Um catálogo de *stubs* para apoiar o teste de integração de programas orientados a aspectos

Reginaldo Ré[1], André L. dos S. Domingues[2], Paulo Cesar Masiero[2]

[1] Universidade Tecnológica Federal do Paraná – Campus Campo Mourão
Caixa Postal 271 – 87301-005, Campo Mourão, PR, Brazil
reginaldo@utfpr.edu.br

[2]Depto de Sistemas de Computação – ICMC/USP - São Carlos
Caixa Postal 668 – 13560-970, São Carlos, SP, Brazil
{alsd,masiero}@icmc.usp.br

Resumo. *Um dos problemas encontrados no teste de programas orientados a objetos é a ordem em que classes são integradas e testadas na presença de ciclos de dependência. Esse problema, também observado em programas orientados a aspectos, provoca aumento do esforço da atividade de teste pela necessidade de implementação de stubs para quebrar os ciclos de dependência. Este trabalho apresenta um conjunto de stubs e drivers coletados durante a condução de um estudo com quatro diferentes estratégias de teste de integração aplicadas a dois diferentes sistemas implementados com aspectos. Também são mostradas decisões de implementação tomadas em relação ao contexto em que os stubs e drivers são usados, além de diretrizes a serem seguidas durante a implementação desses artefatos de software para apoiar o teste de programas orientados a aspectos.*

Abstract. *A problem on the integration testing of object oriented programs is the order which classes are integrated and tested in the presence of dependency cycles. This problem, also observed in aspect oriented (AO) programs, causes additional effort during the testing activity, especially implementing stubs to break dependency cycles. This paper presents some stubs and drivers involving classes and aspects collected in a categorization study of four different testing strategies of two AO programs. The context which stubs and drivers were used and decisions about how to implement them according this context are also presented. Furthermore, guidelines to implement these software artifacts are proposed to support the testing of aspect-oriented programs.*

1. Introdução

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 (Kiczales et al., 1997; Gradecki et al., 2003). Não obstante essa nova abordagem possua diversas vantagens (Kiczales et al., 1997; Kienzle et al., 2003; Alexander and Bieman, 2004), ela também apresenta novos desafios para o desenvolvimento de software. No contexto de teste de programas OA, várias estratégias propõem o teste incremental dos programas que utilizam aspectos (Zhou et al., 2004; Ceccato et al., 2005). Nesta abordagem, 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. 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; Kienzle et al., 2003; Douence et al., 2004; The AspectJ Team, 2002; Kienzle and Guerraoui, 2002; Colyer et al., 2004; Filman and Friedman, 2005). 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.

Complementarmente aos trabalhos que visam ao desenvolvimento de técnicas e critérios de teste, o presente trabalho é um esforço que faz parte de um trabalho maior (Ré and Masiero, 2007) que tem por objetivo estabelecer diretrizes para integração de classes e aspectos e, por conseguinte, como utilizar técnicas e critérios de teste no contexto da estratégia de integração aplicada. Portanto, este trabalho é aderente aos demais trabalhos em que técnicas e critérios de teste são propostos para apoiar tanto o teste de unidade como o teste de integração. Enquanto o objetivo do trabalho proposto por Ré and Masiero (2007) foi avaliar o custo de implementação de *stubs* de acordo com a estratégia de teste de integração aplicada, o objetivo deste foi analisar diferentes situações de teste em que foi necessário a implementação de *stubs* e *drivers* para que a atividade de teste pudesse ser conduzida. Como resultado da análise dessas situações, foram coletados diferentes *stubs* e *drivers* de teste e diretrizes foram estabelecidas para guiar a implementação desses artefatos de software.

Não foram encontrados na literatura trabalhos que tratam a implementação de *stubs* na programação OA. Dentre alguns trabalhos interessantes, pode-se citar o de Binder (1999), que discute o uso de várias estratégias de integração e padrões de projeto para o teste de sistemas orientados a objeto. Da mesma forma que neste trabalho, o trabalho proposto por Binder (1999) mostra como e quando implementar *stubs* e *drivers*.

O restante do trabalho é organizado da seguinte maneira: na Seção 2 é apresentado resumidamente o estudo de caracterização das diferentes estratégias de ordenação. Na Seção 3 são mostrados os *stubs* e *drivers* catalogados e o contexto em que foram usados. Uma discussão sobre o estudo conduzido e os *stubs* e *drivers* catalogados é apresentada na Seção 4. Por fim, na Seção 5 são apresentadas as conclusões do trabalho.

2. Resumo do estudo de caracterização realizado

No estudo de caracterização conduzido por (Ré and Masiero, 2007) foram escolhidos dois sistemas implementados usando programação OA, mais especificamente AspectJ: um sistema que simula o controle de chamadas telefônicas chamado Telecom (The AspectJ Team, 2002) e um sistema de comércio desenvolvido por um dos autores, baseado no trabalho de Cibrán et al. (2003). Em ambos os sistemas foi adicionada a funcionalidade de persistência, utilizando-se um framework de persistência orientado a aspectos proposto por Camargo and Masiero (2005). O sistema Telecom utiliza aspectos para implementar requisitos funcionais e não funcionais, enquanto o sistema de comércio utiliza aspectos funcionais para implementar regras de negócio, segundo a proposta de Cibrán et al. (2003). O primeiro sistema é composto de 12 classes e aspectos e o segundo sistema de 18 classes e aspectos. As estratégias combinada, incremental+, reversa e randômica fo-

ram aplicadas nos dois sistemas, constituindo, então, dois estudos de diferentes sistemas para cada estratégia (Ré and Masiero, 2007).

