Leonardo Humberto Guimarães Silva

Definição de Conjuntos de Junção Robustos usando Aspect-Aware Interfaces e Aspectos Anotadores

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Junho 2008

FICHA CATALOGRÁFICA

Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Silva, Leonardo Humberto Guimarães
S586d Definição de conjuntos de junção robustos usando Aspect-Aware Interfaces e Aspectos Anotadores / Leonardo Humberto Guimarães Silva. – Belo Horizonte, 2008.
74 f. : il.

Orientador: Prof. Dr. Marco Túlio de Oliveira Valente.
Dissertação (mestrado) – Pontifícia Universidade Católica de Minas Gerais, Programa de Pós-graduação em Informática.
Bibliografia.

1. Engenharia de software – Teses. 2. Programação (Computadores) 3. Linguagem orientada a objetos I. Valente, Marco Túlio de Oliveira. II. Pontifícia Universidade Católica de Minas, Programa de Pós-graduação em Informática. III. Título

CDU: 681.3.066

Bibliotecário: Fernando A. Dias – CRB6/1084

PUC Minas
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

" Definição de Conjuntos de Junção Robustos usando Aspect-Aware Interfaces e Aspectos Anotadores"

Leonardo Humberto Guimarães Silva

Dissertação defendida e aprovada pela seguinte banca examinadora:

Prof. Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Prof. Roberto da Silva Bigonha (UFMG)
Prof. Vladimir Oliveira Di Iorio (UFV)
Prof. Mark Alan Junho Song (PUC Minas)

Prof. Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Doutor em Ciência da Computação - UFMG

Prof. Roberto da Silva Bigonha (UFMG)
Doutor em Ciência da Computação – University of California, Los Angeles

Prof. Vladimir Oliveira Di Iorio (UFV)
Doutor em Ciência da Computação - UFMG

Prof. Mark Alan Junho Song (PUC Minas)
Doutor em Ciência da Computação - UFMG

Belo Horizonte, 16 de junho de 2008.

Anel Rodoviário – KM 23,5, Rua Walter Ianni, 255 – São Gabriel / Belo Horizonte – MG / CEP: 31980-110
Fone: (31) 3439-5204 – Fax: (31) 3439-5204 – e-mail: mestradoinfsec@pucminas.br

Resumo

Aspectos foram propostos para incrementar os níveis de reúso, facilitar o entendimento e diminuir o esforço demandado por manutenções em sistemas orientados por objetos. No entanto, aspectos também podem comprometer a implementação e evolução em separado de módulos de um sistema. O principal motivo é que conjuntos de junção utilizam nomes e expressões regulares para definir pontos de junção de um programa alvo. Assim, o modelo de definição de conjuntos de junção de linguagens como AspectJ acaba por produzir um forte acoplamento entre conjuntos de junção e a estrutura do programa alvo. Este problema já foi investigado por outros pesquisadores, sendo usualmente chamado de problema dos conjuntos de junção frágeis.

Anotações são freqüentemente lembradas como uma possível alternativa para reduzir a fragilidade de conjuntos de junção em AspectJ. Entretanto, anotações são consideradas como interesses transversais porque elas ficam normalmente espalhadas e misturadas a regras de negócio específicas da aplicação. Assim, nesta dissertação, apresenta-se uma proposta para atenuar o problema dos conjuntos de junção frágeis de AspectJ baseada na combinação de dois componentes principais: *aspect-aware interfaces* (AAIs) e aspectos anotadores. *Aspect-aware interfaces* possuem informações sobre todas as classes, campos e métodos impactados por conjuntos de junção. Por essa razão, AAIs constituem um instrumento útil para auxiliar na detecção de fragilidades em conjuntos de junção. Por outro lado, aspectos anotadores são responsáveis por introduzir anotações no programa base, de forma não invasiva. Além de aumentar a robustez dos conjuntos de junção, aspectos anotadores evitam dois problemas decorrentes do uso de anotações convencionais: entrelaçamento e espalhamento do código relativo às anotações. Uma ferramenta foi desenvolvida com o propósito de gerar AAIs e aspectos anotadores de forma semi-automática. São apresentados também exemplos de aplicação da ferramenta proposta em dois sistemas reais orientados por aspectos, além de uma análise de robustez da solução proposta envolvendo possíveis manutenções no sistema clássico do Editor de Figuras.

Abstract

Aspects have been designed to increase reuse and comprehensibility and to reduce the maintenance effort required when evolving object-oriented systems. However, aspects can also compromise the implementation and evolution of a system. The main reason is that pointcuts use names and regular expressions to define join points in a base program. For example, AspectJ's pointcut model implies in a strong coupling between pointcuts and the structure of the base system. This problem has been already investigated by other researchers, being usually named as the fragile pointcut problem.

Annotations are often mentioned as a potential alternative to tackle the fragile nature of AspectJ's pointcuts. However, annotations themselves can be considered crosscutting elements because they are normally pervasive and tangled with business-specific functionality. In this work, we propose a solution to reduce the fragile pointcut problem that relies on two central components: aspect-aware interfaces and annotator aspects. Aspect-aware interfaces contain information about classes, fields and methods impacted by pointcuts. For this reason, AAIs represent a useful instrument to detect fragilities in pointcuts. On the other hand, annotator aspects are responsible for superimpose annotations to the base code in a non-invasive way. Besides increasing the robustness of pointcuts, the use of annotator aspects avoids two problems commonly associated to conventional annotations: scattering and tangling of annotations. In this master thesis work, a tool has been developed to generate AAIs and annotator aspects semi-automatically. We also illustrate the use of the proposed solution in pointcut descriptors of two real-world aspect-oriented systems. Moreover, we describe a robustness study that has evaluated the proposed solution in face of possible changes to the classical Figure Editor system.

Agradecimentos

Ao meu orientador e amigo, Prof. Marco Túlio de Oliveira Valente, pelo constante incentivo, sempre indicando a direção a ser tomada em momentos de dificuldade.

À minha esposa, pelo carinho, compreensão e apoio em todos os momentos nos quais estive envolvido com este trabalho de dissertação.

À minha mãe, que me acompanha desde o meu nascimento e que com certeza estará sempre presente na minha vida.

Aos meus colegas de trabalho e à diretoria do Datapuc, que em vários momentos permitiram que eu tivesse uma flexibilidade no meu horário, para que eu pudesse conciliar o trabalho com os estudos.

A todos aqueles que se fizeram presentes, que se preocuparam, que foram solidários, que torceram por mim.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Programação orientada por aspectos estende paradigmas tradicionais de programação com novas abstrações destinadas a modularizar a implementação de interesses transversais (*crosscutting concerns*) [KLM+97]. Alguns dos exemplos mais comuns de interesses transversais são: *logging*, autenticação, controle de acesso, distribuição, controle de concorrência, controle de transações, persistência, etc. O desenvolvimento de software sem a devida separação de interesses transversais resulta nos fenômenos a seguir:

- Código entrelaçado (*code tangling*): ocorre quando a implementação do objetivo central de um módulo concorre com outros interesses.
- Código espalhado (*code scattering*): ocorre quando os interesses transversais estão inseridos em diversos módulos, isto é, a mesma codificação está presente em diversos pontos do sistema.

Linguagens orientadas por aspectos foram desenvolvidas com o objetivo principal de reduzir ao máximo o entrelaçamento e o espalhamento de código em um sistema. Em AspectJ [KHH+01, Lad03] – a linguagem de programação orientada por aspectos mais utilizada atualmente – aspectos podem declarar conjuntos de junção (*pointcuts*), os quais especificam de forma precisa pontos bem definidos da execução de um programa Java, os quais são chamados de pontos de junção (*join points*). Adendos (*advices*) correspondem a métodos anônimos que são implicitamente chamados antes, depois ou no lugar de pontos de junção. Em essência, aspectos foram propostos para incrementar os níveis de reúso, para facilitar o entendimento e para diminuir o esforço demandado por manutenções em sistemas orientados por objetos.

No entanto, aspectos também podem comprometer a implementação e evolução em separado de módulos de um sistema, isto é, podem impactar um dos principais benefícios conquistados ao longo de décadas por técnicas e conceitos de programação modular [Par72]. O principal motivo é que conjuntos de junção utilizam nomes e expressões regulares para definir pontos de junção de um programa alvo. Se por um lado isso viabiliza quantificação, por outro torna os conjuntos de junção sensíveis a mudanças nos nomes e nas abstrações do programa alvo. Por exemplo, uma simples alteração no nome de um método pode, como efeito colateral, remover ou inserir pontos de junção em um dado conjunto de junção. Como um programa é normalmente implementado sem conhecimento dos aspectos que atuarão sobre o mesmo (propriedade conhecida como *obliviousness* [FF00]), uma alteração como a mencionada pode, de forma silenciosa, habilitar a execução de adendos indesejáveis ou então desabilitar a execução de adendos necessários ao correto funcionamento de um sistema. Em outras palavras, o modelo de definição de conjuntos de junção de linguagens como AspectJ, a fim de conciliar quantificação e *obliviousness*, acaba por produzir um importante efeito colateral: a criação de um forte acoplamento entre conjuntos de junção e a estrutura do programa alvo. Este problema já foi investigado por outros pesquisadores, sendo usualmente chamado de problema dos conjuntos de junção frágeis (*fragile pointcut problem*) [KS04, SG05, KMBG06a, KMBG06b].

1.1.1 Conjuntos de junção frágeis

Tradicionalmente, em orientação por objetos, o acoplamento entre o código cliente e o código que implementa um módulo (ou classe) é controlado por meio de interfaces. O cliente adquire o direito de usar os métodos de uma interface e a classe assume o compromisso de implementar tais métodos. Caso a assinatura dos métodos de uma interface seja alterada, esta alteração impacta tanto os clientes da interface, como as classes que implementam a mesma. Além disso, este impacto é verificado pelo compilador, que notifica os clientes sobre a necessidade de utilizar os métodos com suas novas assinaturas e as classes implementadoras sobre a necessidade de prover uma implementação para os novos métodos.

No caso de orientação por aspectos, é interessante observar que aspectos podem ser vistos como clientes do programa base [Ste06, OMB05]. Isto é, embora aspectos complementem o programa base com código para implementação modular de requisitos transversais, este código não é explicitamente chamado pelo programa base. Em vez disso, para seu correto funcionamento, aspectos pressupõem que o programa base disponibiliza determinados pontos de junção, os quais funcionam como ganchos para chamada implícita

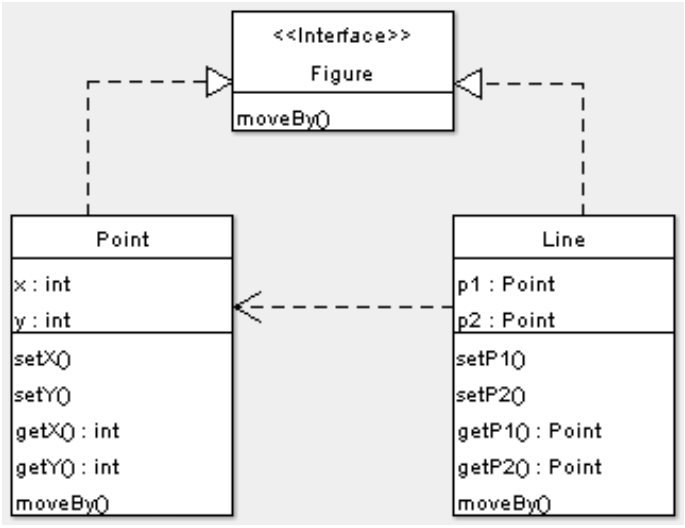

Figura 1.1: Programa base

de adendos[1]. No entanto, esta dependência entre aspectos e programa base não é documentada nem verificada por um contrato (ou interface). Com isso, estabelecem-se as condições para ocorrência do problema dos conjuntos de junção frágeis. Mais especificamente, o problema dos conjuntos de junção frágeis em sistemas orientados por aspectos se manifesta quando, em razão de uma modificação no programa base, conjuntos de junção passam a capturar pontos de junção indesejados ou deixam de capturar determinados pontos de junção essenciais ao correto funcionamento de um sistema [KMBG06a].

