aspectos são incrementalmente combinados com as classes-base e o teste é realizado, correspondendo ao teste de integração; iv) e, as classes-base e todos os aspectos são testados, correspondendo ao teste de sistema.

Ceccato et al. [2005] propõem uma abordagem de teste incremental na qual as classes-base devem ser testadas inicialmente sem considerar os aspectos, com o mesmo intuito de revelar erros que não são relacionados aos aspectos. Gradativamente os aspectos são adicionados, o software é testado com o conjunto de casos de teste projetado para a verificação das classes-base e um novo conjunto de casos de teste deve ser preparado especificamente para averiguar as alterações no comportamento das classes-base na presença dos aspectos. Como as classes-base devem ser implementadas inicialmente, os autores argumentam que um dos problemas da abordagem é a necessidade da criação de *stubs* e *drivers* com o intuído de simular o comportamento de aspectos dos quais as classes-base são dependentes, como por exemplo, persistência de dados.

Embora vários trabalhos apresentem discussões sobre teste de aspectos independentes entre si e totalmente ortogonais, algumas pesquisas, apresentam exemplos de sistemas orientados a aspectos projetados com duas características interessantes: aspectos dependentes entre si e aspectos que não são totalmente ortogonais [Ceccato et al. 2005; Colyer et al. 2004; Douence et al. 2004; Filman e Friedman 2000; Kienzle e Guerraoui 2002; Kienzle et al. 2003; The AspectJ Team 2002]. The AspectJ Team [2002], apresenta um exemplo de um sistema de controle de ligações telefônicas que

possui aspectos que usam aspectos. Um aspecto é responsável pela contagem do tempo de chamadas de longa distância, que usa o aspecto que faz as chamadas. Por sua vez, um aspecto que é responsável pela tarifação usa o aspecto de temporização e também o aspecto que faz as chamadas. Um exemplo de aspecto que não é totalmente ortogonal pode ser encontrado no trabalho de Kienzle e Guerraoui [2002], em que um sistema tem consciência de um aspecto responsável por controle de transações. O sistema saber quando, por exemplo, uma transação falha, para que possa corrigir o problema e garantir a integridade dos dados.

Em teste de integração, aspectos com essas características podem apresentar ciclos de dependência entre si e, em casos específicos, podem existir ciclos de dependência não somente entre aspectos, mas também entre classes e aspectos. Dessa forma, a necessidade de criação de *stubs* para o teste de sistemas orientados a aspectos que apresentem ciclos de dependência deve ser considerada, fazendo com que uma estratégia de ordenação de classes seja utilizada para que o número de *stubs* seja minimizado. A ordem do teste de integração das classes e aspectos pode ser usada também como uma forma de indicar a sua ordem de implementação.

Não foram encontrados na literatura trabalhos que tratem especificamente o problema de formação de ciclos em sistemas orientadas a aspectos propondo uma estratégia de ordenação de implementação e teste de classes e aspectos. Neste trabalho, os tipos de dependência existentes em orientação a aspectos são estudados, duas alternativas de ordenação de classes em sistemas orientados a aspectos são aplicadas a um exemplo contendo aspectos e os resultados são avaliados. O exemplo e todo o estudo deste trabalho foi feito usando a linguagem AspectJ, portanto, as construções sintáticas analisadas são específicas dessa tecnologia. A utilização da estratégia de Briand et al. [2003] mostra que em alguns casos pode-se implementar e integrar classes e aspectos conjuntamente, em outros casos, aspectos devem ser implementados e integrados antes das classes e, no caso geral, os aspectos devem ser implementados e integrados posteriormente às classes.

O restante do trabalho é organizado da seguinte forma. Na Seção 2 é mostrada a evolução das pesquisas em torno da ordenação de classes para o teste de integração, com ênfase na proposta de Briand et al. [2003]. A investigação sobre os tipos de dependência entre aspectos e classes é apresentada na Seção 3, na qual as construções sintáticas e a semântica usada pelo AspectJ são discutidas. Nessa seção também é mostrada a forma de mapeamento das construções da orientação a aspectos em um diagrama de relacionamento de objetos (ORD, *Object Relation Diagram*). Ainda na Seção 3, um estudo mapeando um conjunto de classes e aspectos um ORD e usando a estratégia de Briand et al. [2003] em duas diferenes abordagens de integração é apresentado. A Seção 4 e a Seção 5 finalizam este trabalho com uma discussão sobre os resultados obtidos, as conclusões e os trabalhos futuros.

2. Estratégias de ordenação de classes

O trabalho de Kung et al. [1995] foi um dos primeiros a apresentar uma solução para o problema de ciclos de dependência e discutir este tipo de problema no teste de integração para programas orientados a objetos. Os autores propuseram um diagrama chamado ORD para representar as dependências dos relacionamentos de herança “I”,

de agregação "Ag" e de associação "As" entre classes (Figura 1). A estratégia apresentada nesse trabalho é baseada em dois conceitos principais: i) *cluster*, que é o conjunto com o máximo de vértices mutuamente alcançáveis no dígrafo (um ORD é analisado como um dígrafo); ii) e, a quebra de ciclos, que é a remoção temporária de uma aresta com o objetivo de tornar o dígrafo acíclico. Na estratégia apresentada por Kung et al. [1995], somente as arestas que representam associações entre as classes devem ser removidas, já que herança e agregação apresentam além do acoplamento de controle, acoplamento de dados e dependência de código. A ordem de teste das classes é dada ao se efetuar uma ordenação topológia do dígrafo acíclico.