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 (do inglês, Object Relation Diagram) para representar as dependências dos relacionamentos de herança, de agregação e de associação entre classes. O trabalho de Ré et al. (2007) adaptou a proposta de Kung et al. (1995) com novos tipos de arestas no ORD para representar as diferentes dependências presentes na programação OA. A estratégia apresentada nesse trabalho é baseada em dois conceitos principais: *cluster*, que é o conjunto com o máximo de vértices mutuamente alcançáveis no dígrafo (um ORD é um dígrafo); e, a quebra de ciclos, que é a remoção temporária de uma aresta com o objetivo de tornar o dígrafo acíclico. Em todas as ordens de teste de aplicação aplicadas, um ORD foi mapeado a partir dos aspectos e classes implementados. Em uma aplicação real de uma estratégia de ordenação o mapeamento deve ser feito com algum modelo estrutural. Dependendo de como foi feito o planejamento de teste de integração, PCDs podem ainda não ter sido implementados. Assim, a informação contida em artefatos de projeto, que podem ser chamados de interfaces de entrecorte (Krechetov et al., 2006), pode ser usado. Existem várias propostas na literatura com extensões da notação utilizada em diagramas de classes para representar aspectos, não obstante detalhes de implementação não conhecidos em tempo de projeto possam não ser conhecidos.

Na estratégia incremental+, primeiramente as classes, uma a uma, foram integradas e testadas e, posteriormente, os aspectos, um a um. A ordem em que os módulos foram integrados foi determinada pela aplicação do algoritmo de Briand em dois ORDs separados, um contendo apenas as classes e outro contendo apenas os aspectos. Na estratégia combinada, a ordenação das classes e aspectos foi determinada pela aplicação do algoritmo de Briand em um ORD conjunto, constituído por classes e aspectos. Na estratégia reversa, as classes foram integradas em ordem reversa à produzida pela estratégia incremental+. A estratégia randômica representa uma estratégia de integração ad-hoc. A Tabela 1 e a Tabela 2 apresentam a ordem em que classes e aspectos foram integrados com o auxilio das estratégias nos dois estudos. Os aspectos estão sublinhados e as classes e aspectos abstratos estão em itálico.

Tabela 1. Ordens de teste de classes e aspectos para sistema de Telecom.

	Combinada	Incremental+	Reversa	Randômica
1	*PersistentRoot*	*Connection*	TimerLog	Timer
2	*Connection*	Customer	Timing	*Connection*
3	*PersistentEntities*	LongDistance	Billing	Local
4	Customer	Local	Timer	Billing
5	LongDistance	Call	MyPersistentEntities	Customer
6	Local	Timer	Call	TimerLog
7	Call	*PersistentRoot*	Local	*PersistentEntities*
8	MyPersistentEntities	*PersistentEntities*	LongDistance	Timing
9	Timer	MyPersistentEntities	Customer	MyPersistentEntities
10	Billing	Billing	*PersistentEntities*	*PersistentRoot*
11	Timing	Timing	*Connection*	Call
12	TimerLog	TimerLog	*PersistentRoot*	LongDistance

Tabela 2. Ordens de teste de classes e aspectos para sistema de comércio

	Combinada	Incremental+	Reversa	Randômica
1	*PersistentRoot*	Product	ApplyFrequentDiscount	ApplyFrequentCustomer
2	*PersistentEntities*	ShoppingCart	ExtendCustomer	BRFrequentCustomer
3	Product	Store	MyPersistentEntities	ECheckout
4	ShoppingCart	Customer	ECheckout	ApplyChristmasDiscount
5	Store	*BRPriceDiscount*	ShoppingCart	ApplyFrequentDiscount
6	Customer	BRFrequentDiscount	BRFrequentCustomer	CaptureCustomer
7	MyPersistentEntities	BRFrequentCustomer	ApplyChristmasDiscount	BRFrequentDiscount
8	*BRPriceDiscount*	*PersistentRoot*	Product	ExtendCustomer
9	BRChristmasDiscount	BRChristmasDiscount	ApplyFrequentCustomer	EPricePersonalisation
10	EPricePersonalisation	*PersistentEntities*	*BRPriceDiscount*	BRChristmasDiscount
11	ExtendCustomer	MyPersistentEntities	*PersistentEntities*	*BRPriceDiscount*
12	BRFrequentDiscount	EPricePersonalisation	Customer	MyPersistentEntities
13	CaptureCustomer	ExtendCustomer	BRChristmasDiscount	Customer
14	ApplyFrequentDiscount	CaptureCustomer	*PersistentRoot*	Store
15	ApplyChristmasDiscount	ApplyFrequentDiscount	BRFrequentDiscount	ShoppingCart
16	ECheckout	ApplyChristmasDiscount	EPricePersonalisation	Product
17	BRFrequentCustomer	ECheckout	Store	*PersistentEntities*
18	ApplyFrequentCustomer	ApplyFrequentCustomer	CaptureCustomer	*PersistentRoot*

As quatro estratégias de integração foram executadas em um número de passos igual ao número de módulos a serem testados, 12 passos para o sistema de Telecom e 18 passos para o sistema de comércio. Em cada passo foram realizadas as seguintes tarefas:

- a lista de ordenação produzida pela estratégia indicou o módulo a ser testado, que foi adicionado à configuração de teste;
- classes e aspectos já testados e integrados em passos anteriores foram adicionados à configuração de teste;
- *stubs* e *drivers* foram implementados e adicionados à configuração de teste;
- novos casos de testes foram implementados e adicionados juntamente com os casos antigos à configuração de teste;
- os testes foram executados e os resultados analisados;
- os *stubs* e *drivers* foram analisados e coletados.

Embora não exista consenso entre pesquisadores da área sobre qual elemento de uma linguagem orientada a objetos deve ser considerado uma unidade de teste (Binder, 1999; McGregor and Sykes, 2001), as estratégias de teste utilizadas no trabalho em que os *stubs* e *drivers* foram implementados consideram que as fases de teste são delimitadas pela existência de diferentes níveis de teste: intra-método/adendo, intra-classe/aspecto, inter-classes/aspectos e de sistema. Essa categorização em diferentes níveis de teste foi adotada porque o foco do trabalho são nas estratégias que podem ser utilizadas para a integração de classes e aspectos e não em técnicas e critérios de teste.