Para caracterizar melhor este problema, suponha o sistema descrito pelo diagrama de classes da Figura 1.1, usado com freqüência para ilustrar o uso de aspectos e de conceitos relacionados com este paradigma. Suponha ainda o seguinte aspecto, usado para capturar o requisito transversal que define quando o desenho de figuras deve ser atualizado no `Display` do sistema:

```
aspect UpdateSignaling {
   pointcut change(): execution(void Figure+.set*(..))
      || execution(void Figure+.moveBy(int, int));
   after() returning: change() {
     Display.update();
   }
}
```

[1] Esta inversão de responsabilidades constitui uma diferença fundamental entre orientação por objetos e orientação por aspectos. Em orientação por objetos, uma classe cliente explicitamente invoca métodos implementados por uma classe servidora. Em orientação por aspectos, cabe a um aspecto – que, do ponto de vista desta explicação, constitui um cliente – implementar métodos que serão implicitamente chamados pelo código base.

Supondo o sistema da Figura 1.1 como programa base, descrevem-se a seguir as duas principais situações nas quais a natureza frágil dos conjuntos de junção de AspectJ se manifesta:

- **Captura acidental de pontos de junção:** Suponha que um novo campo, chamado `date`, seja introduzido na classe `Line` para armazenar a data e a hora em que foi realizada a última alteração em um objeto desta classe. Para isso, acrescentam-se também na classe `Line` os métodos `setDate` e `getDate`. Após esta alteração no programa base, o conjunto de junção `change` do aspecto mostrado irá incluir um novo ponto de junção, correspondente à execução do método `setDate`. No entanto, esta captura é acidental, pois viola a premissa básica de definição deste conjunto de junção, segundo a qual o mesmo somente deve capturar a execução de métodos que alteram propriedades visuais de uma figura. Além de acidental, esta captura é também silenciosa, visto que nem o desenvolvedor da classe, nem o desenvolvedor do aspecto são notificados a respeito da mesma.

- **Falha na captura de pontos de junção:** Suponha que uma alteração no programa base renomeie os métodos `setX` e `setY` da classe `Point` para, respectivamente, `changeX` e `changeY`. Com isso, o conjunto de junção `change` não irá mais capturar a execução dos métodos renomeados. No entanto, para o correto funcionamento do sistema, eles deveriam continuar sendo capturados, pois efetuam modificações em propriedades visuais de um ponto. Além disso, esta falha de captura também é silenciosa.

A captura acidental de pontos de junção ocorre notadamente quando conjuntos de junção são definidos por meio de expressões regulares. Basicamente, uma evolução no programa pode fazer com que estas expressões passem a capturar pontos de junção indesejados. Já uma falha na captura de pontos de junção pode ocorrer em conjuntos de junção definidos tanto por meio de expressões regulares como por enumerações. Assim, a princípio, poder-se-ia supor que conjuntos de junção devem preferencialmente ser definidos por meio de enumerações (a fim de evitar capturas acidentais). No entanto, tais definições são mais susceptíveis a falhas na captura de pontos de junção do que definições baseadas em expressões regulares. A razão é que mesmo mudanças simples no programa base costumam demandar revisões nas enumerações usadas em conjuntos de junção [GB03].

Os dois problemas mencionados, além de ocorrerem na especificação de pontos de junção por meio dos designadores `call` e `execution`, podem ocorrer também quando da

especificação de pontos de junção via designadores `set` e `get`. Neste caso, eles ocorrem quando adicionam-se ou removem-se campos em classes do programa base.

1.2 Objetivos

O objetivo central desta dissertação é apresentar uma proposta para atenuar o problema dos conjuntos de junção frágeis de AspectJ baseada na combinação de dois conceitos: *aspect-aware interfaces* (AAIs) e aspectos anotadores. Basicamente, uma *aspect-aware interface* estende uma interface tradicional de linguagens orientadas por objetos com informações sobre os adendos que são habilitados pela chamada ou execução de seus métodos. AAIs foram originalmente propostas por Kiczales e Mezini [KM05a] para tratar um outro problema típico de orientação por aspectos: a impossibilidade de se raciocinar localmente sobre um determinado módulo, visto que aspectos podem modificar o comportamento de qualquer ponto de junção do mesmo. A idéia é que, dispondo das AAIs de um módulo, um programador pode desenvolver uma forma expandida de raciocínio modular, isto é, um raciocínio que considera não só o código do módulo, mas também dos aspectos aplicados sobre o mesmo. Portanto, na proposta de Kiczales e Mezini, AAIs constituem um documento extra ao código do sistema, o qual tem como objetivo viabilizar raciocínio modular na presença de aspectos. Nesta dissertação, pretende-se usar AAIs com um segundo objetivo: detectar ampliações e reduções silenciosas nos conjuntos de junção de um programa.

Pretende-se também investigar o uso de anotações para a criação de conjuntos de junção mais robustos. Na atual versão de AspectJ, anotações permitem definir conjuntos de junção que não são baseados em elementos sintáticos do programa base, mas sim em atributos que definem propriedades semânticas desses elementos. Entretanto, na atual versão de AspectJ, a utilização de anotações juntamente com aspectos apresenta os seguintes problemas:

- *Anotações representam interesses transversais.* Anotações em Java apresentam um comportamento transversal porque são normalmente invasivas (o que gera entrelaçamento de código) e necessárias em vários pontos do programa (o que gera espalhamento de código).

- *Conjuntos de junção baseados em anotações são frágeis.* Tradicionalmente, conjuntos de junção baseados em anotações pressupõem que os desenvolvedores tenham anotado corretamente o programa base. Anotações colocadas fora do local correto, ou que não atendam às convenções de nomes esperadas, podem, silenciosamente,

desabilitar pontos de junção que deveriam ser capturados pelos aspectos de um sistema.

A solução proposta investiga o uso de conjuntos de junção baseados em anotações de forma que os problemas descritos acima sejam minimizados. Nesta solução, anotações são introduzidas no código base de forma não invasiva, através do uso de aspectos anotadores. Esses aspectos são gerados de forma semi-automática, a partir de uma linguagem declarativa usada para definir as anotações de um determinado sistema.

1.3 Principais contribuições

A seguir são descritas as principais contribuições alcançadas com este trabalho:

- Projeto e implementação de um sistema para geração automática de AAIs, a partir das classes e dos aspectos de uma aplicação Java. O objetivo é fornecer um documento que possa ser usado pelos desenvolvedores de sistemas orientados por aspectos para auxiliar na detecção de alterações indesejadas nos pontos de junção capturados pelos aspectos definidos em um sistema.

- Elaboração de uma proposta para atenuar o problema de fragilidade dos conjuntos de junção em AspectJ utilizando anotações e AAIs. Os componentes centrais da solução proposta são chamados de aspectos anotadores (*annotator aspects*) que são responsáveis pela introdução de anotações no código base de forma controlada e não invasiva. Os aspectos anotadores são gerados de forma semi-automática, partindo de uma linguagem desenvolvida especificamente para definição de anotações. As AAIs são utilizadas na identificação de alterações nos pontos de junção do programa base que poderiam afetar o funcionamento dos aspectos anotadores.

- Apresentação de exemplos envolvendo sistemas reais orientados por aspectos, HealthWatcher e JAccounting, que utilizam a solução proposta para reduzir a fragilidade dos conjuntos de junção dessas aplicações. Descreve-se também um estudo detalhado para avaliar a robustez da solução proposta diante de alterações e evoluções realizadas no exemplo clássico do Editor de Figuras, mostrado na Figura 1.1.

1.4 Estrutura da dissertação

O restante desta dissertação está organizado da seguinte forma:

- O Capítulo 2 apresenta os principais trabalhos relacionados com o tema da dissertação.
- O Capítulo 3 apresenta a geração e utilização de *aspect-aware interfaces* para auxiliar na identificação de capturas adicionais e falhas na captura de pontos de junção.
- O Capítulo 4 apresenta o conceito de aspectos anotadores e o uso de anotações para classificar objetos segundo suas características semânticas.
- O Capítulo 5 apresenta as conclusões e indica possíveis linhas de trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Este capítulo descreve os principais trabalhos relacionados com o tema desta dissertação. Inicialmente, é feita uma introdução ao conceito de anotações e sobre como utilizá-las para anotar uma aplicação Java. Em seguida são apresentados os principais comandos e estruturas da linguagem orientada por aspectos AspectJ. Os trabalhos apresentados em seguida, que estão na ordem cronológica de suas publicações, são: análise de delta de conjuntos de junção [KS04, SG05], *aspect-aware interfaces* (AAIs) [KM05a], *open modules* [Ald05], *crosscutting interfaces* (XPIs) [SGS+05], conjuntos de junção baseados em modelos [OMB05, KMBG06a, CPA06, CKAA06], introdução de anotações usando aspectos [HNBA06] e linguagens para definição de regras de projeto [DNBS07].

2.1 Anotações

A linguagem Java permite o uso de anotações para associar informações semânticas a elementos (métodos, classes, etc) de um determinado sistema [Mic]. A anotação `@deprecated`, por exemplo, é utilizada para marcar métodos que estão ultrapassados, e que não devem mais ser utilizados. A partir do lançamento da versão 5.0 de Java, desenvolvedores têm a possibilidade de criar suas próprias anotações. Anotações não afetam diretamente a semântica das aplicações, mas sim a maneira como essas são tratadas por ferramentas e bibliotecas que se baseiam em anotações para realização de suas tarefas. Anotações podem ser lidas diretamente do código fonte ou por meio de reflexão.

A definição de uma anotação é feita de forma semelhante a de uma interface em uma aplicação Java. O símbolo `@` precede a palavra reservada `interface`. Cada método declarado nessa interface define um elemento da anotação. Esses elementos não podem possuir parâmetros e nem gerar exceções. É possível definir um valor *default* como retorno dos elementos. Anotações que não possuem elementos são chamadas de marcadores (*markers*).

Após a declaração de uma anotação, a mesma pode ser utilizada para anotar elementos da aplicação. A anotação é um tipo especial de modificador e, portanto, pode ser inserida nos mesmos locais onde os demais modificadores, tais como `public`, `static` ou `final`, são usados. Para marcar o código com uma anotação deve-se utilizar o símbolo `@` seguido do nome da anotação e, quando necessário, de uma lista, entre parêntesis, contendo os elementos da anotação e seus respectivos valores.

Definições de anotações também podem ser anotadas. O exemplo a seguir apresenta a definição da anotação `@Test`, que é anotada pela anotação `@Target(<ElementType>)`:

```
@Target(ElementType.METHOD)
public @interface Test { }
```

A anotação `@Target(<ElementType>)`, nativa da linguagem Java, especifica qual tipo de elemento será alvo da anotação que está sendo definida. No caso do exemplo anterior, o parâmetro `ElementType.METHOD` determina que a anotação `@Test` será utilizada para anotar métodos.

O trecho de código a seguir mostra a utilização da anotação `@Test`, definida no exemplo anterior, para marcar métodos do programa base que precisam ser testados:

```
class Foo {
   ...
   @Test public static void m1() { ... }
   public static void m2() { ... }
   @Test public static void m3() { ... }
   @Test public static void m4() { ... }
   ...
}
```

O exemplo a seguir identifica e executa métodos que foram marcados com a anotação `@Test`.

```
1:  public class RunTest {
2:    public static void main(String[] args) throws Exception {
3:       for (Method m : Class.forName(args[0]).getMethods())
4:          if (m.isAnnotationPresent(Test.class))
5:             try {
6:                m.invoke(null);
7:                System.out.printf("Test %s succeeded!", m);
8:             } catch (Throwable ex) {
```

```
9:                 System.out.printf("Teste %s failed: %s%n", m, ex.getCause());
10:            }
11:    }
12: }
```

O programa anterior basicamente usa reflexão para obter informações sobre os métodos de uma classe fornecida como parâmetro na linha de comando. O programa verifica então, para cada método, se o mesmo possui a anotação `@Test` (linha 4). Caso a anotação seja encontrada, o método será chamado (linha 6). Se a execução gerar uma exceção do tipo `Throwable`, o programa exibirá a mensagem de erro correspondente (linha 9).

2.2 AspectJ

O objetivo desta seção é introduzir algumas funcionalidades da linguagem AspectJ que serão úteis para o entendimento do restante desta dissertação.

O surgimento das linguagens orientadas por aspectos se deve principalmente ao fato de que os mecanismos de modularização de linguagens orientadas por objetos não são capazes de tratar adequadamente interesses transversais de um sistema [KHH+01, GL03].