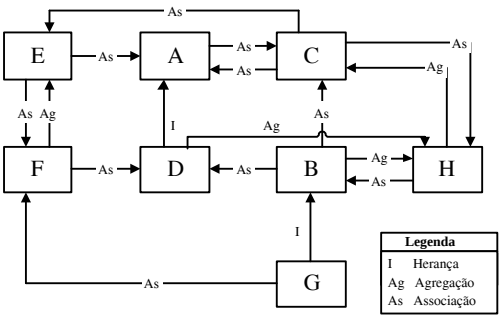

Figura 1. Exemplo do ORD estudado por Briand et al. [2003].

O trabalho de Tai e Daniels [1999] toma como base o trabalho de Kung et al. [1995], permitindo apenas a eliminação de arestas que representam associação. Os autores propõem um algoritmo para a escolha de quais arestas do ORD devem ser eliminadas para quebra dos ciclos de dependência, ao invés de escolher aleatoriamente uma aresta. O algoritmo 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: i) 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; ii) 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 sub-ótima ao indicar a geração de *stubs* desnecessários.

Le Traon et al. [2000] propõem um outro tipo de representação gráfica das dependências entre classes, denominada TDG (*Test Dependency Graph*). A proposta usa o algoritmo de Tarjan [1972] para identificar SCCs (componentes fortemente conexos), que podem ser triviais, quando os SCCs são compostos de apenas um vértice, e não triviais quando os SCCs são compostos por mais de um vértice. O algoritmo de Tarjan [1972] 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 [1972] e da remoção de arestas, porém o resultado da aplicação da proposta de Le Traon et al. [2000] é não determinístico e, portanto, existem várias possibilidades de ordenação das classes.

A abordagem de Le Traon et al. [2000] apresenta dois pontos bastante diferentes de Kung et al. [1995] e de Tai e Daniels [1999]: i) arestas que representam herança ou agregação podem ser removidas para que os ciclos sejam quebrados; ii) e, o algoritmo minimiza 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].

Briand et al. [2003] apresentam uma estratégia que utiliza algumas características das propostas de Le Traon et al. [2000] e Tai e Daniels [1999]: a) usa o algoritmo de Tarjan [1972] recursivamente para identificar SCCs; b) e, associa pesos às arestas que representam dependência de associação para indicar quais devem ser removidas e quebrar os ciclos de dependência. A estratégia é determinística e minimiza o número de *stubs* específicos, ao contrário de Le Traon et al. [2000], e o resultado não é sub-ótimo como Tai e Daniels [1999].

A estratégia inicia com a aplicação do algoritmo de Tarjan [1972] em um grafo construído a partir de um ORD. Deve-se, recursivamente, remover arestas dos SCCs não triviais para quebrar os ciclos e aplicar o algoritmo de Tarjan [1972]. A aresta que deve ser removida é aquela que apresenta maior peso, obtido por meio da multiplicação das arestas de entrada do vértice origem pelas arestas de saída do vértice destino de cada aresta de associação. Esses passos devem ser seguidos até não existirem mais SCCs não triviais.

A Figura 2 mostra um exemplo da aplicação da estratégia, em que é possível observar em um ORD e as arestas que foram removidas para a quebra dos ciclos. Nesse exemplo, após a aplicação inicial do algoritmo de Tarjan [1972] são encontrados 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

No exemplo, a remoção da aresta A->C ou a remoção da aresta H->B é indiferente pois ambas geram o mesmo número de *stubs* já que elas possuem o mesmo peso. Ao se eliminar a aresta A->C o algoritmo de Tarjan [1972] 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 seqüência de aplicação do algoritmo de Tarjan [1972] de acordo com a eliminação das arestas resulta na ordem de implementação e teste das classes, mostrada na Figura 3(a). Na Figura 3(b) apresenta-se os *stubs* necessários para implementar as classes.

3. Estratégia de ordenação de classes e aspectos

Em orientação a aspectos, os tipos de dependência entre aspectos e entre classes e aspectos são diferentes dos tipos de dependência existentes em orientação a objetos. O tipo de dependência gerada por relacionamentos de herança e por relacionamentos de associação é encontrada em aspectos, ao contrário da agregação, que não existe no contexto da orientação a aspectos. Da mesma forma, a semântica e os diferentes