As atividades de teste conduzidas no estudo se concentraram na integração inter-classe/aspecto. Um método ou adendo em teste é chamado de unidade de teste e uma classe ou aspecto em teste é chamado de módulo em teste. Assim sendo, o teste de unidade é conduzido quando uma unidade em teste não utiliza métodos ou atributos de outras módulos, e a atividade de teste concentra-se apenas no módulo em teste. O teste de integração acontece quando uma unidade em teste utiliza métodos ou atributos de outras classes/aspectos, em que a atividade de teste se concentra na unidade em teste e também nas interfaces entre os componentes, isto é, no uso de atributos e métodos de outras classes. Neste caso, quando o objeto referenciado é uma instância de uma classe já implementada, não é necessário a implementação de um *stub*, evitando o trabalho de codificação

de um *stub* desnecessário, uma vez que a classe já foi testada. Embora essa abordagem tenha sido utilizada, outros trabalhos discutem a necessidade de se implementar *stubs*, mesmo que já exista uma implementação real, para que a procura por defeitos seja restrita à unidade em teste.

A Tabela 3 apresenta uma descrição dos casos de implementação de *stubs* e *drivers* encontrados no estudo.

Tabela 3. Resumo dos casos de implementação de *stubs* e *drivers*

Caso	Descrição
Classes e Aspectos abstratos	*Stubs* são necessários em duas situações distintas: quando é necessário simular o comportamento de uma classe abstrata alvo de introduções de um aspecto; e, quando o aspecto em teste é abstrato e é necessário simular o comportamento de uma concretização desse aspecto.
Transferencia de dependência	*Drivers* de teste podem ser implementados para simular pontos de junção capturados por definições de 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.
Dependência transitiva	Quando um módulo em teste, seja ele aspecto ou classe, 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.
Stub de aspecto para testar adendos e de classes para testar introduções	Algumas vezes, adendos de um aspecto usam PCDS de outros aspectos para capturar pontos de junção. Nesses casos, a implementação de *stubs* de aspectos com conjuntos de junção que capturam o contexto pode ser necessária. Quando existe uma introdução de método pode ser necessário implementar um *stub* que simulam o contexto em que o método introduzido deve ser testado, uma vez que ele pode usar atributos e outros métodos da classe.
Stubs de aspectos para testar PCDs	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* de aspectos e de classes para que a verificação da captura dos conjuntos de junção possa ser feita.

3. Catálogo de *stubs*

3.1. Classes e aspectos abstratos

A presença de classes e aspectos abstratos trouxe dificuldades na execução dos testes, além de fazer com que *stubs* fossem criados para concretizar o módulo abstrato em teste. Quando o módulo em teste é uma classe abstrata, um *stub* é necessário para concretizar o módulo abstrato. Além disso, existe o esforço de manter o *stub* em configurações de teste relativos a outros módulos até que uma especialização concreta seja implementada. No teste do sistema Telecom, a classe `Connection` é uma classe abstrata, e a solução adotada foi torná-la uma classe concreta durante os testes. Isso foi possível porque não existem métodos abstratos nessa classe. Constatou-se posteriormente ao estudo que essa solução não foi a mais adequada porque é invasisa e suscetível a introdução de defeitos.

A existência de aspectos abstratos influencia a implementação de *stubs* no teste de integração da seguinte maneira:

- quando o aspecto em teste é concreto e faz introduções de métodos concretos e abstratos que se relacionam:

  - Deve-se criar um *stub* concreto para cada alvo abstrato das introduções, desde que já não existam especializações concretas dos alvos das introduções presentes na configuração de teste;
- quando o aspecto em teste é abstrato:
  - Deve-se concretizar métodos e PCDs abstratos por meio de um *stub* de aspecto, desde que já não existam especializações concretas do aspecto em teste presentes na configuração de teste.
- quando o aspecto faz alterações em hierarquias de herança:
  - Este caso é discutido na Seção 4.

Note-se que um mesmo aspecto pode apresentar combinações dessas características, por exemplo, ser abstrato e fazer introduções de métodos concretos e abstratos que se relacionam. Outro ponto importante a ser observado é que para testar corretamente alterações em hierarquia de herança, do ponto de vista de teste de integração, é necessário testar todas as classes e aspectos a partir do ponto de alteração na hierarquia e todos os clientes dessas classes e aspectos.

Verificou-se que são duas as origens de *stubs* nessas condições: quando o módulo em teste é um aspecto abstrato; e quando o módulo em teste faz introduções de métodos concretos e métodos abstratos que se relacionam. Quando o aspecto em teste é abstrato, pode-se utilizar soluções análogas àquelas indicadas para o teste de classes abstratas, como por exemplo, criar *stubs* concretos. A existência de PCDs abstratos e adendos que utilizam esses PCDs deve ser considerada na implementação da solução. No segundo caso, *stubs* devem ser implementados para cada classe/aspecto alvo de introduções. Isso deve ser feito caso não haja especializações concretas da classe/aspecto alvo da introdução na configuração de teste, que depende da estratégia de integração usada.

Um exemplo interessante de introdução de métodos concretos e abstratos que se relacionam pode ser observado na Figura 1. Nela são ilustrados alguns aspectos e classes que fazem parte do sistema de Telecom. `PersistentEntities` é um aspecto abstrato que introduz vários métodos concretos e abstratos na classe `PersistentRoot`. `MyPersistentEntities` é a especialização concreta de `PersistentEntities` que declara que a classe `Customer` é uma especialização da classe `PersitentRoot`. Na estratégia combinada, a ordem de integração desse conjunto de classes e aspectos é: `PersistenRoot`, `Customer`, `PersistentEntities` e `MyPersistentEntities`. No momento do teste de `PersistentEntities` deveria ser criado um *stub* apenas se uma especialização concreta de `PersistentRoot` não existir. Contudo, apesar de `Customer` já estar na configuração de teste, a relação de herança entre ela e `PersistentRoot`, denotado na Figura 1 por um relacionamento de herança com linha tracejada, não existe, porque `MyPersistentEntities` ainda não foi integrado e testado. No estudo, a solução utilizada foi, ao invés de criar um *stub* concretizando `PersistentRoot`, criou-se um *stub* para simular `MyPersistentEntities`, com a declaração intertipos que determina que `PersistentRoot` é uma generalização de `Customer`. O *stub* criado pode ser visto na Listagem 1. Observa-se que a implementação de um *stub* de aspectos contendo apenas alterações na hierarquia de herança é menos custosa que implementar uma classe com métodos e atributos. Na Listagem 2 é apresentado parte do *driver* JUnit de teste, em que pode ser observado o teste da operação de persistencia `save()`.