AspectJ suporta dois tipos de implementações de interesses transversais: dinâmica e estática. Transversalidade estática permite alterar as assinaturas estáticas das classes e interfaces de um programa Java. Transversalidade dinâmica, que é a forma mais simples e usual para definição de interesses transversais, permite declarar conjuntos de junção (*pointcuts*), os quais especificam de forma precisa pontos bem definidos da execução de um programa Java, chamados de pontos de junção (*join points*). Métodos especiais que definem o comportamento dos pontos de junção são chamados de adendos (*advices*). Um adendo pode ser executado antes, depois, ou "no lugar" dos pontos de junção especificados. Essas construções ficam reunidas em unidades chamadas de aspectos.

O conjunto de junção a seguir agrupa todas as chamadas do método `setX` da classe `Point`:

```
pointcut move(): call(void Point.setX(int))
```

Conjuntos de junção podem ser combinados por meio dos operadores lógicos *e* (`&&`), *ou* (`||`) e *negação* (`!`). Por exemplo, o conjunto de junção a seguir captura chamadas dos métodos `setX` ou `setY` da classe `Point`:

```
pointcut move(): call(void Point.setX(int)) || call(void Point.setY(int))
```

O exemplo de conjunto de junção apresentado anteriormente baseia-se na enumeração de uma ou mais assinaturas de métodos. É possível também definir conjuntos de junção que utilizam expressões regulares. Por exemplo, o conjunto de junção a seguir captura chamadas de métodos cujos nomes começam com `set` ("`set*`"), como `setX` ou `setY`, de todas as classes e subclasses que implementam `Figure` ("`Figure+`"), independente de quais são os parâmetros ("`(..)`") desses métodos.

```
pointcut move(): call(void Figure+.set*(..));
```

Além de expressões regulares, alguns seletores podem ser utilizados para identificar outras propriedades dos pontos de junção. O seletor `cflow(<conjunto de junção>)`, por exemplo, é capaz de determinar se um ponto de junção ocorre no fluxo de execução de outro conjunto de junção. O trecho de código a seguir apresenta um exemplo da utilização do seletor `cflow`:

```
pointcut changePoint():
   call(public void Point.setX(..)) ||
   call(public void Point.setY(..));
pointcut changeLine(): call(public void Line.setPoint(..));
pointcut localChange(): changePoint() && cflow(changeLine());
```

No exemplo anterior, o conjunto de junção `changePoint` captura chamadas dos métodos `setX` e `setY` da classe `Point`. O conjunto de junção `changeLine` captura chamadas do método `setPoint` da classe `Line`. Já o conjunto de junção `localChange` engloba todas as chamadas capturadas por `changePoint` que tenham sido feitas no fluxo de execução de métodos capturados por `changeLine`.

O seletor `cflowbelow(<conjunto de junção>)` também determina se um ponto de junção ocorre no fluxo de execução de outro conjunto de junção, porém sem considerar o próprio ponto de junção que originou a chamada.

Adendos associam conjuntos de junção a trechos de código específicos, que são executados antes, depois, ou "no lugar" dos pontos de junção. O exemplo a seguir apresenta um trecho de código que é executado antes (`before`) dos pontos de junção capturados pelo conjunto de junção `move()`:

```
before(): move() {
    Display.update();
}
```

2.2.1 Anotações em AspectJ

AspectJ permite a definição de conjuntos de junção baseados na presença ou ausência de anotações [Foub]. A sintaxe utilizada para representar as anotações é a mesma utilizada em Java. O exemplo a seguir apresenta um conjunto de junção baseado na presença da anotação `@Displayable` em qualquer método da aplicação:

```
pointcut change(): execution(@Displayable * *(..));
```

Além disso, é possível introduzir anotações no código base de forma não invasiva, com o uso de aspectos. Para tal, deve-se utilizar a seguinte diretiva:

```
declare @<kind>: ElementPattern:Annotation;
```

Neste caso, `<kind>` é o tipo do elemento que será anotado (método, classe, campo, etc), `ElementPattern` é o padrão de nomenclatura que identifica o elemento e `Annotation` representa a anotação que será introduzida na aplicação. O exemplo a seguir mostra a adição da anotação `@transactional` no método `setColor` da classe `Point`.

```
declare @method: public void Point.setColor(java.awt.Color): @transactional;
```

2.3 Análise de delta de conjuntos de junção

O objetivo principal da análise de delta de conjuntos de junção (*pointcut delta analysis*) [SG05] é apresentar uma solução para o problema dos conjuntos de junção frágeis por meio da comparação de duas versões orientadas por aspectos de um mesmo sistema, indicando diferenças entre os relacionamentos transversais das mesmas. Essas diferenças podem incluir novos pontos de junção que são capturados ou então pontos de junção que deixaram de ser capturados por conjuntos de junção na nova versão do sistema. Um *plug-in* para o Eclipse, denominado PCDiff, foi criado para implementar a análise de delta de conjuntos de junção [KS04]. Esse *plug-in* permite selecionar as versões que devem ser comparadas e visualizar os resultados em uma janela do ambiente Eclipse.

O processo de análise de delta de conjuntos de junção avalia o impacto de manutenções realizadas, tanto no código base quanto nos aspectos, comparando a versão inicial, sem as alterações, com uma versão gerada após a implementação das alterações. Para isso, é necessário armazenar uma imagem da versão original antes de iniciar as alterações. Na interface disponibilizada pelo PCDiff, o comando *Take Snapshot* é responsável por criar essa imagem, que irá conter informações relativas aos pontos de junção capturados

juntamente com a relação de todos os adendos a eles associados. Como adendos não possuem um nome, a ferramenta introduz um comentário no código fonte com uma chave identificadora única para representar cada adendo. Pontos de junção são identificados por meio das assinaturas dos métodos capturados no código base. Depois de implementadas as alterações requisitadas por uma nova versão do sistema, uma nova imagem é criada para que possa ser comparada com a imagem anterior às alterações. Durante esse processo de comparação, a análise de delta de conjuntos de junção é utilizada para identificar possíveis falhas de captura e capturas indesejadas de pontos de junção, que são os problemas característicos de conjuntos de junção frágeis. Quando a análise termina, as diferenças encontradas são apresentadas na forma de árvore em uma janela do Eclipse, denominada *Diff Visualizer*. Esta janela é dividida em três partes: dados estatísticos (*PointcutDiff Statistics*), hierarquia de adendos (*Advice Hierarchy*) e hierarquia de mudanças (*Change Hierarchy*). A sub-janela de dados estatísticos mostra um resumo das diferenças encontradas nas duas versões do sistema, incluindo os novos pontos de junção e adendos associados. A sub-janela de hierarquia de adendos apresenta uma visão diferente, ordenada por adendos, para exibir as mesmas informações, enquanto que a sub-janela de hierarquia de mudanças apresenta as alterações ordenadas pelos tipos das mudanças realizadas. Como complemento ao *Diff Visualizer*, os sinais de + e - aparecem no código base como marcadores especiais, indicando adição ou remoção de métodos capturados com relação à versão inicial. Esses marcadores são semelhantes aos símbolos utilizados pelo próprio Eclipse para expor os pontos de junção.

A análise de delta de conjuntos de junção utiliza conjuntos que contêm os relacionamentos transversais das duas versões analisadas. Esses relacionamentos são avaliados com base nas seguintes situações que podem ocorrer em função de uma manutenção no sistema:

1. Alterações nas definições dos conjuntos de junção, as quais podem acarretar alterações nas relações de captura dos pontos de junção.

2. Alterações no código base. São as mais comuns e representam a maioria dos problemas relacionados a mudanças indesejadas nos relacionamentos transversais de um sistema.

Para um programa orientado por aspectos P, a função que descreve o conjunto dos relacionamentos transversais possui a seguinte assinatura:

$$match : P \rightarrow JP \times ADV \times Q$$

onde:

- JP é o conjunto de todos os pontos de junção em P,
- ADV é o conjunto de todos os adendos em P e
- Q é o tipo do relacionamento transversal, que pode ser dinâmico ou estático.

Dessa forma, a função $match(P)$ retorna os pontos de junção da aplicação P que são capturados por conjuntos de junção, a relação de adendos associados a cada um deles e os tipos dos relacionamentos transversais que associam os pontos de junção aos adendos.

Considerando P a versão anterior do sistema e P' a sua versão mais nova, a expressão anterior deve ser aplicada sobre ambas versões para comparar as informações de $match(P)$, com sua versão editada $match(P')$, visando identificar pontos de junção e adendos que apresentam divergências.

Os autores visualizam um cenário para desenvolvimento de sistemas orientados por aspectos com análise de delta onde existem três equipes distintas: uma responsável pelo programa base, outra equipe que cuida dos aspectos e uma terceira que é responsável pela qualidade e pelos testes finais. Nesse caso, o PCDiff seria utilizado pela equipe de testes.

Apesar de apresentar as diferenças entre conjuntos de junção de duas versões distintas, a análise de delta não é capaz de detectar a inserção de novos pontos de junção no programa base que deveriam, mas que efetivamente não são capturados por um determinado conjunto de junção. Isso ocorre quando os referidos pontos de junção não seguem as convenções de nomes pressupostas no código dos aspectos. Por exemplo, na Figura 1.1 da Seção 1.1.1, suponhamos que sejam adicionados na classe `Point` um campo `color`, representando a cor do ponto, e um método `changeColor(Color)` que altera a cor do ponto. Chamadas do método `changeColor` deveriam ser capturadas pelo conjunto de junção `change` do aspecto `UpdateSignaling`, mostrado na Seção 1.1.1, pois `changeColor` altera a cor de uma figura. No entanto, o conjunto de junção `change` captura apenas chamadas dos métodos `moveBy` ou `set*`, não reconhecendo portanto a adição do método `changeColor`.

AJDT *Crosscutting Comparison*: O *plug-in* AJDT (*AspectJ Development Tools*) é responsável por tornar o ambiente Eclipse compatível com AspectJ [Foua]. A partir da versão 1.3, o AJDT inclui uma funcionalidade, denominada *Crosscutting Comparison*, que compara duas versões de um sistema. Essencialmente, essa nova funcionalidade do AJDT possui as mesmas características da ferramenta PCDiff.

2.4 Aspect-aware interfaces

Aspect-Aware Interfaces (AAIs) foram originalmente propostas por Kiczales e Mezini para tentar minimizar a dificuldade de se raciocinar localmente sobre módulos de sistemas orientados por aspectos, visto que aspectos podem modificar o comportamento de qualquer ponto de junção da aplicação [KM05a]. A idéia é que, dispondo das AAIs de um módulo, um programador possa desenvolver uma forma expandida de raciocínio modular, isto é, um raciocínio que considera não só o código do módulo, mas também dos aspectos aplicados sobre o mesmo. Portanto, na proposta de Kiczales e Mezini, AAIs constituem um documento extra ao código do sistema, o qual tem como objetivo viabilizar raciocínio modular na presença de aspectos.

AAIs podem ser definidas para classes e interfaces. Basicamente, a AAI associada a uma classe lista a assinatura de todos os métodos desta classe, incluindo construtores, independentemente de serem públicos ou não. Uma AAI lista também a declaração de todos os campos da classe (públicos e não-públicos). Interfaces convencionais de Java também possuem uma AAI. No caso de interfaces, AAIs listam os métodos das mesmas. Tanto no caso de interface ou de classe, para cada método definido em uma AAI são acrescentadas informações sobre os possíveis adendos que são implicitamente executados quando tais métodos são chamados. Esta definição inclui o tipo do adendo (`after`, `before` ou `around`), o nome do aspecto ao qual ele pertence e o nome do conjunto de junção associado ao mesmo. No caso de campos, informam-se os adendos executados quando de uma leitura ou escrita em seus valores.

Na Figura 2.1, mostram-se as AAIs geradas para as interfaces e classes do Editor de Figuras, mostrado na Seção 1.1.1. Para ser mais claro, suponha a seguinte entrada da AAI associada à classe `Line`:

```
void setP1(Point): after returning UpdateSignaling.change();
```

Esta entrada informa que no aspecto `UpdateSignaling` existe um adendo do tipo `after returning` associado ao conjunto de junção `change`. Este adendo captura chamadas ou execuções do método `setP1` da classe `Line`.