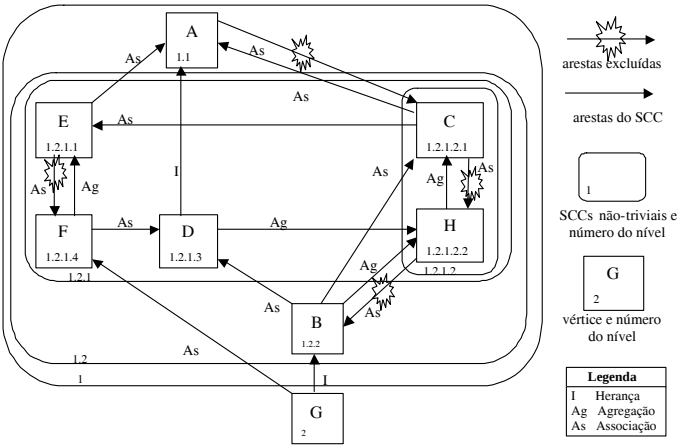

Figura 2. Aplicação da estratégia de Briand et al. [2003].

```
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}
```

(a) Aplicação recursiva do algoritmo de Tarjan.

1. A é testado com *stub*(C, A);
2. E é testado com A e *stub*(F, E);
3. C é testado com A, E e *stub*(H, C);
4. H é testado com C e *stub*(B, H);
5. D é testado com A e H;
6. F é testado com E e D;
7. B é testado com C, D e H;
8. G é testado com F e B.

(b) Ordem de teste e *stubs* gerados.

Figura 3. Aplicação da estratégia no ORD da Figura 2.

elementos sintáticos da orientação a aspectos não possuem correspondente similar em orientação a objetos. Dois pontos importantes devem ser observados em decorrência dos novos tipos de dependência característicos da orientação a aspectos à luz do teste de integração: a) quais são esses novos tipos de dependência e quais suas principais características; b) e, como representar essa dependência no ORD. Este trabalho

apresenta um modelo de tipos de dependência, que visa a auxiliar o entendimento do relacionamento entre aspectos e classes no contexto de teste de software, e uma adaptação do ORD a partir do modelo de tipos de dependência.

3.1. Definição e representação de dependências entre aspectos e classes

Os tipos de dependência são caracterizados por conjuntos de junção, adendos, declarações inter-tipo, herança e métodos pertencente aos aspectos. De forma geral, os aspectos são dependentes das classes porque as classes não têm conhecimento de quais aspectos a afetam, um conceito denominado inconciência (do inglês, *obliviousness*) [Filman e Friedman 2000]. No entanto, em alguns casos específicos identificados na literatura [Ceccato et al. 2005; Kienzle e Guerraoui 2002; The AspectJ Team 2002], existem classes que possuem conhecimento dos aspectos que a entrecortam, denominado por Filman e Friedman [2000] de inconsciência incompleta. A inconsciência imcompleta gera dependência circular [Kienzle et al. 2003] entre as classes e os aspectos, que é tratada nesse trabalho como dependência bidirecional.

Aspectos sempre dependem apenas dos pontos de junção definidos pelos conjuntos de junção, com exceção dos conjuntos de junção com verificação condicional. Os pontos de junção podem ser definidos mediante uma combinação de diversos elementos sintáticos contidos na declaração do conjunto de junção, que é composto pelo tipo do conjunto de junção e sua assinatura. Portanto, para estabelecer uma relação de dependência é necessária uma análise de todos os elementos que compõem a declaração de um conjunto de junção e não apenas a análise separada de cada elemento. Por exemplo, pode-se notar que é mais difícil detectar dependências em um conjunto de junção que contém os elementos sintáticos `execution`, `args`, `within` e `if` simultaneamente, ou usa operador lógico de negação `!`, do que em um conjunto de junção com assinatura simples, conforme ilustrado na linha 7 da Figura 4.

```
1  public class A {
2   public void imprimeA(){
3     System.out.println("ImprimeA() invocado;");
4   }
5  }
6  public aspect Aa {
7    pointcut executionAImprimeA(): execution (public void A.imprimeA());
8
9    before(): executionAImprimeA(){
10     System.out.println("Before xecutionAImprimeA()");
11   }
12 }
```

Figura 4. Exemplo de dependências geradas por `execution`.

Os tipos de conjuntos de junção `get`, `set`, `handler`, `staticinitialization`, `preinitialization`, `initialization`, `cflow`, `cflowbelow`, `within` e `whitincode` geram dependências explícitas a classes e aspectos selecionados por suas assinaturas, bem como `target`, `args` e os tipos de associações de aspectos `pertarget`, `percflow` e `percflowbelow`.

Alguns tipos de conjuntos de junção geram dependências apenas entre aspectos, como por exemplo `adviceexecution`, que faz com que o aspecto dependa de todos os outros aspectos do sistema que possuam adendos definidos. Essa característica de dependência também ocorre quando define-se um conjunto de junção em

função de outro conjunto de junção. Por exemplo, um aspecto com `pointcut pcExecutionAImprimeA(): Aa.executionAImprimeA()` é dependente do aspecto "Aa", porque o aspecto "Aa" possui o conjunto de junção `Aa.executionAImprimeA()`.