Listagem 1. *Stub* para simular o aspecto `MyPersistentEntities`.

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

Listagem 2. Parte do *driver* de `PersistentEntitiesTestCases`.

```
1  public class PersistentEntitiesTestCase extends TestCase {
2     public void testSaveAndSetDBToObjectCustomer(){
3          Customer c1 = new Customer(1, "João", 18);
4          c1.save();
5          Customer c2 = new Customer();
6          c2.setID(1);
7          c2 = (Customer) c2.setDBToObject();
8          assertEquals(c1.getName(), c2.getName());
9          assertEquals(c1.getId()), c2.getId();
10         assertEquals(c1.getAreacode(), c2.getAreacode());
11         c2.delete();
12     }
13  }
```

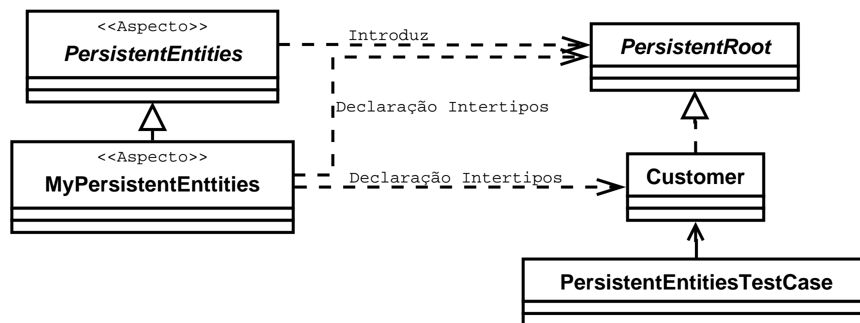

Figura 1. Classes e aspectos necessários ao teste de `PersistentEntities`.

Uma estratégia de integração que posterga o teste de módulos abstratos até que suas especializações concretas façam parte da configuração de teste substitui a implementação de *stubs* como o descrito, porém, o teste de módulos-cliente também deve ser postergado. Além disso, em grandes hierarquias de herança uma estratégia desse tipo pode postergar em demasia o teste de módulos abstratos e seus clientes.

3.2. Transferência de dependência

Durante o teste do aspecto `TimerLog` do sistema Telecom usando a estratégia reversa, cuja ordem de teste é `TimerLog`, `Timer` e `Timing`, é necessário implementar um *stub* de aspecto para simular o comportamento de `Timing`. `TimerLog` é um aspecto que depende do aspecto `Timing` e da classe `Timer`, conforme pode ser observado na Listagem 3 e na Listagem 4. `TimerLog` entrecorta adendos de `Timing`, linhas 5 e 10 da Listagem 4, para registrar o início e o término de cada chamada telefônica.

Listagem 3. 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 4. 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.getCallerO();
9      c1.totalConnectTime += getTimer(c).getTime();
10     c2 = c.getReceiverO();
11     c2.totalConnectTime += getTimer(c).getTime();
12     c1.save();
13     c2.save();
14   }
15   public long Customer.totalConnectTime = 0;
16   public long getTotalConnectTime(Customer cust) {
17     return cust.totalConnectTime;
18   }
19   public void Customer.setTotalConnectTime(Long vlr){
20     this.totalConnectTime=vlr;
21   }
22   public Long Customer.getTotalConnectTime() {
23     return this.totalConnectTime;
24   }
25   public Timer Connection.timer = new Timer();
26   public Timer getTimer(Connection conn) {
27     return conn.timer;
28   }
29 }
```

Para efetuar o teste são necessários os seguintes artefatos: um *stub* de `Timing` com dois adendos e dois PCDs; um *stub* que simula o contexto para os adendos de `Timing` entrecortarem, chamado na Figura 2 de `StubJPTiming`; e um *driver* JUnit que manipula `StubJPTiming` para cada caso de teste.

Vários são os problemas com essa abordagem: a criação de um *stub* apenas para simular o ponto de junção, `StubJPTiming`; o aumento da complexidade da implementação de `Timing`, com a implementação de PCDs; 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 durante o teste de integração. 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.

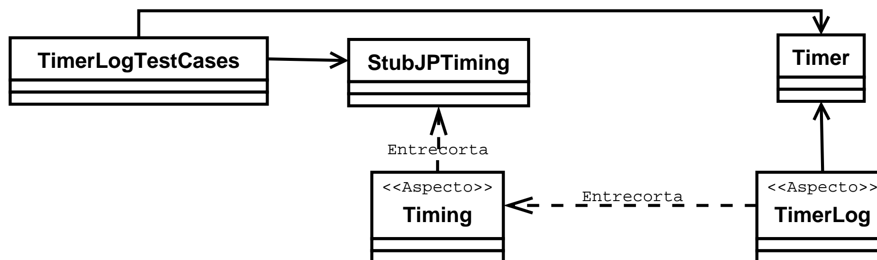

Figura 2. Classes e aspectos necessários ao teste de `Timing`.

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 o *driver* JUnit no momento do teste. De maneira mais geral, seja O um objeto que possui um ponto de junção J definido por um conjunto de junção P de um aspecto A, se O não é utilizado por P nem por qualquer adendo de A sensibilizado por intermédio de P, então, no momento do teste de A, a dependência entre A e O é transferida para o *driver* de teste.

Note-se, pela Listagem 3, que todos os conjuntos de junção são do tipo `call`. Nos conjuntos de junção, um objeto da classe `Timer` é capturado pelo comando `target` e passado por parâmetro para os adendos de `TimerLog`. Apesar do ponto de junção

ocorrer originalmente no aspecto Timing, a dependência entre TimerLog e Timing deixa de existir no momento do teste porque o objeto em que ocorre o ponto de junção não é utilizado pelos adendos de TimerLog. Isso acontece porque a dependência fica estabelecida entre o módulo em teste TimerLog e o *driver* de teste, pois é exatamente no *driver* de teste que existem as chamadas aos métodos que são capturados pelo conjunto de junção de TimerLog e a simulação do contexto de execução dos adendos de TimerLog, conforme pode ser observado na Figura 3.

Listagem 5. Parte do *driver* de TimerLogTestCases.