A proposta original de Kiczales e Mezini contempla também a definição de AAIs para aspectos. Basicamente, a AAI associada a um aspecto possui uma entrada para cada um de seus adendos. Nestas entradas, além do tipo do adendo, informam-se também os pontos de junção do programa base sobre o qual ele se aplica. Ou seja, AAIs associadas a aspectos disponibilizam informação inversa àquela provida por AAIs associadas a classes, conforme pode ser visto no último trecho de código da Figura 2.1.

```
interface Figure
  void moveBy(int, int): after returning UpdateSignaling.change();

Point implements Figure
  int x;
  int y;
  void setY(int): after returning UpdateSignaling.change();
  void setX(int): after returning UpdateSignaling.change();
  void moveBy(int, int): after returning UpdateSignaling.change();

Line implements Figure
  Point P1;
  Point P2;
  void setP1(Point): after returning UpdateSignaling.change();
  void setP2(Point): after returning UpdateSignaling.change();
  void moveBy(int,int): after returning UpdateSignaling.change();

UpdateSignaling
  after returning UpdateSignaling.change():
    Figure.moveBy(int,int),
    Point.setX(int),
    Point.setY(int),
    Point.moveBy(int,int),
    Line.setP1(Point),
    Line.setP2(Point),
    Line.moveBy(int, int);
```

Figura 2.1: Exemplos de AAIs

2.5 Open modules

Assim como ocorre com *aspect-aware interfaces*, a solução que utiliza *open modules* tem como foco a questão do raciocínio modular em sistemas orientados por aspectos [Ald05]. Porém, enquanto AAIs fornecem uma documentação de apoio para os desenvolvedores, *open modules* procura restringir o poder de ação dos aspectos para que os mesmos não interfiram de forma não autorizada no código do programa base. Para tanto, cada módulo do sistema deve expor quais pontos de junção podem ser capturados por aspectos. Isso funciona como uma espécie de contrato entre o módulo e o aspecto, de forma que o aspecto só pode utilizar o que for exportado pelo módulo. Além de pontos de junção específicos, *open modules* podem exportar também conjuntos de junção.

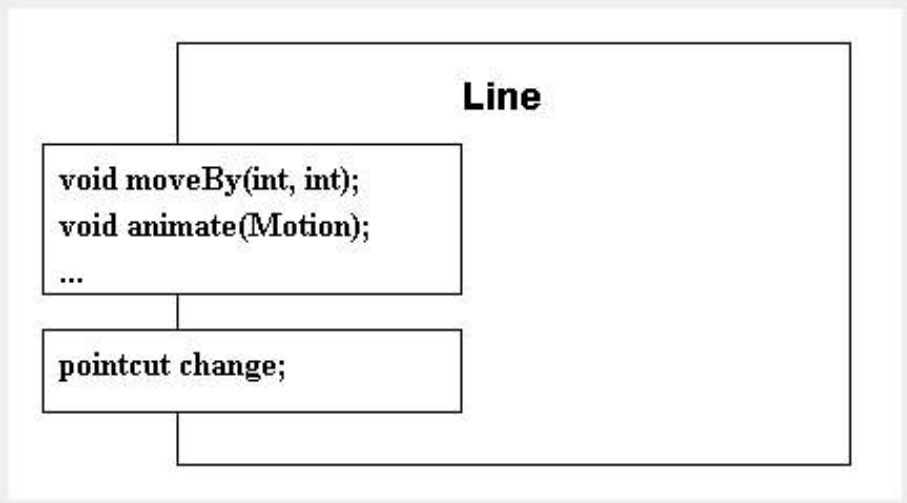

Figura 2.2: Visão conceitual de *open modules*

A Figura 2.2 apresenta uma visão conceitual de *open modules*. Nesse exemplo, a classe `Line` exporta dois métodos, `moveBy` e `animate`, e um conjunto de junção denominado `change`. O conjunto de junção `change` é disparado sempre que um objeto do tipo `Line` se move. Aspectos podem construir conjuntos de junção baseados nas funções exportadas pelo módulo ou podem utilizar diretamente o conjunto de junção `change` para acionar adendos relacionados ao movimento de objetos do tipo `Line`. Entretanto, aspectos não podem capturar chamadas de outros métodos internos ao módulo. Suponhamos agora que, durante a evolução do sistema, o desenvolvedor crie outro método que também faz com que objetos do tipo `Line` se movam. Nesse caso, o desenvolvedor deve alterar o conjunto de junção `change` para garantir que ele capture o novo método que foi criado.

Assim como ocorre com sistemas orientados por aspectos tradicionais, sempre que o módulo base evoluir, deve-se verificar a necessidade de alterar a definição do conjunto de junção para que o mesmo acompanhe a evolução do sistema. A diferença é que, com *open modules*, as duas alterações ocorrem no mesmo módulo, ou seja, no caso da Figura 2.2, ocorrem no escopo da classe `Line`. Isso não significa que o conjunto de junção `change` deixou de ser frágil com relação a mudanças no programa base, mas, ao implementar os dois códigos que precisam ser alterados em um único módulo, a idéia é que o mesmo desenvolvedor responsável pelo módulo `Line` seja também responsável pela definição dos conjuntos de junção exportados por ele.

Outro ponto a ser considerado é que a solução baseada em *open modules* sacrifica claramente a propriedade de *obliviousness* em POA, uma vez que transfere parte do código que implementa os interesses transversais para módulos do programa base. Além disso, a definição de conjuntos de junção no escopo de módulos pode ser considerada um

interesse transversal, pois, no caso do exemplo da Figura 2.2, deve ser repetida em todos os módulos que representam figuras.

2.6 Crosscutting interfaces (XPIs)

Conforme proposto por Sullivan e colegas, uma interface transversal define um conjunto de regras de projeto (*design rules*) que desenvolvedores do código base devem seguir e nas quais projetistas de aspectos podem se basear [GSS+06, SGS+05]. Em essência, a idéia é separar aspectos de detalhes de implementação do código base de um sistema, permitindo que tanto aspectos quanto classes possam evoluir de forma independente, desde que respeitem as regras de projeto especificadas nas XPIs. Uma XPI deve conter um nome, conjuntos de junção abstratos, um escopo sobre o qual os pontos de junção atuarão e uma implementação parcial das regras de projeto. Cada conjunto de junção abstrato é formado por um descritor de conjunto de junção e pela especificação da semântica de pré-condições e pós-condições que devem ser satisfeitas toda vez que um adendo for executado. A implementação parcial de uma XPI define o padrão que os pontos de junção reais devem ter para serem aceitos, além de um conjunto de regras que determinam como o código da aplicação deve ser escrito para seja corretamente tratado pela XPI.

Assim como interfaces de linguagens orientadas por objetos, uma XPI define um contrato de utilização entre fornecedores de serviços e usuários. A implementação de uma XPI é feita então por meio da definição e codificação dos descritores de conjuntos de junção baseados nos padrões definidos pela interface. Além disso, os desenvolvedores do núcleo de um sistema não precisam conhecer os aspectos específicos que implementarão os interesses transversais, mas devem sim decidir quais abstrações devem ser disponibilizadas para as XPIs, facilitando o desenvolvimento e a evolução do código dos aspectos. Da mesma forma, os responsáveis pela criação dos aspectos não precisam conhecer detalhes do código sobre os quais os aspectos atuarão, bastando seguir as regras definidas pelas XPIs.

Na proposta mais recente de implementação de XPIs, regras de projeto são expressas em AspectJ [GSS+06]. O exemplo a seguir apresenta uma regra de projeto que define que campos da classe `Point` somente podem ser alterados em métodos `set*` dessa classe.

```
public aspect XFigure {
  ...
  public pointcut change(): withincode (void Point.set*);
  ...
```

```
  declare error: ( !XFigure.change() && (set(Point.*) ):
    "Campos da classe Point somente podem ser alterados em métodos set*"
}
```

XPIs representam uma alternativa que abandona o conceito de *obliviousness*, uma vez que obriga os desenvolvedores a conhecerem e respeitarem as regras de projeto definidas para a aplicação. Além disso, o desenvolvimento baseado em XPIs permite também que os aspectos sejam implementados antes que as classes de negócio estejam prontas, possibilitando desenvolvimento em paralelo.

Em resumo, o projeto de uma XPI consiste em identificar e especificar propriedades estruturais que devem ser seguidas na implementação de um programa base. Como conseqüência, quando comparadas com AAIs, XPIs tendem a detectar com mais facilidade casos de métodos que não obedecem às convenções de nomes presentes na definição de conjuntos de junção. Por outro lado, existem dúvidas sobre o real poder de expressão de AspectJ como linguagem para definição de regras de projeto.

2.7 Conjuntos de junção baseados em modelos

Soluções baseadas em modelos propõem uma alteração no nível de abstração para declaração de conjuntos de junção, os quais passam a fazer parte de modelos conceituais. O objetivo é eliminar a dependência estrutural de aspectos em relação ao código fonte do programa base, além de prover mecanismos para documentação e verificação de regras de projeto impostas na criação dos conjuntos de junção. Com isso, os problemas de fragilidade dos conjuntos de junção passam a ser identificados em um modelo que representa os elementos do código fonte da aplicação. Nesta seção, são apresentados alguns dos principais trabalhos relacionados com o uso de modelos conceituais para definição de conjuntos de junção.

2.7.1 Intensional views

Intensional views representa uma técnica para descrever o modelo conceitual de um programa e verificar a consistência desse modelo com a estrutura do programa [KMBG06a]. A solução proposta por Kellens et al. utiliza o formalismo de *intensional views* para gerar os modelos conceituais e definir regras de projeto que irão estabelecer as relações de dependência existentes entre o código da aplicação e o código dos aspectos. A partir do modelo conceitual, os autores propõem a criação de conjuntos de junção baseados em

visões desse modelo (*view-based pointcuts*). Dessa forma, os problemas de captura acidental e falha na captura de pontos de junção irão ocorrer em situações de violação das regras pré-estabelecidas nos modelos.

A definição de um conjunto de junção é baseada em visões do modelo conceitual, as quais agrupam pontos de junção do programa base. As visões são criadas a partir de características estruturais comuns, compartilhadas entre elementos (classes, métodos, campos, etc) do programa. O trecho de código a seguir mostra um exemplo de um conjunto de junção cujo objetivo é para capturar métodos selecionados pela visão `AccessorMethods` e que possuem em seu escopo uma chamada a um método representado pela variável `?methSignature`:

```
pointcut accessors():
    classifiedAs(?methSignature, AccessorMethods) &&
    call(?methSignature);
```

No trabalho desenvolvido por Kellens et al. foi utilizado o conjunto de ferramentas IntensiVE para definir as visões que constituem o modelo conceitual de um programa e para verificar a consistência desse modelo com o código fonte. Um estudo de caso foi realizado com o sistema SmallWiki, o qual é um sistema de Wiki totalmente implementado em Smalltalk. Uma aplicação Wiki é uma aplicação *web* colaborativa que permite que usuários adicionem e editem o conteúdo de suas páginas. No estudo de caso realizado, foi criada uma visão denominada `WikiActions`, que modela o seguinte conceito: "todas as ações sobre páginas Wiki" (incluindo *save*, *login*, etc). Essa visão engloba todos os métodos cujo nome começa com `execute`, pois se observou que os métodos responsáveis pela implementação das ações realizadas sobre páginas respeitam essa convenção. Considerando-se futuras manutenções no código do sistema, a simples existência dessa visão não diminui a fragilidade dos conjuntos de junção que nela se baseiam, pois capturas acidentais e falhas na captura de pontos de junção podem ocorrer dependendo dos nomes dos métodos adicionados ou renomeados.

Para evitar que a fragilidade dos conjuntos de junção continue ocorrendo, são definidas regras de projeto que atuam sobre as visões, permitindo a validação de outras características que devem ser respeitadas pelos pontos de junção selecionados. As regras associadas às visões são implementadas em uma linguagem denominada Soul e são divididas em dois grupos:

- *Alternative Intensions.* Representam regras distintas que se complementam para validar os pontos de junção que devem ser capturados. Por exemplo, todos os métodos

que realizam ações sobre páginas Wiki, além de possuírem o nome começando com `execute`, são implementados em um protocolo chamado `action`[1]. Assim, ambas as alternativas são requeridas para determinar que um ponto de junção é uma *wiki action*. Durante o estudo de caso realizado, a ferramenta utilizada identificou que os métodos `save` e `authenticate` estão implementados em um protocolo chamado `action`, mas não podem ser classificados como *wiki actions*, pois seus nomes não começam com `execute`. A ferramenta identificou também que os métodos `executeSearch` e `executePermission` não estão no protocolo `action`, o que também faz com que eles atendam a apenas uma das regras requeridas pela visão `WikiActions`.