O conjunto de junção com verificação condicional gera dependência de associação no paradigma de orientação a aspectos, semelhante à orientação a objetos, porque utiliza o comando `if`, que avalia uma expressão booleana que, por sua vez, pode invocar ou usar métodos e atributos de objetos e de aspectos. Por essa característica, o conjunto de junção com verificação condicional difere dos demais tipos de conjuntos de junção, pois gera dependência do ponto de junção selecionado e também do objeto ou do aspecto usado na assinatura do conjunto de junção. A Figura 5 ilustra um caso de conjunto de junção que gera uma dependência do tipo "P" com a classe "A" (linha 3) e gera uma dependência do tipo "As" com a classe "B" (linha 5).

```
1 public aspect Aa {
2  pointcut condicionalBImprimeB(B b):
3    execution (public void A.imprimeA())
4    || (target(b)
5    && if (b.verifica()));
6  ...
7  ...
8 }
```

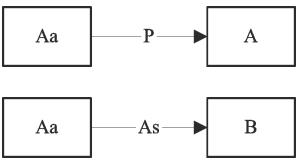

Figura 5. Exemplo de dependências geradas por conjuntos de junção.

Os adendos e métodos dos aspectos geram dependências semelhantes à orientação a objetos, porque é possível declarar variáveis locais, declarar atributos e invocar métodos de instâncias de outras classes e de aspectos, como pode-se notar na linha 8 da Figura 6 que gera uma dependência do tipo "As". A herança por sua vez, pode ocorrer quando um aspecto é uma especialização de um outro aspecto abstrato, ou quando um aspecto é uma especialização de uma classe, por exemplo no caso mostrado na linha 4 da Figura 6. Em ambos os casos, pode-se considerar que a herança da orientação a aspectos tem as mesmas características da herança em orientação a objetos e é representada por uma aresta do tipo "I".

```
1  public abstract aspect Ab {
2    public abstract void imprime();
3  }
4  public aspect Aa extends Ab{
5    ...
6    ...
7    before(B b): condicionalBImprimeB(b){
8        if (b.verifica())
9            imprime(b);
10   }
11   public void imprime(){
12       System.out.println("CondicionalBImprimeB");
13   }
14 }
```

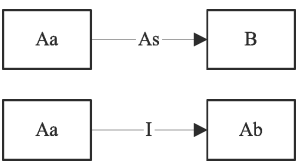

Figura 6. Exemplo de dependências geradas por adendos, herança e chamada de métodos.

Na orientação a aspectos pode-se incluir atributos e métodos em classes e aspectos, e criar ou alterar relacionamentos de herança entre aspectos e classes por

meio de declarações inter-tipo. Para um aspecto introduzir métodos ou atributos em classes e aspectos ele deve explicitamente indicar o método e a classe, por exemplo, para introduzir um método na classe "A" um aspecto declara `public void A.imprimeB()`, como ilustrado na linha 14 da Figura 7.

Para declarar uma herança entre classes, entre aspectos ou entre aspectos e classes, um aspecto deve possuir uma declaração do tipo `declare parents`, como apresentado na linha 12 da Figura 7. Por fim, a declaração que estabelece a ordem em que adendos de diferentes aspectos afetam um mesmo ponto de junção, `declare precedence` gera dependência somente entre aspectos. Qualquer dependência gerada por declarações inter-tipo é representada no ORD por uma aresta do tipo "IT".

```
1  public class A {
2    public void imprimeA(){
3      System.out.println("ImprimeA() invocado;");
4    }
5  }
6  public class B {
7    public void imprimeB(){
8      System.out.println("ImprimeB() invocado;");
9    }
10 }
11 public aspect Aa {
12   declare parents: A extends B;
13
14   public void A.imprimeB(){
15     super.imprimeB();
16     imprimeA();
17   }
18 }
```

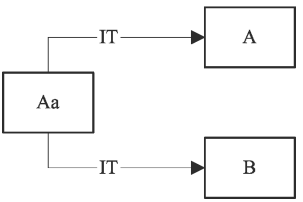

Figura 7. Exemplo de dependências geradas por declarações inter-tipo.

Apesar das declarações inter-tipo apresentarem um tipo de dependência aparentemente simples, o uso desse recurso tem forte impacto nas classes-base [Störzer e Krinke 2003] e nos testes [Alexander e Bieman 2005; Mortensen e Alexander 2005], já que a alteração na estrutura do sistema pode ser bastante significativa. Portanto, é importante representar esse tipo de dependência no ORD, que é usado em teste de regressão para identificar quais classes são afetadas em caso de alterações de classes que fazem parte do sistema [Kung et al. 1995; Milanova et al. 2002].

Considerando a discussão sobre os vários elementos sintáticos da orientação a aspectos, conclui-se que os tipos de dependência que diferem da orientação a objetos podem ser categorizadas em dois grupos: i) dependências geradas por conjuntos de junção; ii) e, dependências geradas por declarações inter-tipo. Os conjuntos de junção e as declarações inter-tipo geram tipos de dependência que não são suportadas em um ORD tradicional e são representadas em um ORD estendido por arestas do tipo "P" e do tipo "IT", respectivamente.