```
1  public class TimerLogTestCase extends TestCase {
2    Timer t;
3          PrintStream oldps;
4          BufferedReader br;
5
6    public void testCallToStart(){
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 Listagem 5 é mostrado parte do *driver* de teste de TimerLogTestCases, em que pode-se observar na linha 19 o ponto de junção capturado. Com a transferência da dependência do ponto de junção para o *driver*, não é necessário implementar o *stub* que deveria simular o comportamento do aspecto Timing. A regra é válida para outros tipos de conjunto de junção além de call, porém a ocorrência é bem menos freqüente.

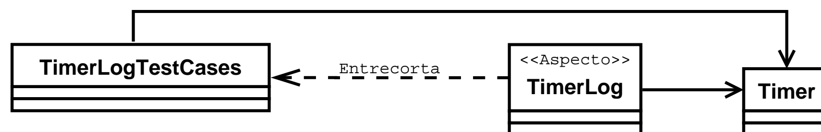

Figura 3. Classes e aspectos necessários ao teste de Timing usando a transferência de dependência.

As maneiras de averiguar se a dependência entre o aspecto em teste e os pontos de junção pode ser transferida são: quando o objeto em que ocorre o ponto de junção não é passado por parâmetro a um adendo; quando não existir em um adendo uma referência ao ponto de junção por reflexão; e quando um adendo de contorno não faz chamada ao método original capturado pelo conjunto de junção. É importante salientar que, apesar desse exemplo ter ocorrido na aplicação da estratégia reversa, ele também pode ocorrer em virtude de uma quebra de ciclos de dependência.

3.3. Dependência transitiva

No diagrama da Figura 4 são mostradas as classes e os aspectos importantes para o teste de Call do sistema Telecom. No diagrama, pode-se observar que o aspecto chamado MyPersistentEntities é uma concretização do aspecto PersistentEntities, que introduz métodos e atributos em PersistentRoot. Também é possível observar que MyPersistentEntities altera a hierarquia de herança fazendo com que Local seja uma especialização de PersistentRoot e, portanto, faz com que métodos e atributos de PersistentRoot e de

PersistEntities sejam herdados por Local. O relacionamento de herança é mostrado no diagrama por uma seta tracejada entre Connection e PersistentRoot para enfatizar que essa dependência só existe na presença do aspecto MyPersistentEntties.

No teste da classe Call usando a estratégia reversa, a ordem de teste dos módulos é: Call, Local, PersistentEntities, Connection e PersistentRoot. No momento do teste de Call, o aspecto MyPersistentEntities já está na configuração de teste. São necessários *stubs* de Local e PersistentEntities, porque estão diretamente ligados a Call.

Na dependência transitiva, *stubs* deixam de ser implementados porque ao invés de se implementar introduções feitas em classes/aspectos alvos, implementa-se diretamente os métodos e atributos nessas classes/aspectos alvos. Como exemplo, pode-se observar que a classe Call, além de utilizar alguns métodos de Local, utiliza métodos que PersistentEntities introduz em PersistentRoot e que são herdados por Local. Ao invés de implementar PersistentEntities e os métodos que ela introduz, pode-se implementar diretamente em Local esses métodos, conforme pode ser observado Figura 5. De maneira mais geral, seja M o conjunto de classes e aspectos, $m_1, m_2, m_3 \in M$ e $m1$ é o módulo em teste. Seja, também, m_1 um módulo que depende de m_2 e m_3, e que m_2 possui um conjunto de atributos Att e/ou um conjunto de métodos Met usados por m_1 que ou são introduzidos por m_3 ou por uma alteração de hierarquia de herança feita por um aspecto qualquer que faça com que m_3 seja uma generalização de m_2. Dado que s_2 e s_3 são *stubs* que simulam m_2 e m_3, respectivamente, então pode-se fazer com que s_2 simule diretamente os atributos e métodos em Att e Met, e não seja necessária a implementação de s_3.

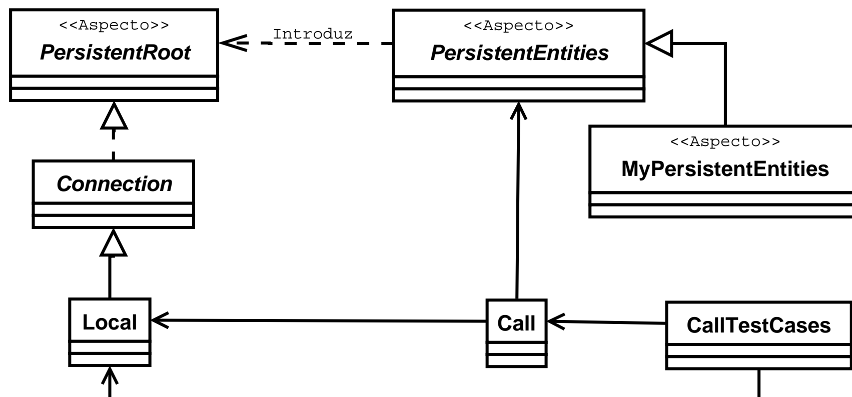

Figura 4. Classes e aspectos necessários ao teste de Call.

Assim, não é necessário implementar um módulo que simula PersistentEntities porque seus métodos e atributos foram simulados diretamente no *stub* correspondente Local. Apesar do número de módulos ser menor, o número de métodos e atributos simulados não é alterado. Pode-se inferir desse fato que existe uma diminuição do esforço de implementação de *stubs*, mas que essa diminuição é muito pequena, 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 na declaração das introduções.