- *Intensional Relations.* Representam relacionamentos binários que são estabelecidos entre visões. Essas relações são representadas pela seguinte expressão: $Q_1\ x \in View_1\ :\ Q_2\ y\ \in\ View_2\ :\ x\ R\ y$, onde Q_i são quantificadores lógicos, $View_i$ são visões e `R` é o relacionamento binário verificado entre os elementos `x` e `y` de um programa. Por exemplo, pode-se definir uma relação do tipo: $\forall\ x \in WikiActions, \exists\ y \in ActionHandlers \mid x\ isImplementedBy\ y$, onde *isImplementedBy* é um relacionamento binário que verifica se um dado método `x` é implementado por uma determinada classe `y`.

As inconsistências detectadas na validação das visões e na verificação das regras de projeto sobre o código fonte indicam a possível presença de capturas acidentais ou falhas na captura de pontos de junção. Por exemplo, a visão `PageElements` agrupa todas as classes que representam componentes de páginas Wiki, tais como `Text`, `Links`, `Tables`, `Lists`, etc. A visão `Output Generation` agrupa todas as classes que geram arquivos de saída (HTML, Latex, TXT) para componentes de páginas Wiki. A relação "são aceitas por" determina que todos os componentes de páginas Wiki devem ser tratados, ou aceitos, por um elemento da visão `Output Generation`. Assim, no estudo de caso realizado, o conjunto de ferramentas IntensiVE foi capaz de advertir os desenvolvedores para a existência de uma inconsistência, pois os componentes `LinkInternal` e `LinkMailTo` não são tratados por nenhum elemento correspondente da visão `Output Generation`, o que poderá resultar em falhas na captura desses pontos de junção.

Após a definição do modelo conceitual, a implementação de conjuntos de junção baseados nesse modelo ocorre em uma linguagem orientada por aspectos chamada CARMA.

[1]Em Smalltalk, os métodos de uma classe podem ser organizados em grupos lógicos chamados de protocolos.

Essa linguagem é similar à linguagem AspectJ, porém estende Smalltalk, ao invés de Java. Conjuntos de junção definidos em CARMA podem ser estáticos ou dinâmicos, e fazem uso de predicados lógicos na construção de consultas que irão capturar os pontos de junção.

O trabalho apresentado utiliza a linguagem CARMA em conjunto com o conceito de *Intensional Views* buscando transferir o problema dos conjuntos de junção frágeis para o nível de modelo conceitual, onde capturas indesejadas e falhas de captura não estão mais associadas à estrutura do programa base, mas sim às regras pré-definidas. Não se sabe, entretanto, se o poder de definição das regras associadas às visões é suficiente para atender a todo espectro de problemas de conjuntos de junção frágeis. Além disso, a geração do modelo conceitual e o mapeamento desse modelo para o código fonte de um programa base não são tarefas simples, e requerem o domínio de linguagens para definição de conjuntos de junção baseadas em predicados lógicos.

2.7.2 Linguagem Alpha

A linguagem Alpha utiliza-se de informação extraída de fontes distintas, tais como a árvore de sintaxe abstrata (AST), o rastreamento da execução do programa, o *heap* e a definição de tipos estáticos para montagem de modelos conceituais que visam a diminuição do grau de acoplamento entre o código orientado por aspectos e o programa base [OMB05]. Os conjuntos de junção em Alpha são relações lógicas, escritas em Prolog, que atuam sobre os modelos conceituais gerados. O principal objetivo é determinar "quando" (sobre quais condições), e não "onde" (estrutura léxica), adendos devem ser executados.

Os autores apresentam uma análise de robustez que compara diferentes formas de implementação para um mesmo conjunto de junção no sistema clássico do Editor de Figuras. A comparação utilizou conjuntos de junção definidos com as seguintes abordagens: (1) enumerações; (2) expressões regulares; (3) aproximação do fluxo de controle (predicado `pcflow`); (4) fluxo de controle em tempo de execução (predicado `cflow`) e (5) acessibilidade de objetos no fluxo de execução (predicado `cflowreach`).

As duas primeiras abordagens geram conjuntos de junção que são semelhantes aos conjuntos de junção tradicionais de AspectJ. Dessa forma, as implementações que utilizam enumerações e expressões regulares foram consideradas as menos robustas no estudo realizado.

A terceira abordagem utiliza o predicado lógico `pcflow` para tentar identificar os campos que serão lidos durante o fluxo de controle do método `Display.drawAll()`, que é responsável por desenhar as figuras. A idéia é fazer com que apenas as atualizações desses campos sejam capturadas pelo conjunto de junção. Apesar da abordagem com `pcflow`

ser mais robusta do que as duas primeiras, ela só utiliza informações estáticas, fazendo com que a identificação dos campos não seja precisa.

A quarta abordagem diferencia-se por fazer, em tempo de execução, a seleção dos campos que serão lidos pelo método `Display.drawAll()`. Isso faz com que a abordagem que utiliza o predicado `cflow` seja mais robusta do que a anterior, que é inteiramente baseada na AST do programa.

A quinta e última abordagem é mais robusta em relação a todas as outras porque, segundo os autores, é a única resistente a alterações realizadas no grafo de objetos. Por exemplo, essas alterações podem fazer com que campos lidos pelo método `Display.drawAll()` sejam movidos para classes que não estão na hierarquia de `FigureElement` (interface que define as figuras). Nesse caso, a abordagem que utiliza `cflowreach` é robusta porque permite acessibilidade a qualquer objeto que seja alcançável, em tempo de execução, a partir do grafo de objetos de `FigureElement`.

O núcleo OO da linguagem Alpha baseia-se em L2, que é uma linguagem orientada por objetos mais simples, porém similar à linguagem Java. Classes implementadas em Alpha podem definir, além de campos e métodos, conjuntos de junção que podem ser associados aos seus respectivos adendos. Na seção de trabalhos futuros, os autores afirmam que pretendem adaptar sua tecnologia a uma linguagem de programação mais conhecida.

2.7.3 Outras soluções baseadas em modelos

O trabalho desenvolvido por Cazzola et al. apresenta uma proposta semelhante à anterior, porém utilizando uma linguagem baseada em UML para montagem de diagramas de seqüência e atividades sobre os quais serão definidos os pontos de junção da aplicação [CPA06]. Os aspectos devem conter a declaração dos padrões de pontos de junção que serão utilizados para execução dos adendos. Os autores propõem ainda uma nova linguagem para implementação de conjuntos de junção, denominada *join point pattern specification language*, que seleciona pontos de junção utilizando o modelo definido para o código base. O nível de abstração fornecido pelo modelo permite que os padrões dos pontos de junção sejam norteados pelo comportamento da aplicação, e não mais pela sua estrutura.

O trabalho desenvolvido em [CKAA06] apresenta uma solução, denominada *model-based aspects*, para resolver o problema dos adendos frágeis. A fragilidade dos adendos é causada pela maneira com que eles acessam informações do programa base, o que é diferente da fragilidade dos conjuntos de junção, que está associada à estrutura e ao fluxo de execução do programa. A idéia é retirar o acoplamento existente entre o código dos

adendos e o programa base introduzindo uma camada abstrata intermediária, que descreve o domínio da aplicação. Essa camada é responsável pelo mapeamento dos objetos do programa base e pela criação de adaptadores que serão utilizadas no código dos aspectos. O *framework* `SetPoint`, criado inicialmente para tratar o problema de fragilidade dos conjuntos de junção [AC04], foi estendido para tratar o problema dos adendos frágeis.

2.8 Introdução de anotações usando aspectos

O uso de anotações em linguagens orientadas por aspectos aumenta o poder de expressão dos conjuntos de junção. Anotações tornam explícitas características semânticas das aplicações. Informações semânticas podem ser utilizadas para escrever conjuntos de junção que são menos frágeis do que aqueles baseados em convenções de nomes ou padrões estruturais [HNBA06]. O trabalho desenvolvido por Havinga et al. analisa os efeitos da utilização de aspectos para introdução de anotações no código base e propõe uma solução para o problema da interdependência entre anotações na linguagem `Compose*`.

O problema da interdependência entre anotações introduzidas no código base ocorre quando a linguagem orientada por aspectos não possui um mecanismo capaz de realizar essa tarefa na ordem desejada. Suponha o seguinte exemplo, em AspectJ, de anotações interdependentes:

```
declare @field: (@TransientClass *) * :@TransientField();
```

Esse exemplo determina que todos os campos de classes que possuem a anotação `@TransientClass` devem ser anotados com `@TransientField`. Nesse caso, espera-se que as classes sejam anotadas primeiro com `@TransientClass`, caso contrário, a introdução da anotação `@TransientField` não terá nenhum efeito. No entanto, se a anotação `@TransientClass` também for introduzida via aspecto, não há como especificar qual das duas será adicionada primeiro no código base.

A solução proposta por Havinga et al. para esse problema é a utilização de uma extensão da linguagem `Compose*` para introdução de anotações. `Compose*` é baseada no conceito de filtros de composição (*Composition Filters*). Mais especificamente, conjuntos de junção, chamados "seletores" (*selectors*), são escritos em Prolog, usando um conjunto de predicados baseados na estrutura do programa. Os filtros de composição são responsáveis pela introdução de anotações em elementos selecionados do programa base. Para superar o problema da interdependência entre as anotações, é necessário analisar todas as possibilidades de ordenação entre os predicados das anotações que estão para serem inseridas. Todas as combinações devem ser tentadas. Os resultados obtidos com

cada tentativa são analisados a fim de se decidir quais anotações devem ser introduzidas primeiro.

2.9 Linguagem para definição de regras de projeto

A definição de regras de projeto para sistemas orientados por aspectos é uma alternativa para reduzir os problemas de fragilidade dos conjuntos de junção. Contudo, a utilização de linguagens como AspectJ para a criação de tais regras tem sido alvo de questionamentos [DNBS07]. Como ocorre no caso das XPIs descritas em [GSS+06], regras de projeto implementadas em AspectJ resultam em aspectos complexos, difíceis de compreender e manter. Por esta razão, Dósea et al. propõem a criação de uma linguagem para especificação de regras de projeto com o objetivo de minimizar os problemas de fragilidade dos conjuntos de junção de uma maneira mais simples e intuitiva do que por meio de AspectJ.

A linguagem proposta permite a especificação de regras estruturais para classes, interfaces e aspectos, possibilitando também a definição de regras comportamentais. Regras estruturais baseiam-se em informações estáticas presentes no código fonte, como por exemplo, assinaturas de métodos. Já regras comportamentais estão relacionadas com condições impostas aos elementos sobre os quais elas atuam. As regras comportamentais são válidas tanto para métodos quanto para adendos. A Tabela 2.1 apresenta as regras comportamentais definidas na linguagem. A sintaxe é semelhante à de AspectJ. Nas regras cujos nomes começam com letra "x", garante-se que as mesmas serão seguidas apenas no escopo definido. A regra `xcall`, por exemplo, garante que o método será chamado exclusivamente dentro do escopo no qual a regra foi definida. Assim, a chamada do método em outro local implica em uma quebra da regra.