3.2. Aplicação da estratégia de Briand et al. para a integração conjunta de classes e aspectos

Como estudo de caso para obter evidências sobre qual estratégia de teste de integração é mais adequada no teste de sistemas orientados a aspectos, foi aplicado a

estratégia de Briand et al. [2003] em um ORD que contém classes e aspectos, apresentado na Figura 8. O exemplo de ORD foi adaptado de Briand et al. [2003] e possui todos os tipos de dependência da orientação a objetos e da orientação a aspectos: a) aspectos que dependem de classes; b) aspectos que dependem de aspectos; c) e, classes que dependem de aspectos. O aspecto "Aa" e as classes "B", "D" e "F", como pode ser observado na Figura 8, possuem dependência cíclica. O aspecto "Aa" foi adicionado ao ORD para que fosse possível estudar o comportamento da estratégia de Briand et al. [2003] quando existe dependência bidirecional, como já comentado anteriormente (persistência). As classes são os retângulos "A", "B", "C", "D", "E", "F", "G" e "H", enquanto os aspectos são os retângulos "Aa", "Ab", "Ac", "Ad", "Ae", "Af", "Ag", "Ah" e "Ai". As classes e os aspectos mostrados no ORD da Figura 8 foram implementados, simulados e testados segundo a ordem de teste resultante do uso da estratégia de Briand et al. [2003].

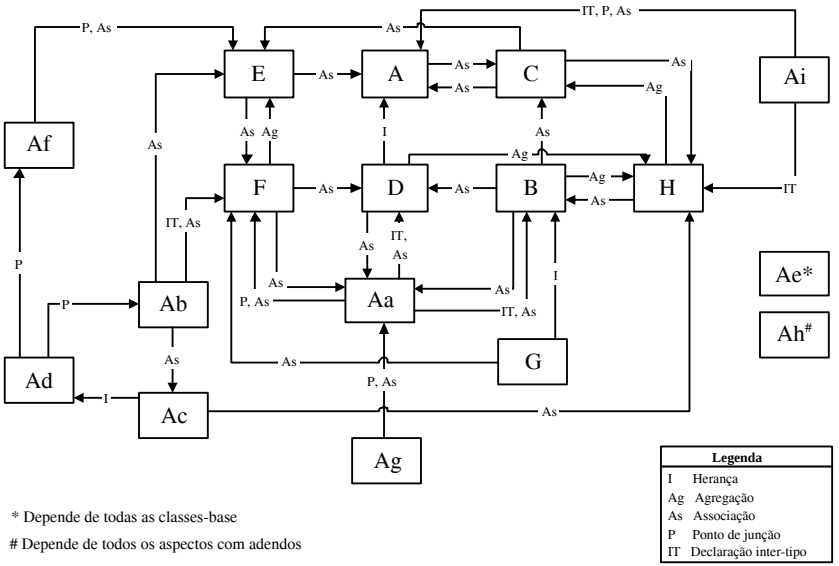

Figura 8. Exemplo de ORD com classes e aspectos.

Neste trabalho, considera-se que as dependências representadas por arestas do tipo "P" e do tipo "IT" têm o mesmo nível de acoplamento que as dependências representadas por arestas de associação, podendo ser eliminadas para quebrar ciclos dos aspectos e classes. Portanto, considerando que um mesmo aspecto pode gerar simultaneamente esses três tipos de dependência com um outro aspecto ou classe, uma mesma aresta pode ser nomeada como "As", "P" e "IT" no ORD estendido, como pode ser visto na Figura 8.

Na Figura 9 é mostrado o resultado da aplicação da estratégia de Briand et al. [2003] no ORD da Figura 8. Conforme pode ser observado, a aplicação do algoritmo de Tarjan [1972] no ORD original resulta na seguinte seqüência de teste: [SCC1, Af,

SCC2, G, Ae, Ag, Ai, Ah]. Para obter o resultado final da ordem de teste e dos *stubs* necessários, mostrado na Figura 10(a), o algoritmo foi aplicado recursivamente nos SCCs não triviais SCC1={Aa, B, C, A, E, F, D, H} e SCC2={Ab, Ac, Ad}. É interessante notar que o aspecto "Aa" deve ser testado antes de algumas classes e que existe um ciclo de dependência constituído apenas de aspectos, o que implica na criação de um *stub* de aspecto (*stub*(Ac, Ab)). O custo de teste em números de *stubs* de classes, de aspectos e de métodos é apresentado na Figura 10(b).

```
ORD={Aa, Ab, Ac, Ad, Ae, Af, Ag, Ah, Ai, A, B, C, D, E, F, G, H}
1. {Aa, B, C, A, E, F, D, H} : aresta de maior peso Aa->B=12
  1.1 {Aa, B, C, A, E, F, D, H} : aresta de maior peso H->B=12
    1.1.1 {Aa, C, A, E, F, D, H} : aresta de maior peso A->C=9
      1.1.1.1 {A}
      1.1.1.2 {Aa, C, E, F, D, H} : aresta de maior peso Aa->F=6
        1.1.1.2.1 {E}
        1.1.1.2.2 {Aa, D, H, C} : aresta de maior peso H->D=4
          1.1.1.2.2.1 {H, C} : aresta de maior peso C->H=1
            1.1.1.2.2.1.1 {C}
            1.1.1.2.2.1.2 {H}
          1.1.1.2.2.2 {Aa, D} : aresta de maior peso Aa->D=1
            1.1.1.2.2.2.1 {Aa}
            1.1.1.2.2.2.2 {D}
        1.1.1.2.3 {F}
    1.1.2 {B}
2. {Af}
3. {Ab, Ac, Ad} : aresta de maior peso Ab->Ac=1
  3.1 {Ab}
  3.2 {Ad}
  3.3 {Ac}
4. {G}
5. {Ae}
6. {Ag}
7. {Ai}
8. {Ah}
```

Figura 9. Aplicação recursiva da estratégia no ORD da Figura 8.