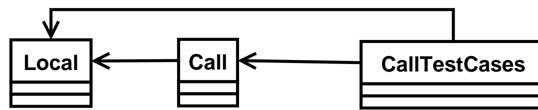

Figura 5. Classes e aspectos necessários ao teste de `Call` usando dependência transitiva.

3.4. Stub de aspectos para testar adendos e de classes para testar introduções

Para testar o aspecto `Timing` do sistema de Telecom na ordem inversa, mostrado na Listagem 4, foi necessário implementar um *stub* de `Billing`, mostrado na Listagem 7. A dependência é causada pelo uso que `Timing` faz de dois PCDs definidos em `Billing`. Além dessa dependência, `Timing` também depende de `Call`, `Timer`, `Customer` e `Connection`, como pode ser observado na Figura 6. No teste de `Timing` a ordem de teste na estratégia reversa é: `Timing`, `Billing`, `Timer`, `Call`, `Customer` e `Connection`.

Um dos pontos interessantes no teste de `Timing` é a existência de um *stub* de aspecto que implementa PCDs para testar outro aspecto. Conforme discutido na Seção 3.2, do ponto de vista do teste de adendos, é necessário simular o contexto onde o adendo vai atuar e não necessariamente todo o ponto de junção. Deve-se ressaltar que a implementação dos dois PCDs no *stub* de aspecto é idêntica à implementação real. Um problema que surge da simulação de PCDs é que caso seja mal projetado ou implementado, o PCD pode capturar pontos de junção indesejados que afetam a execução do módulo em teste. Além disso, o problema continua nas próximas etapas do teste, uma vez que pode ser necessário manter o *stub* com PCDs durante o teste de módulos integrados subseqüentemente.

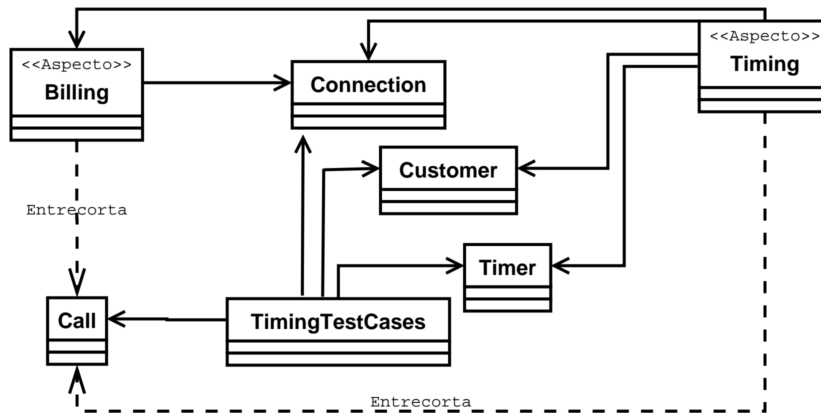

Figura 6. Classes e aspectos necessários ao teste de `Timing`.

Além da simulação dos PCDs, foram implementados conjuntos de junção que esses PCDs capturam para criar o contexto de execução dos adendos de `Timing`. Para isso, os métodos `complete` e `drop` foram declarados em um *stub* da classe `Connection`, porém sem instruções no corpo de cada um dos métodos, e invocados no *driver* de teste, conforme pode ser visto nas linhas 13 e 14 da Listagem 6.

Listagem 6. Parte do *driver* de `TimigTestCases`.

```
1 public class TimingTestCase extends TestCase {
2   Customer cust;
3   Connection conn;
4   public void testGetTotalConnectTime(){
5     long total;
6               cust.setTotalConnectTime(1);
7     total=Timing.aspectOf().getTotalConnectTime(cust);
8     assertEquals(total, 1)
```

```
9   }
10  public void testAfterConnectionDrop(){
11    long total;
12    conn.complete();
13    conn.drop();
14    total=Timing.aspectOf().getTimer(conn).getTime();
15    assertEquals(total, 1);
16  }
17 }
```

Outro ponto interessante é a necessidade de criar um *stub* da classe `Customer` apenas para testar as introduções que `Timing` faz em `Customer` e `Connection`. O teste das introduções de `Timing` foi trivial por que os métodos introduzidos não interagem com outros métodos da classe. Em exemplo pode ser visto nas linhas 4 a 9 da Listagem 6, em que é mostrada a chamada ao método `setTotalConnectTime()` introduzido na classe `Customer`. No entanto, algumas vezes os *stubs* apenas com as declarações das classes devem ser uma especialização de alguma classe ou aspecto já presente na configuração de teste, que permite testar introduções de métodos que interagem com generalizações. *Stubs* com métodos sem corpo para simular pontos de junção e classes apenas para que introduções fossem feitas foram bastante utilizados durante a condução do estudo de caracterização.

Listagem 7. *Stub* para simular o aspecto `Billing`.

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

3.5. Stubs de aspectos para testar PCDs

Na Figura 7 é apresentado um diagrama de classes com classes e aspectos necessários ao teste de `ECheckout` do sistema de comércio. O aparecimento desse tipo de *stub* ocorreu na estratégia randômica e também na estratégia reversa, cuja ordem de teste é: `ECheckout`, `Customer` e `Store`. O sistema de comércio implementa regras de negócio segundo uma arquitetura proposta por Cibrán et al. (2003) em que aspectos são responsáveis por capturar e aplicar regras de negócio implementadas em classes. Os aspectos responsáveis por capturar pontos de junção são constituídos apenas de PCDs, conforme pode ser observado na Listagem 8, enquanto que os aspectos responsáveis por aplicar as regras de negócio possuem adendos e fazem introduções nas classes base. Portanto, foi frequente a necessidade de testar aspectos que não possuem adendos, uma situação diferente do teste do sistema Telecom. O problema é testar código fonte que não é executável que, do ponto de vista de teste de integração, é constituído de duas etapas: testar se são capturados os pontos de junção corretos e se os objetos de contexto são capturados corretamente.