Regra	Descrição
`call(método)`	Deve existir uma chamada do método no escopo
`xcall(método)`	Deve existir uma chamada do método exclusivamente no escopo
`set(campo)`	O campo deve ser alterado no escopo
`xset(campo)`	O campo deve ser alterado exclusivamente no escopo
`get(campo)`	O campo deve ser lido no escopo
`xget(campo)`	O campo deve ser lido exclusivamente no escopo

Tabela 2.1: Regras comportamentais definidas na linguagem

A Figura 2.3 apresenta uma possível especificação de uma regra de projeto para o exemplo do Editor de Figuras mostrado na Seção 1.1.1. A regra `DRUpdateFigure` estabelece que é necessária a existência de uma interface `Figure` contendo pelo menos o

```
 1: dr DRUpdateFigure {
 2:    interface Figure {
 3:      void moveBy(int dx, int dy);
 4:    }
 5:
 6:    class Point implements Figure {
 7:       void moveBy(int dx, int dy) {
 8:          xset(Figure+.*);
 9:       }
10:    }
11:
12:    aspect UpdateSignaling {
13:       pointcut change(): execution( void Figure.moveBy(int, int) );
14:
15:       after() returning: change() {
16:          xcall( <update.method> );
17:       }
18:    }
19: }
```

Figura 2.3: Exemplo de regra de projeto

método `moveBy` (linhas 2-4). Além disso, deve haver uma classe `Point` que implementa `Figure`, e por conseqüência o método `moveBy`, que é o único método que pode realizar alterações nos campos (linhas 6-10). A regra especifica ainda a existência de um aspecto `UpdateSignaling`, com o conjunto de junção `change`, que captura execuções do método `moveBy` de classes que implementam `Figure` (linhas 12-13). A linguagem permite a utilização de parâmetros de configuração para regras de projeto, como é o caso do parâmetro `<update.method>`, que representa o método que será executado, exclusivamente, pelo adendo do tipo `after returning` que está associado ao conjunto de junção `change` (linhas 15-17).

Antes de utilizar uma regra de projeto, é necessário associar seus elementos (classes, interfaces, aspectos, etc) com os elementos correspondentes na aplicação que irá implementá-la. É nesse momento também que são informados os valores dos parâmetros definidos no escopo da regra, como é o caso do parâmetro `<update.method>`, que precisa ser associado a algum método real da aplicação durante a configuração. Essa configuração permite ainda que a mesma regra de projeto possa ser reutilizada por módulos diferentes. Os nomes dos elementos da aplicação podem ser iguais ou diferentes aos nomes dos seus

correspondentes na regra de projeto. Por exemplo, ao configurar a regra definida na Figura 2.3, pode-se associar a classe `Point` da regra a uma classe similar que possui outro nome no programa base.

A linguagem para definição das regras de projeto ainda está em fase experimental. Assim, não se sabe se o poder de expressão da linguagem será suficiente para resolver os diversos tipos de problemas decorrentes da fragilidade dos conjuntos de junção. Além disso, considera-se que em sua forma atual a linguagem proposta requer que programadores repliquem nas regras de projeto diversas estruturas do programa base. Ou seja, as regras propostas tendem a requerer especificações detalhadas e extensas.

2.10 Comentários finais

Inicialmente, apresentou-se neste capítulo uma breve introdução ao conceito de anotações, além dos principais comandos e estruturas da linguagem AspectJ. Em seguida, foi feita uma revisão das principais pesquisas e trabalhos relacionados com o problema de fragilidade de conjuntos de junção em linguagens orientadas por aspectos. No entanto, nenhuma das soluções apresentadas se mostra capaz de resolver de forma completa o problema de fragilidade de conjuntos de junção.

A análise de delta de conjuntos de junção compara versões diferentes de sistemas orientados por aspectos para criar uma visão global dos impactos decorrentes de alterações realizadas no código fonte do programa. Entretanto, a análise das diferenças encontradas não permite a identificação de pontos de junção que deveriam, mas que efetivamente não são capturados por conjuntos de junção porque não seguem as convenções de nomes esperadas.

Aspect-aware interfaces foram originalmente propostas para serem utilizadas como documentação de apoio a raciocínio modular. A análise dessas interfaces pode auxiliar na detecção de falhas na captura e na detecção de capturas acidentais de pontos de junção. Entretanto, a geração de tais interfaces é manual. Isto é, não existe menção na literatura a ferramentas que automatizem a geração de AAIs. Além disso, também não é possível usar AAIs para identificar pontos de junção que deveriam, mas que efetivamente não são capturados, da mesma forma que ocorre com a análise de delta de conjuntos de junção.

Open modules procuram restringir o poder de ação dos aspectos fazendo com que módulos do sistema base exponham quais pontos de junção podem ser capturados. Contudo, a transferência do código que define os conjuntos de junção para o programa base não os torna menos frágeis. Porém, ao implementar os dois códigos que precisam ser alterados em um único módulo, a idéia é que o desenvolvedor do módulo seja responsável

pelos seus requisitos não-transversais e também pela definição dos conjuntos de junção exportados por ele.

Crosscutting interfaces ou XPIs implementam regras de projeto que devem ser seguidas pelo código base da aplicação e sobre as quais o código dos aspectos deve se basear. Em sua mais atual proposta, XPIs são implementadas em AspectJ, o que limita o poder de definição das regras de projeto, fazendo com que elas não sejam capazes de lidar com todos os problemas de fragilidade de conjuntos de junção.

Conjuntos de junção baseados em modelos propõem uma alteração no nível de abstração para declaração de conjuntos de junção, os quais passam a utilizar modelos conceituais, eliminando a dependência estrutural de aspectos em relação ao código fonte do programa base. Não se sabe, entretanto, se o poder de definição das regras associadas aos modelos conceituais é suficiente para atender a todos os tipos de problemas de conjuntos de junção frágeis. Além disso, tais propostas requerem que o modelo usado esteja sempre sincronizado com o código fonte da aplicação, o que não ocorre com freqüência em projetos de desenvolvimento de software.

Por fim, apresentou-se a proposta de uma linguagem que objetiva tornar mais simples e intuitiva a implementação de regras de projeto, evitando assim os problemas detectados com o uso de AspectJ para implementação de tais regras. Entretanto, o projeto dessa linguagem ainda está em fase experimental, não sendo, portanto, possível avaliar seu efetivo poder de expressão.

Capítulo 3

Geração de Aspect-Aware Interfaces

3.1 Visão geral

Conforme mencionado na Seção 2.4, *aspect-aware interfaces* auxiliam desenvolvedores a raciocinar localmente sobre módulos de sistemas orientados por aspectos. Nesta dissertação, propõe-se a geração automática de AAIs durante o desenvolvimento de uma aplicação orientada por aspectos. Para tal, foi desenvolvida uma ferramenta que, a cada compilação, irá gerar AAIs e realizar uma comparação da versão atual com a anterior, registrando em um arquivo as diferenças encontradas. Uma vez que AAIs possuem informações sobre todas as classes, campos e métodos impactados por conjuntos de junção, elas podem ser utilizadas para identificar capturas adicionais e falhas na captura de pontos de junção.

A Figura 3.1 apresenta as AAIs para o exemplo do Editor de Figuras da forma como elas são geradas pela ferramenta proposta. Conforme discutido na Seção 2.4, estas AAIs listam todos os métodos das classes do programa base que são capturados por algum conjunto de junção, bem como a relação de adendos que são disparados pela execução desses métodos. Comparando-se as AAIs da Figura 3.1 com as AAIs da forma como foram originalmente propostas por Kiczales e Mezini, na Figura 2.1, percebe-se que as AAIs geradas pela ferramenta proposta nesta dissertação não incluem as informações de AAIs para aspectos. AAIs associadas a aspectos possuem uma entrada para cada um de seus adendos. Nessas entradas, além do tipo do adendo, informam-se também os pontos de junção do programa base sobre o qual ele se aplica. Ou seja, AAIs associadas a aspectos disponibilizam a mesma informação das AAIs associadas a classes, porém de forma invertida, conforme pode ser visto na AAI associada ao aspecto `UpdateSignaling`, apresentada anteriormente na Figura 2.1. Como o foco principal da geração de AAIs proposta nesta dissertação é auxiliar no diagnóstico de falhas na captura e capturas acidentais de pontos

```
interface Figure
  void moveBy(int, int): after returning UpdateSignaling.change();

Point implements Figure
  int x;
  int y;
  void setY(int): after returning UpdateSignaling.change();
  void setX(int): after returning UpdateSignaling.change();
  void moveBy(int, int): after returning UpdateSignaling.change();

Line implements Figure
  Point P1;
  Point P2;
  void setP1(Point): after returning UpdateSignaling.change();
  void setP2(Point): after returning UpdateSignaling.change();
  void moveBy(int,int): after returning UpdateSignaling.change();
```

Figura 3.1: Exemplos de AAIs geradas

de junção, decorrentes de manutenções realizadas no programa base, AAIs associadas a aspectos não são geradas.

O restante deste capítulo está organizado conforme descrito a seguir. A Seção 3.2 descreve com mais detalhes a implementação da ferramenta utilizada nos processos de geração e atualização de AAIs. Na Seção 3.3, mostra-se um estudo de caso que descreve o uso de AAIs para detectar capturas adicionais e falhas na captura de pontos de junção. Por fim, a Seção 3.4 apresenta os comentários finais e encerra o capítulo.

3.2 Geração de AAIs

Propõe-se nesta dissertação uma ferramenta para geração incremental das AAIs de uma aplicação. Esta ferramenta foi implementada como um *plug-in* do ambiente Eclipse. À medida que o sistema base sofre manutenções ou evoluções, as informações sobre pontos de junção e adendos contidas nas AAIs são atualizadas automaticamente. Com isso, criam-se as condições para que o desenvolvedor da aplicação seja notificado das duas situações definidoras do problema dos conjuntos de junção frágeis de AspectJ: capturas acidentais de pontos de junção e falhas na captura de pontos de junção, conforme mencionado na Seção 1.1.1.

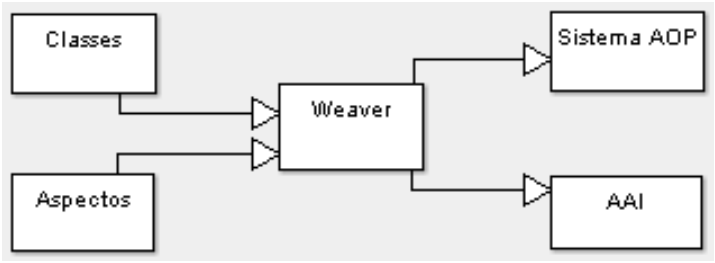

Figura 3.2: Geração de AAIs

Mais especificamente, a proposta é que a geração de AAIs seja incorporada ao *weaver* de um ambiente de desenvolvimento orientado por aspectos, conforme ilustrado na Figura 3.2. Como o *weaver* realiza o espalhamento do código dos adendos em pontos de junção de um programa base, ele possui as informações necessárias para a geração das AAIs. Além disso, como o *weaver* é sempre chamado após a compilação das classes e aspectos de um sistema, garante-se que novas versões de AAIs serão sempre geradas quando ocorrerem alterações no programa base, o que é essencial para tratar a fragilidade dos conjuntos de junção de AspectJ.

A ferramenta desenvolvida se utiliza de classes e interfaces exportadas pelo pacote AJDT, que permitem acesso a informações transversais de um projeto Eclipse. Por exemplo, a interface `AJBuilder.IAdviceChangedListener` do AJDT permite que a ferramenta registre um *listener* que será chamado toda vez que um projeto for compilado no ambiente. O *listener* registrado se encarrega então de gerar uma nova AAI a cada compilação do projeto. Já a classe `AJModel` fornece informações sobre classes e interfaces do código base do sistema que possuem adendos associados a seus métodos. Os campos das classes que aparecem nas AAIs, mas cuja leitura e atualização não é capturada por nenhum conjunto de junção, são obtidos por meio de reflexão.

A ferramenta proposta recupera e organiza as informações obtidas em uma estrutura de dados própria, representada na Figura 3.3. Basicamente, essa estrutura consiste de uma tabela *hash* cujas chaves são nomes de classes do programa base que possuem adendos associados. A entrada de uma classe nesta tabela armazena uma lista com informações sobre os campos e métodos desta classe que podem disparar a execução de adendos. Cada método ou campo é então associado a uma outra lista, contendo informações sobre os adendos disparados pelo mesmo. Para cada adendo, são armazenadas informações sobre seu tipo (`after`, `before` ou `around`), nome do conjunto de junção usado para definir o adendo e nome do aspecto no qual o adendo foi implementado.