No exemplo em que a estratégia foi aplicada, o *stub* implementado é gerado para simular o comportamento de um relacionamento de associação que contém a declaração de um aspecto e a implementação de um método. Porém, podem existir outros casos em que *stubs* devem ser utilizados para simular dependências referentes a conjuntos de junção e declarações inter-tipo.

3.3. Aplicação da estratégia de Briand et al. para integração incremental

Além do uso da estratégia de [Briand et al. 2003] para a ordenação de classes e aspectos de forma conjunta, ela também pode ser utilizada para a determinação da ordem de teste de programas orientados a aspectos segundo a abordagem incremental. Esta abordagem quebra sistematicamente as dependências bidirecionais fazendo com que as classes sejam mapeadas em um ORD e os aspectos em outro ORD. A Figura 1 apresenta um exemplo de como fica o ORD do sistema representado pela

Figura 8, e a Figura 3 mostra os resultados da aplicação da estratégia de Briand et al. [2003] apenas nas classes.

1. A é testado com *stub*(C, A);
2. E é testado com A e *stub*(F, E);
3. C é testado com A, E e *stub*(H, C);
4. H é testado com C, *stub*(B, H) e *stub*(D, H);
5. Aa é testado com *stub*(B, Aa), *stub*(D, Aa) e *stub*(F, Aa);
6. D é testado com A, H e Aa;
7. F é testado com D, E e Aa;
8. B é testado com C, D, H e Aa;
9. Af é testado com E;
10. Ab é testado com E, F e *stub*(Ac, Ab)
11. Ad é testado com Ab e Af;
12. Ac é testado com H e Ad;
13. G é testado com B e F;
14. Ae é testado com todas as classes-base;
15. Ag é testado com Aa;
16. Ai é testado com A e H;
17. Ah é testado com todos os aspectos.

(a) Ordem de teste e *stubs* gerados.

	Classe/ Aspecto	Método
stub(C, A)	1	4
stub(F, E)	1	4
stub(H, C)	1	4
stub(B, H)	1	4
stub(D, H)	1	4
stub(B, Aa)	1	3
stub(D, Aa)	1	3
stub(F, Aa)	1	3
stub(Ac, Ab)	1	1
Total	9	30

(b) Custo do teste em número de *stubs*.

Figura 10. Ordem e custo de teste em número de stubs.

Após os testes das classes-base os aspectos devem ser integrados e incrementalmente testados. A estratégia de Briand et al. [2003] também pode ser usada para a ordenação do teste de aspectos. No exemplo da Figura 11, existe apenas um ciclo entre aspectos que deve ser quebrado por meio da criação de um *stub*, conforme ilustrado na Figura 12. Quando a estratégia de [Briand et al. 2003] é aplicada primeiramente nas classes e depois nos aspectos o custo é de 4 *stubs* de classes (*stub*(C, A), *stub*(F, E), *stub*(H, C), *stub*(B, H)) e um *stub* de aspecto (*stub*(Ac, Ab)).

4. Análise dos resultados obtidos

Considerando o critério de minimização de stubs, nota-se uma tendência geral em desenvolver e testar as classes antes dos aspectos, o que de certa forma confirma a intuição dos pesquisadores que usam essa estratégia [Ceccato et al. 2005; Zhou et al. 2004]. Mas da análise cuidadosa dos resultados dos dois estudos de casos realizados, conforme relatado neste artigo, emerge uma situação que exige uma análise mais refinada.

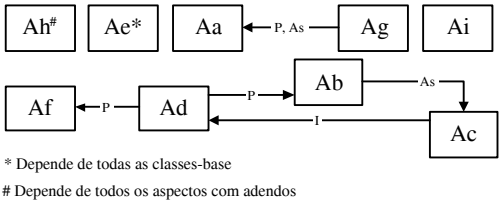

Figura 11. Exemplo de um ORD contendo apenas aspectos.

ORD={Aa, Ab, Ac, Ad, Ae, Af, Ag, Ah, Ai}
1. {Aa}
  1.1 {Af}
  1.2 {Ab, Ac, Ad} : aresta de maior peso A->C=1
    1.2.1 {Ab}
    1.2.1 {Ad}
    1.2.1 {Ac}
  1.2.1 {Ae}
  1.2.1 {Ag}
  1.2.1 {Ai}
  1.2.1 {Ah}
2. {G}

(a) Aplicação recursiva do algoritmo de Tarjan.