Listagem 8. Aspecto `ECheckout`.

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

Do ponto de vista do teste de integração, a tarefa mais problemática é testar se são capturados os pontos de junção corretos. Podem ser necessários vários *stubs* para simular os pontos junção a serem capturados, afim de verificar se eles são sensibilizados, além

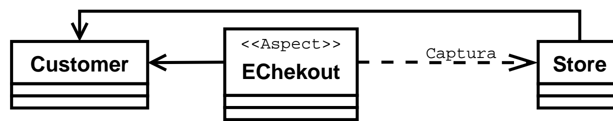

Figura 7. Classes e aspectos necessários ao teste de `ECheckout`

de um *stub* para simular o adendo que usa o PCD e o *driver* de teste JUnit. Contudo, essa solução funciona apenas para verificar se pontos de junção desnecessários foram capturados, mas não funciona para averiguar se pontos de junção que deveriam ter sido capturados foram efetivamente capturados.

Os *stubs* criados para testar se um PCD captura os pontos de junção corretos são classes que possuem métodos sem corpo, ou seja, sem implementação, que são interceptados pelo PCD. É necessário, também, implementar *drivers* que são aspectos constituídos de adendos que usam os PCDs a serem testados, conforme ilustrado na Listagem 10. Como pode ser visto na Figura 8, o *driver* de `AspectToTestECheckout` é responsável por usar o PCD de `ECheckout` para entrecortar o *stub* de `Store` e verificar se os objetos de contexto, no caso um objeto *stub* da classe `Customer`, são capturados corretamente. Note-se que o *driver* de `AspectToTestECheckout` deve ser uma especialização da classe `TestCases` para ser compatível com o JUnit e possuir código de teste. O *driver* JUnit de `ECheckoutTestCases` também é necessário para invocar o método que será interceptado, mostrado na linha 7 da Listagem 9.

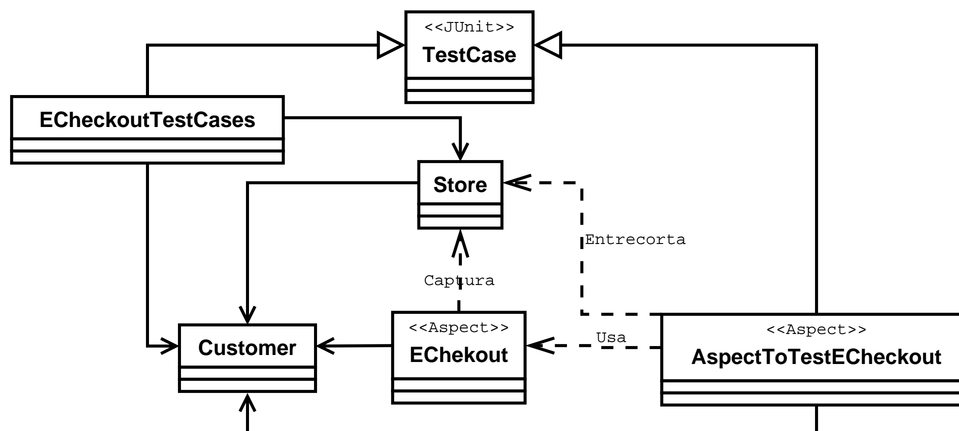

Figura 8. Classes, aspectos, *drivers* e *stubs* necessários ao teste de `ECheckout`.

Usando outras estratégias, o teste de PCDs também foi anotado em um contexto um pouco diferente. Generalizando o caso, existem duas outras possbilidades em que o teste de PCDs pode ocorrer, além desta, em que *stubs* são necessários para simular os pontos de junção e *drivers* são necessários para simular o adendo que usa o PCD: quando um aspecto com adendos que usam o PCD em teste já foi integrado e faz parte da configuração de teste, fazendo com que não seja necessário implementar um *stub*; e quando classes e aspectos que são interceptados pelo PCD já estão testados e integrados à configuração de teste, fazendo com que não seja necessária a implementação dos *stubs* que simulam os pontos de junção.

Listagem 9. *Driver* de `ECheckoutTestCases`

```
1 public class ECheckoutTestCases extends TestCase {
2   public void testCheckoutPCD(){
3     Store store = new Store();
4     Customer cust = new Customer();
5     cust.id=1;
```

```
6
7     store.checkout(cust);
8     assertTrue(cust.id==0);
9   }
10 }
```

Listagem 10. *Stub* de `AspectToTestECheckout`

```
1 public aspect AspectToTestECheckout extends TestCase{
2   after(Customer aCustomer): ECheckout.checkout(aCustomer){
3     assertTrue(aCustomer.id==1);
4     aCustomer.id=0;
5   }
6 }
```

4. Discussão dos resultados

Alguns *stubs* implementados durante a condução dos estudos apresentam caracteristicas interessantes que merecem ser discutidas. Aspectos que possuem declarações intertipos que fazem apenas alterações de hierarquia de herança ou declarações de precedência de aspectos foram encontrados. A solução utilizada para testar alterações de hierarquia de herança foi implementar *stubs* apenas com a declaração das classes, sem métodos ou atributos, e o uso do operador `instanceof`, que verifica se uma instância é de uma determinada classe ou se pode sofrer uma coerção para se tornar uma instância de tal classe. A peculiaridade é que esses são casos de teste que têm por objetivo investigar se um relacionamento é correto, especificamente o de herança. Casos de teste com esse objetivo não são encontrados no teste de programas orientados a objetos.

O teste de métodos de aspectos é feito de maneira similar o teste feito para métodos de classes. Um *driver* deve simular o contexto de execução do aspecto para que seus métodos possam ser testados. Nas linhas 5 a 9 da Listagem 6 é apresentado o teste de um dos métodos do aspecto do tipo *singletron* `Timing`.