Na Figura 3.3, a classe `Point` está representada na tabela que contém os nomes das classes que estão associadas à execução de algum adendo. O elemento da tabela *hash* que

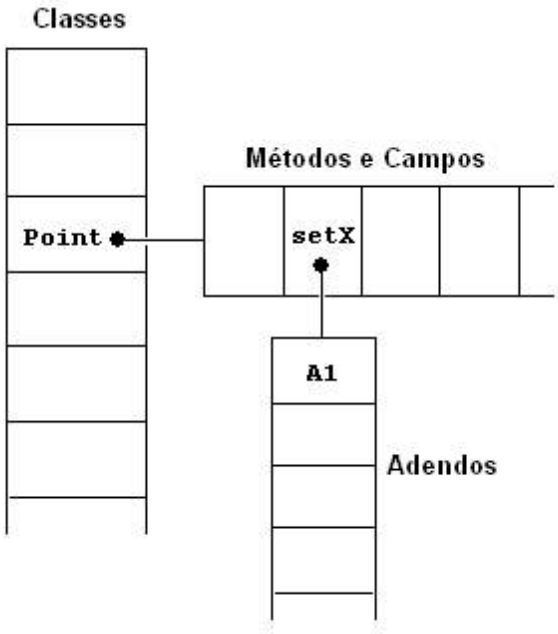

Figura 3.3: Estrutura de dados usada para armazenar AAIs logicamente

está associado à classe `Point` referencia uma estrutura de dados que armazena os nomes dos métodos dessa classe que podem disparar algum adendo, como é o caso do método `setX`. O elemento da estrutura que contém o método `setX` possui uma referência para uma lista de adendos que podem ser disparados quando o método for acionado. Como os adendos não possuem nomes, o adendo utilizado como exemplo na figura recebeu o nome de `A1`. Informações necessárias para identificar os adendos (tipo, nome do aspecto e nome do conjunto de junção) ficam armazenadas na lista que contém os adendos. Em resumo, a estrutura de dados mostrada na Figura 3.3 é usada internamente para representar as AAIs. Percorrendo-se esta estrutura, é possível gerar uma versão textual das AAIs.

Sempre que ocorrer uma nova compilação do projeto, a ferramenta recupera os dados das últimas AAIs que estavam armazenadas em disco e compara com as AAIs geradas a partir da nova versão do sistema. As diferenças encontradas são atualizadas no arquivo que contém as AAIs, de forma que o mesmo se mantenha sempre sincronizado com o programa base. Por exemplo, suponha a seguinte manutenção sobre o programa base do Editor de Figuras: o método `setX` é renomeado para `alteraX`. Após a compilação dessa alteração, a ferramenta irá gerar uma nova versão da AAI para a classe `Point`, na qual o método `setX` não estará presente. Ao comparar a nova versão da AAI com a versão anterior, a ferramenta identifica a remoção do método `setX` e grava esta informação em

```
Seja I a nova versão da AAI gerada
Seja I' a versão anterior da AAI

Para cada classe C de I faça
  Para cada método M de C faça
    Para cada adendo A associado a M faça
      Se A não está em I' então
        Registrar que M passou a disparar a execução de A

Para cada classe C de I' faça
  Para cada método M de C faça
    Para cada adendo A associado a M faça
      Se A não está em I então
        Registrar que M deixou de disparar a execução de A
```

Figura 3.4: Representação de parte do algoritmo que compara as AAIs

um arquivo de *log*. Sabendo-se que o conjunto de junção `change` captura chamadas a métodos de classes que implementam a interface `Figure` e cujo nome começa com `set` – `Figure+.set*()` – é importante notar que, no exemplo anterior, se o método `setX` fosse renomeado para `setZ`, duas entradas seriam incluídas no arquivo de *log*: uma informando sobre a retirada de `setX` e outra informando sobre a adição de `setZ` na AAI da classe `Point`. A Figura 3.4 representa um esboço de parte do algoritmo utilizado para identificar diferenças entre AAIs de versões diferentes do sistema.

Novas versões de uma AAI – contendo informações atualizadas sobre pontos de junção de um programa base – podem ser criadas tanto em decorrência de manutenções realizadas nos aspectos como nas classes desse programa. Manutenções podem levar à inclusão de novas informações em uma AAI (quando a manutenção faz com que um novo ponto de junção do programa seja capturado) ou à remoção de informações contidas em uma AAI (quando a manutenção faz com que um ponto de junção anteriormente capturado não seja mais). Em ambos os casos, informações relativas ao que foi alterado na versão atual das AAIs em relação à versão anterior são gravadas em um arquivo de *log*. O *log* gerado pode ser utilizado pelos desenvolvedores para controlar a evolução de sistemas orientados por aspectos, uma vez que auxilia na identificação dos problemas de falha na captura e captura acidental dos pontos de junção.

No caso de pontos de junção cuja captura somente é decidida em tempo de execução, como aqueles capturados por meio de `cflow` ou `cflowbelow`, adota-se uma solução con-

servadora na geração das respectivas AAIs. Basicamente, em um conjunto de junção definido como PC_{static} && $PC_{dynamic}$, onde PC_{static} e $PC_{dynamic}$ denotam respectivamente descritores de conjuntos de junção estáticos e dinâmicos, assume-se na geração de AAIs que $PC_{dynamic}$ é sempre verdade. Com isso, se existe a possibilidade de um adendo ser disparado em função da chamada de um método, este adendo é incluído na AAI deste método.

A idéia de comparar versões diferentes de um sistema para detectar alterações nos pontos de junção capturados também é utilizada nos sistemas PCDiff e Eclipse AJDT Crosscutting Comparison, conforme relatado na Seção 2.3. No entanto, a geração de AAIs proposta nesta dissertação se diferencia das soluções anteriores principalmente por incorporar essa geração ao processo de *weaving*, fazendo com que a identificação das alterações nos pontos de junção que são capturados seja feita automaticamente após cada compilação de um sistema orientado por aspectos. Além disso, AAIs são componentes importantes para a criação de aspectos anotadores, conforme será apresentado no Capítulo 4.

3.3 Estudo de caso

Nesta seção, apresenta-se um estudo de caso que ilustra alguns dos principais impactos em aspectos decorrentes de manutenções no código base de um sistema. Suponha que as seguintes manutenções sejam realizadas na classe `Point` do sistema do Editor de Figuras, apresentado na Seção 1.1.1:

1. Acrescenta-se o método `setDate`;
2. Renomeia-se o método `setX` para `changeX`;
3. Acrescenta-se o método `setColor` (e o campo `color`);
4. Remove-se o método `setColor` (e o campo `color`);
5. Acrescenta-se o método `changeColor` (e o campo `color`);

A Tabela 3.1 resume o comportamento da ferramenta após a compilação de cada uma das manutenções propostas. Este comportamento é descrito com detalhes a seguir:

- **Manutenção 1**: Esta manutenção dá origem a uma captura acidental de um ponto de junção, pois o campo `Date` não é um atributo visual de `Point` e, portanto, o método `setDate` não deveria ser capturado pelo conjunto de junção `change`. A

ferramenta detecta a captura do novo ponto de junção e insere essa informação no *log*. Nesse caso, cabe ao desenvolvedor analisar o arquivo de *log* e refazer a manutenção, de forma a restaurar a semântica original dos conjuntos de junção afetados. Por exemplo, ele pode optar por chamar o método de `changeDate`.

- **Manutenção 2**: Esta manutenção dá origem a uma falha na captura de um ponto de junção, uma vez que o campo `x` é um atributo visual da figura e o método `changeX` não será capturado pelo conjunto de junção `change`. A ferramenta detecta a remoção de `setX` e insere essa informação no *log*. Nesse caso, também cabe ao desenvolvedor refazer a manutenção, por exemplo, mantendo o nome original, para garantir que a figura será redesenhada quando o campo `Point.x` for alterado.

- **Manutenção 3**: Esta manutenção representa uma ampliação desejável do grau de quantificação de um conjunto de junção, pois o campo `color` é um atributo visual e o método `setColor` é automaticamente capturado pelo conjunto de junção `change`. A ferramenta detecta a captura do novo ponto de junção e insere essa informação no *log*. O desenvolvedor analisa o arquivo de *log* e certifica-se de que a alteração está correta. A Figura 3.5 apresenta a entrada gravada no arquivo de *log* quando o método `setColor` é adicionado.

- **Manutenção 4**: Esta manutenção representa uma redução desejável do grau de quantificação de um conjunto de junção, pois o método `setColor` é excluído do programa base. A ferramenta detecta a ausência do ponto de junção e insere essa informação no *log*. O desenvolvedor analisa o arquivo de *log* e certifica-se de que a alteração está correta.

- **Manutenção 5**: Após esta manutenção, execuções do método `changeColor` deveriam ser capturadas pelo conjunto de junção `change`, pois representam alterações na cor de um objeto, mas efetivamente não são capturados porque o nome do método adicionado não segue as convenções de nomes pressupostas por este conjunto de junção. Segundo estas convenções, o método deveria se chamar `setColor` e não `changeColor`. Ou seja, a ferramenta proposta não é capaz de detectar a inserção de novos pontos de junção no programa base que deveriam mas que efetivamente não são capturados por um determinado conjunto de junção.

	Manutenção	Situação	Ferramenta
1	+ `setDate`	Captura acidental	Detecta
2	`setX` → `changeX`	Falha captura	Detecta
3	+ `setColor`	Captura adicional	Detecta
4	- `setColor`	Eliminação captura	Detecta
5	+ `changeColor`	Captura pendente	Não detecta

Tabela 3.1: Exemplos de manutenção no código base e comportamento da solução proposta

Log de alterações nas AAIs.txt

```
04/03/2008 07:49:00 - Method 'public void setColor(Color color)' of class 'Point'
    has been captured by pointcut 'change' of aspect 'UpdateSignaling'.
...
```

Figura 3.5: Log que registra alterações nas AAIs

3.4 Comentários finais

Considera-se que a geração de AAIs, conforme projetada e implementada nesta dissertação, apresenta os seguintes benefícios:

- O arquivo de *log* gerado possibilita que o desenvolvedor obtenha informações sobre capturas acidentais de pontos de junção e sobre falhas na captura de pontos de junção, quando de uma manutenção em um sistema orientado por aspectos.
- A solução é compatível com o atual modelo de captura de pontos de junção de AspectJ, isto é, a mesma pode ser usada em qualquer sistema orientado por aspectos que esteja sendo desenvolvido atualmente.
- A solução inclui uma ferramenta para geração de AAIs. Ressalte-se que AAIs, além de constituírem um instrumento útil para detecção e validação de fragilidades em conjuntos de junção, podem ser usadas também no apoio a raciocínio modular (isto é, a capacidade de se raciocinar sobre um módulo, entendendo suas funções e interfaces).

A solução, no entanto, não é capaz de detectar novos pontos de junção adicionados a um programa que não seguem as convenções de nome pressupostas por um aspecto (conforme é o caso do método `changeColor` do estudo de caso apresentado na Seção 3.3). No entanto, considera-se que esta deficiência é inerente ao modelo de captura de pontos de junção de AspectJ, o qual é baseado exclusivamente em nomes. Uma possível

solução para este problema consiste em adicionar informações semânticas no modelo de captura de pontos de junção. O capítulo seguinte apresenta uma proposta para resolver esse problema através do uso de anotações, que são introduzidas na aplicação com o objetivo de classificar os elementos do código base de acordo com sua função semântica, possibilitando a construção de conjuntos de junção mais robustos e com maior grau de clareza.

Capítulo 4

Aspectos Anotadores

4.1 Introdução

Conforme foi mencionado no final do Capítulo 3, a utilização de AAIs para detecção de falhas na captura e capturas acidentais de pontos de junção pode não apresentar um resultado satisfatório quando são conhecidas apenas características sintáticas dos pontos de junção. Para a construção de conjuntos de junção mais robustos, propõe-se nesta dissertação o uso de anotações para classificar os pontos de junção de acordo com sua função semântica. Os conjuntos de junção passam então a ser construídos com base nessas anotações, deixando de depender da estrutura sintática do programa base.

A utilização de anotações para diminuir a fragilidade dos conjuntos de junção já foi proposta em [KM05b, EA06, HNBA06]. No entanto, todas essas propostas apresentam os problemas listados a seguir:

- *Anotações representam interesses transversais.* Anotações em Java apresentam um comportamento transversal porque são normalmente invasivas e acopladas a regras de negócio específicas. Para evitar esse problema, propõe-se nesta dissertação a utilização de aspectos anotadores, que introduzem anotações no código base de forma não-invasiva. Além disso, propõe-se que aspectos anotadores sejam gerados de forma semi-automática, a partir do uso de uma linguagem declarativa.

- *Conjuntos de junção baseados em anotações são frágeis.* Tradicionalmente, conjuntos de junção baseados em anotações pressupõem que os desenvolvedores tenham anotado corretamente o programa base. Anotações adicionadas em locais incorretos, ou que não seguem as convenções de nomes esperadas, podem, silenciosamente, desabilitar pontos de junção que deveriam ser capturados pelos conjuntos de junção.