1. Aa é testado com B, D e F;
2. Af é testado com E;
3. Ab é testado com E, F e *stub*(Ac, Ab);
4. Ad é testado com Ab;
5. Ac é testado com Ad;
6. Ae é testado com as classes-base;
7. Ag é testado com Aa;
8. Ai é testado com A e H
9. Ah é testado com os aspectos.

(b) Ordem de teste e *stubs* gerados.

Figura 12. Aplicação da estratégia de [Briand et al. 2003] apenas em aspectos.

Primeiramente, nota-se que a estratégia incremental gera menos *stubs*, mas só pode ser aplicada sem problemas quando as classes são completamente inconscientes dos aspectos, isto é, os aspectos implementados na aplicação são todos transversais. O que o estudo acrescentou é que nesse caso, a estratégia de Briand et al. [2003] pode ser útil para ordenar a integração dos aspectos. A necessidade de integração de aspectos que entrecortam outros aspectos não é mencionada na literatura. A intuição dos autores é que esta estratégia deve ser mais eficiente, mas há necessidade de estudos mais aprofundados e experimentos para comprovar a intuição.

Quando há aspectos bidirecionais, como os de Kienzle et al. [2003], de Rashid e Chitchyan [2003], e de The AspectJ Team [2002], o emprego da estratégia incremental exige que uma ou mais classes tenham implementações alternativas que poderão ser modificadas quando a classe for integrada ao aspecto. Por exemplo, no caso do aspecto de persistência, pode-se criar objetos na memória principal, sem persistên-

cia, desenvolver e testar as classes. Isso pode ser mais trabalhoso que criar *stubs* de aspectos porque na integração haverá re-trabalho e execução de testes de regressão.

Com aspectos bidirecionais no modelo, a estratégia de Briand et al. [2003] encontrou uma tendência a que esses aspectos sejam desenvolvidos e testados logo no início, antes de qualquer classe que dele tenha consciência. Esta estratégia no geral, gera mais *stubs*, mas evita re-trabalho e teste de regressão. Considerando a Figura 10(b), nota-se que o aspecto "Aa" é desenvolvido primeiro e criam-se *stubs* para as classes "B", "D" e "F". Entretanto, se houver um *framework* que implemente o interesse bidirecional [Rashid e Chitchyan 2003; Vanhaute et al. 2001], esta estratégia pode ser eficiente, pois implementa-se as classes na ordem do algoritmo e instancia-se o *framework* no momento em que for necessário, porque em princípio ele já está testado. Outra estratégia, conforme Kienzle et al. [2003], é quebrar a dependência bidirecional, transformando o aspecto em dois, cada um dependente da classe, que pode então ser desenvolvida e testada primeiramente. Ele mostra como isso pode ser feito para o interesse de controle de transação, mas não está claro se isso pode ser feito para todos os aspectos em que o classes-base têm consciência do aspecto. Se isso for feito, aplica-se a estratégia incremental.

Outro ponto interessante é que, em casos em que uma aplicação é implementada com muitos aspectos que se entrecortam uns aos outros, pode haver ciclos entre os aspectos e, nesse caso, a estratégia de Briand et al. [2003] com as regras de precedência adotadas neste artigo, indica a ordem de desenvolvimento e teste dos aspectos que minimiza o número de *stubs* de aspectos.

Adicionalmente, considerando a estratégia conjunta, há situações em que o resultado do algoritmo indica que algumas classes podem ser testadas depois de um conjunto de aspectos, como é o caso da "G", testada depois do conjunto de aspectos {Af, Ab, Ad, Ac}. No caso específico do ORD na Figura 8, a classe "G" poderia ser implementada e testada antes dos aspectos "Af", "Ab", "Ad" e "Ac", porque tanto a classe quanto os aspectos dependem de um SCC não trivial e não possuem dependência entre si, conforme pode ser observado na Figura 13. No entanto, podem existir casos de classes que devem ser testadas depois dos aspectos, o que só acontece quando elas têm consciência dos aspectos.

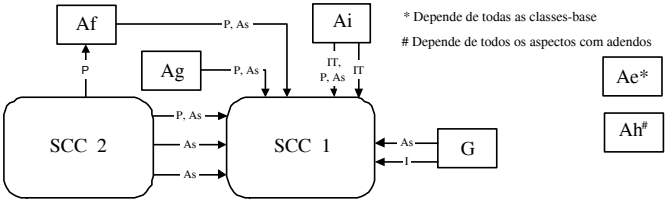

Figura 13. Exemplo de dependência das classes, aspectos e SCCs.

5. Conclusões e trabalhos futuros

Este artigo faz uma análise detalhada das associações entre classes e aspectos e mostra como a estratégia de Briand et al. [2003] pode ser aplicada diretamente a

programas orientados as aspectos com as associações entre classes e aspectos propostas. Analisaram-se dois casos de ordenação de uma aplicação fictícia com o objetivo de avaliar a estratégia incremental, freqüentemente citada na literatura para o teste de integração de programas orientados a aspectos. Como resultado, mostra-se que a estratégia incremental é melhor para quando a aplicação possui apenas aspectos transversais. Quando há aspectos bidirecionais, outras alternativas devem ser consideradas.