Grande parte das situações de teste em que os *stubs* e *drivers* foram catálogados foram coletadas quando a estratégia reversa foi aplicada. Nessa estratégia, o desenvolvimento de *stubs* foi bem maior que nas estratégias combinada e incremental+, e juntamente com a manutenção da configuração de teste, foi bastante custosa. Com o aumento de *stubs*, a probabilidade de que os *stubs* sejam reimplementados para cada caso de teste, quando existem casos de teste complexos, é maior, tornando dificil o manter o controle e a consistência. A estratégia reversa é variação da estratégia de teste top-down, em que o objetivo é demonstrar estabilidade integrando módulos em uma ordem de hierarquia de controle. Isso porque, segundo Binder (1999), os módulos que são integrados e testados primeiramente são aqueles que controlam (ou dependem) os outros módulos em uma árvore de hierarquia. Essa característica faz com que seja possível o desenvolvimento concorrente dos módulos usando variantes do processo incremental e, também, no desenvolvimento de frameworks.

A estratégia combinada e a incremental+ são variações da integração bottom-up, que é usada para demostrar estabilidade do sistema em teste ao direcionar a adição gradativamente de módulos a serem testados, começando por aqueles módulos com menor número de dependências. Nessa estratégia, a implementação de *drivers* para simular o estado do sistema no momento da execução do teste é comum. No estudo, os *drivers* são representados pelos casos de teste implementados usando JUnit. O esforço de desenvolvimento desses *drivers* adiciona mais custo à tão custosa atividade de teste. Esse custo

é minimizado nas estratégias exatamente porque JUnit é a ferramenta utilizada para o teste e, portanto, os *drivers* seriam implementados de qualquer maneira, embora, provavelmente, com menor complexidade.

5. Conclusão e trabalhos futuros

A implementação de *stubs* e *drivers* na atividade de teste de integração de programas orientados a objetos baseia-se no tipo de relacionamento existente entre as classes que compõem o sistema em teste: herança, agregação e associação. Basicamente, deve-se simular por meio do *stub* o comportamento de uma classe pela criação de métodos e atributos que a classe em teste utiliza, enquanto que os *drivers* simulam o estado do sistema para fornecer o contexto com o qual os métodos da classe em teste são executados. Na programação orientada a aspectos, o papel dos *stubs* e *drivers* é semelhante, no entanto, sua implementação baseia-se nos novos tipos de relacionamento caracteríticos da programação orientada a aspectos: entrecorte, uso de PCDs e introduções.

Os *stubs* e *drivers* coletados durante o estudo possuem em comum a simulação desses novos tipos de relacionamentos. Enquanto na programação orientada a objetos os *stubs* e *drivers* devem fornecer suporte ao teste por meio de seqüência de chamada de métodos, na programação orientada a aspectos eles devem fornecer suporte ao teste de: introdução de novos métodos e o contexto onde ocorrem as introduções; entrecorte de métodos; captura de pontos de junção por PCDs. Muitas vezes não é trivial definir quais e como devem ser implementados *stubs* para simular, por exemplo, o contexto de um ponto de junção ou um PCD para capturar corretamente o ponto de junção necessário ao teste de um adendo. A principal contribuição deste trabalho é apóiar a atividade de teste de integração fornecendo diretrizes para implementar os *stubs* e *drivers* de acordo com a configuração de teste existente no momento do teste de cada módulo e estratégia de integração utilizada. Além disso, os *stubs* e *drivers* apresentados visam a minimização do esforço na implementação dos *stubs* e *drivers* bem como para a manutenção da configuração de teste ao longo da atividade de teste.

Como os *stubs* e *drivers* foram coletados apenas estudando dois sistemas, novos estudos com outros sistemas implementados com aspectos estão sendo conduzidos para refinar os casos de *stubs* e *drivers* existentes e, eventualmente, coletar novos casos. Além disso, pretende-se criar padrões de *stubs* e *drivers* para o teste de integração de programas orientados a aspectos.

Referências

Alexander, R. T. and Bieman, J. M. (2004). Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Colorado State University, Fort Collins, Colorado.

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

Camargo, V. V. and Masiero, P. C. (2005). Object-oriented frameworks. In *XIX SBES - Brazilian Symposium on Software Engineering*, pages 200–216, Uberlândia – Brazil. in Portuguese.

Ceccato, M., Tonella, P., and Ricca, F. (2005). Is AOP code easier or harder to test than OOP code? In *First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005)*, Chicago, Illinois – USA.

Cibrán, M., D'Hondt, M., and Jonckers, V. (2003). Aspect-oriented programming for connecting business rules. In *6th International Conference on Business Information Systems*.

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

Douence, R., Fradet, P., and Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects. In Murphy, G. C. and 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. and Friedman, D. P. (2005). Aspect-oriented programming is quantification and obliviousness. In Filman, R. E., Elrad, T., Clarke, S., and Akşit, M., editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Aksit, M. and Matsuoka, S., editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer.

Kienzle, J. and Guerraoui, R. (2002). AOP: Does it make sense? The case of concurrency and failures. In Magnusson, B., editor, *ECOOP 2002, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *LNCS*, pages 37–61. Springer.

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

Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., and Kulesza, U. (2006). Towards an integrated aspect-oriented modeling approach for software architecture design. In *8th Workshop on Aspect-oriented Modeling (AOM'06) at the Fifth International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany.

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

McGregor, J. D. and Sykes, D. A. (2001). *A practical guide to testing object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ré, R., Lemos, O. A. L., and Masiero, P. C. (2007). Minimizing stub creation during integration test of aspect-oriented programs. In *WTAOP '07: Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 1–6, New York, NY, USA. ACM Press.

Ré, R. and Masiero, P. C. (2007). Integration testing of aspect-oriented programs: a characterization study to evaluate how to minimize the number of stubs. In *XXI SBES - Brazilian Symposium on Software Engineering*, volume 21, pages 411–426, João Pessoa – Brazil.

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

Zhou, Y., Richardson, D., and Ziv, H. (2004). Towards a practical approach to test aspect-oriented software. In Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., and Schweiggert, F., editors, *Testing of Component-Based Systems and Software Quality, TECOS 2004 (Workshop Testing Component-Based Systems)*, pages 1–16, Erfurt – Germany.