Outros estudos devem ser realizados para confirmar os resultados obtidos e, eventualmente, fornecer base para mudanças na aplicação da estratégia de teste de integração conjunta. Os estudos também serão úteis para investigar questões especificamente relacionadas à implementação de *stubs* de aspectos e ao teste de regressão em programas com aspectos que usam declarações inter-tipo. Adicionalmente, uma ferramenta deve ser desenvolvida para apoiar o teste de integração em programas orientados a aspectos.

Referências

Aldawud, O., Elrad, T., e Bader, A. (2003). A UML profile for aspect-oriented software development. In Aldawud, O., Kandé, M., Booch, G., Harrison, B., e Stein, D., editors, *Third International Workshop on Aspect Oriented Modeling*.

Alexander, R. T. e Bieman, J. M. (2005). Towards the systematic testing of aspect-oriented programs. In *Fourth International Conference on Aspect-Oriented Software Development (AOSD'2005) – Workshop On Testing Aspect Oriented Programs*, Chicago, Illinois – USA.

Baniassad, E. L. A. e Clarke, S. (2004). Theme: An approach for aspect-oriented analysis and design. In *ICSE*, pages 158–167. IEEE Computer Society.

Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, second edition.

Briand, L. C., Labiche, Y., e Wang, Y. (2003). An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607.

Ceccato, M., Tonella, P., e Ricca, F. (2005). Is aop code easier or harder to test than OOP code? In *Fourth International Conference on Aspect-Oriented Software Development (AOSD'2005) – Workshop On Testing Aspect Oriented Programs*, Chicago, Illinois – USA.

Colyer, A., Rashid, A., e Blair, G. (2004). On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University.

Douence, R., Fradet, P., e Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects. In Murphy, G. C. e Lieberherr, K. J., editors, *Fourth International Conference on Aspect-Oriented Software Development (AOSD'2005) – Workshop On Testing Aspect Oriented Programs*, pages 141–150. ACM.

Filman, R. E. e Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. In Tarr, P., Bergmans, L., Griss, M., e Ossher, H., editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*.

Gradecki, J. D., Lesiecki, N., e Gradecki, J. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., e Griswold, W. G. (2001). An overview of AspectJ. In Knudsen [2001], pages 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., e Irwin, J. (1997). Aspect-oriented programming. In *ECOOP*, pages 220–242.

Kienzle, J. e Guerraoui, R. (2002). Aop: Does it make sense? the case of concurrency and failures. In Magnusson, B., editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 37–61. Springer.

Kienzle, J., Yu, Y., e Xiong, J. (2003). On composition and reuse of aspects. In Leavens, G. T. e Clifton, C., editors, *FOAL: Foundations of Aspect-Oriented Languages*.

Knudsen, J. L., editor (2001). *ECOOP 2001 – Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18–22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*. Springer.

Kung, D. C., Gao, J., Hsia, P., Lin, J., e Toyoshima, Y. (1995). Class firewal, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Program*, 8(2):51–65.

Le Hanh, V., Akif, K., Traon, Y. L., e Jézéquel, J.-M. (2001). Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. In Knudsen [2001], pages 381–401.

Le Traon, Y., Jéron, T., Jézéquel, J., e Morel, P. (2000). Efficient OO integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25.

Milanova, A., Rountev, A., e Ryder, B. G. (2002). Constructing precise object relation diagrams. In *ICSM*, pages 586–595. IEEE Computer Society.

Mortensen, M. e Alexander, R. T. (2005). Adequate testing of aspect-oriented programs. In *Fourth International Conference on Aspect-Oriented Software Development (AOSD'2005) – Workshop On Testing Aspect Oriented Programs*, Chicago, Illinois – USA.

Rashid, A. e Chitchyan, R. (2003). Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, New York, NY, USA. ACM Press.

Stein, D., Hanenberg, S., e Unland, R. (2002). A UML-based aspect-oriented design notation for aspectj. In *AOSD*, pages 106–112.

Störzer, M. e Krinke, J. (2003). Interference analysis for Aspectj.

Tai, K.-C. e Daniels, F. J. (1999). Interclass test order for object-oriented software. *Journal of Object-Oriented Programming*, 12(4):18–25,35.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.

The AspectJ Team (2002). The AspectJ programming guide. Xerox Croporation.

Thompson, S. G. e Odgers, B. (1999). Aspect-oriented process engineering. In Moreira, A. M. D. e Demeyer, S., editors, *ECOOP Workshops*, volume 1743 of *Lecture Notes in Computer Science*, page 295. Springer.

Vanhaute, B., Win, B. D., e Decker, B. D. (2001). Building frameworks with aspectj. In *Workshop on Advanced Separation of Concerns (L. Bergmans, M. Glandrup. J. Brichau and S. Clarke, eds.)*, pages 1–6.

Zhou, Y., Richardson, D., e Ziv, H. (2004). Towards a practical approach to test aspect-oriented software. In *Testing Component-based Systems – TECOS 2004*, Erfurt – Germany.