Para evitar esse problema, a solução proposta requer que os responsáveis pela manutenção do sistema informem, para cada elemento que esteja no escopo léxico de elementos que poderiam ser anotados, quais irão efetivamente receber a anotação.

- *Anotações violam a característica de obliviousness de sistemas orientados por aspectos.* Embora seja considerada uma característica inerente a sistemas orientados por aspectos, o conceito de *obliviousness* tem sido, recentemente, alvo de diversos questionamentos. Por exemplo, Sullivan et al. demonstraram que a utilização de *obliviousness* conduz à criação de conjuntos de junção que são difíceis de desenvolver, manter e entender [GSS+06, SGS+05]. Por esta razão, eles recomendam que os desenvolvedores preparem o programa base para que o mesmo seja interceptado por aspectos. Na solução proposta nesta dissertação, os desenvolvedores têm que decidir se os elementos selecionados no programa base devem ou não ser anotados com anotações que foram pré-definidas. Dessa forma, a solução proposta também descarta a forma mais pura de *obliviousness*, pois utiliza as anotações como se fossem contratos, ou regras de projeto, entre os aspectos e o programa base e expõe essas anotações para os desenvolvedores.

O restante deste capítulo está organizado conforme descrito a seguir. A Seção 4.2 apresenta os principais componentes da solução baseada em aspectos anotadores, incluindo a linguagem para declaração de anotações, as interfaces utilizadas para representar os pontos de junção que foram anotados e os próprios aspectos anotadores, que introduzem as anotações no código base de forma não invasiva. A Seção 4.3 apresenta exemplos de utilização da solução proposta na construção de conjuntos de junção em dois sistemas reais orientados por aspectos. A Seção 4.4 descreve um estudo de caso no qual é avaliada a robustez da solução proposta diante de uma série de possíveis alterações que foram simuladas no sistema exemplo do Editor de Figuras. A Seção 4.5 descreve a implementação de uma ferramenta para viabilizar a solução proposta. A Seção 4.6 apresenta uma discussão sobre os resultados alcançados com a solução proposta, pontuando suas vantagens e suas limitações. Por fim, a Seção 4.7 apresenta os comentários finais e encerra o capítulo.

4.2 Solução proposta

A Figura 4.1 apresenta os principais componentes utilizados na definição e na introdução de anotações no programa base. Inicialmente, a solução proposta requer que desenvolvedores declarem as anotações que poderão ser utilizadas na implementação dos conjuntos de junção. Cada anotação irá atuar sobre o escopo de uma classe ou de um

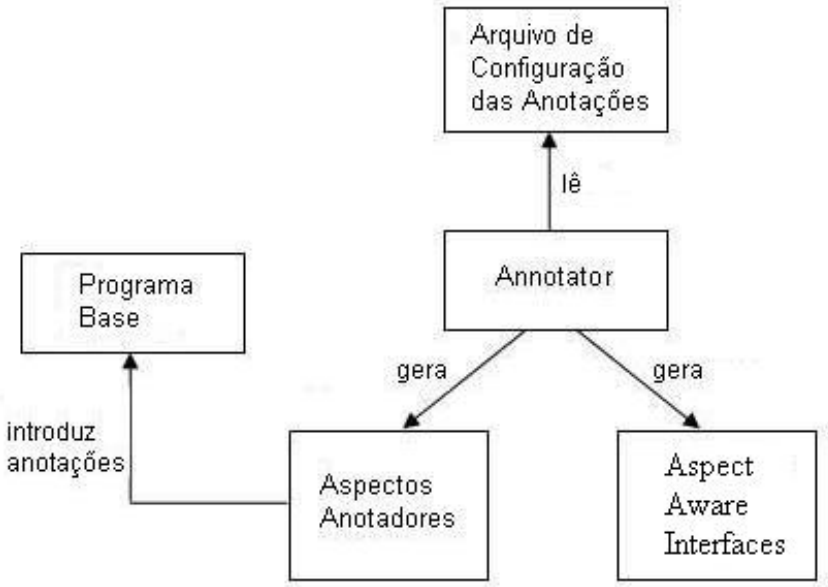

Figura 4.1: Componentes utilizados na solução proposta

grupo de classes, de acordo com a especificação adotada pelo desenvolvedor em um arquivo de configuração específico para definição de anotações. Esse arquivo é então lido por uma ferramenta, denominada Annotator, que irá identificar os elementos do programa base que são candidatos a se tornarem pontos de junção e, em seguida, questionar o desenvolvedor sobre quais desses elementos devem realmente receber anotações. O Annotator, que é uma extensão da ferramenta utilizada para geração de AAIs, descrita no capítulo anterior, é acionado sempre que o processo de *weaving* for executado. Assim que o Annotator receber o retorno do desenvolvedor sobre quais elementos devem ser anotados, os dois últimos componentes da solução proposta são gerados: aspectos anotadores e *aspect-aware interfaces*. Os aspectos anotadores adicionam as anotações no programa base enquanto que as *aspect-aware interfaces* passam a receber também informações relativas às anotações que foram introduzidas. Todos os componentes envolvidos na solução proposta são descritos com detalhes no restante desta seção.

4.2.1 Linguagem para declaração de anotações

As anotações utilizadas na solução proposta devem ser definidas em um arquivo próprio de configuração, por meio de uma linguagem declarativa. Programas escritos nessa linguagem são definidos de acordo com a gramática apresentada na Figura 4.2. Nessa gramática, símbolos não-terminais são escritos entre < e >, e símbolos terminais são escritos em ne-

<AnnotationDeclarationFile> → { **annotation** <AnnotationSection> **end** }

<AnnotationSection> → <Name> <Question> <Scope> <Target>

<Name> → **name** = <String>
<Question> → **question** = <String>
<Target> → **target** = <TargetType>
<Scope> → **scope** = <Type> {, <Type> } | <Package> {, <Package> }

<TargetType> → **fields** | **methods** [**&&** [**!**] <MethQualifier>]
| **classes** [**&&** [**!**] <ClassQualifier>]

<MethQualifier> → **getters** | **setters** | **call(** <Method> **)**
| **declare(** <Type> **)** | **catch(** <Type> **)**

<ClassQualifier> → **usedAs(** <TypeUsedAs> **)** | **field(** <Type> **)**

<TypeUsedAs> → **field** | **param** | **return** | **exception**

<Type> → <String>
<Package> → <String>
<Method> → <String>

Figura 4.2: Gramática para definição das anotações

grito. Na notação utilizada, `{A}` representa uma ou mais repetições de `A`, enquanto que `[A]` indica que `A` é opcional. O símbolo não-terminal `<String>` não foi definido. Porém, ele denota uma seqüência de caracteres.

A declaração de uma anotação inclui as seguintes informações:

- O nome da anotação (entrada `name` de `AnotationSection`).
- O alvo da anotação (entrada `target` de `AnotationSection`). Anotações podem estar associadas a campos, métodos ou classes.
- O escopo de atuação da anotação (entrada `scope` de `AnotationSection`). Representa um grupo de classes ou pacotes que são considerados na procura por possíveis alvos da anotação.
- A pergunta que será apresentada aos desenvolvedores para que eles decidam se determinado elemento alvo deve ser anotado ou não (entrada `question` de `AnotationSection`).

No caso de anotações associadas a métodos ou classes, filtros podem ser utilizados para restringir o número de elementos candidatos a serem anotados, dentro do escopo considerado. É possível utilizar o símbolo de negação "`!`" para indicar que o filtro desejado representa a não satisfação de determinada condição. Os filtros que podem ser utilizados para métodos, denominados `MethQualifiers`, são os seguintes:

- `getters`: métodos que não possuem parâmetros e são utilizados apenas para retornar o valor de um campo da classe.
- `setters`: métodos que possuem exatamente um parâmetro e são utilizados apenas para alterar o valor de um campo da classe.
- `call(Method)`: métodos que realizam pelo menos uma chamada ao método passado como parâmetro.
- `declare(Type)`: métodos que declaram localmente pelo menos uma variável do tipo informado no parâmetro.
- `catch(Type)`: métodos que capturam exceções do tipo passado como parâmetro.

Os filtros que podem ser utilizados para classes, denominados `ClassQualifiers`, são os seguintes:

- `usedAs(field)`: classes que são utilizadas para declarar campos no escopo de uma outra classe.
- `usedAs(param)`: classes que são utilizadas como parâmetro de métodos.
- `usedAs(return)`: classes que são utilizadas como retorno de métodos.
- `usedAs(exception)`: classes que representam exceções retornadas por métodos.
- `field(Type)`: classes que possuem pelo menos um campo do tipo passado como parâmetro.

Depois de definida uma anotação, a ferramenta Annotator monitora o código base, a cada compilação, procurando por elementos do programa que se encaixam na definição criada. Sempre que um elemento candidato a receber a anotação é encontrado, o Annotator irá utilizar a pergunta definida no arquivo de configuração da anotação para questionar o desenvolvedor se aquele elemento detectado deve ser anotado ou não. Se a resposta for positiva, o Annotator irá gerar o código que será responsável por introduzir a anotação para aquele elemento no código base. Esse código é gerado em aspectos, denominados

aspectos anotadores, fazendo com que as anotações sejam introduzidas durante o processo de *weaving* do sistema. Dessa forma, a anotação não aparece no código fonte do programa base, evitando espalhamento e entrelaçamento de código. Além disso, o fato de os desenvolvedores terem que confirmar quais elementos devem ser anotados é importante para evitar ocorrências de falhas na captura de pontos de junção, que geralmente acontecem em aplicações Java/AspectJ que utilizam anotações.

Para exemplificar a utilização de uma anotação na solução proposta, vamos considerar a seguinte regra de projeto para o exemplo do Editor de Figuras: *"Todo método que altera um atributo visual de uma figura deve disparar* `Display.update()`*"*. A maior dificuldade nesse caso é identificar quais métodos alteram atributos visuais e quais não alteram. No caso da classe `Point`, as coordenadas `x` e `y` são visuais, mas, se fosse utilizado um campo de nome `date` (que armazenaria a data de criação do ponto), esse campo não seria visual, pois não tem nenhuma relação com a imagem do objeto. Neste caso, como já foi discutido na Seção 3.3, o uso do método `setDate` para atualizar o campo `date` geraria uma captura acidental de ponto de junção.

O exemplo a seguir apresenta a definição de uma anotação que torna possível a classificação de quais métodos de classes que implementam a interface `Figure` alteram atributos visuais.

```
annotation
  name = @DisplayStateChange
  question = Does method %target change the state of the display?
  target = methods && (! getters)
  scope = Figure
end
```

A especificação da anotação `@DisplayStateChange` determina que os elementos candidatos a serem anotados são os métodos de classes que implementam a interface `Figure` e que não são `getters`. O Annotator usa essa informação para monitorar o programa base na busca por métodos que satisfaçam tais condições. A pergunta (`question`) presente na definição da anotação será apresentada aos desenvolvedores sempre que um ou mais métodos que atendam às condições impostas forem detectados. A variável pré-definida `%target` será substituída pela assinatura de cada método candidato que for encontrado. Na Figura 4.3, o exemplo da detecção do método `Point.setDate` é apresentado, bem como a pergunta realizada pela ferramenta aos desenvolvedores. Para cada pergunta, o desenvolvedor pode responder "sim" ou "não".

Assim que as perguntas são respondidas, as respostas são gravadas em um arquivo de dados. Isso evita que a mesma pergunta seja feita mais de uma vez. Para facilitar,

Figura 4.3: Pergunta sobre elemento candidato a receber uma anotação

respostas iguais podem ser dadas em bloco (ou seja, pode-se marcar mais de uma pergunta e selecionar a resposta uma única vez).

É importante observar que o Annotator monitora constantemente alterações feitas no código da aplicação procurando por alterações no conjunto de elementos que atendem às condições especificadas na definição das anotações. Por exemplo, suponha que um método qualquer `m1` não atenda às condições de uma determinada anotação `ann` no momento em que foi implementado. Isso porque a anotação `ann` só se aplica, por exemplo, a métodos que chamam um outro método `m2`. Se, durante a evolução do sistema, o método `m1` for alterado de forma que passe a chamar o método `m2`, então o Annotator irá identificá-lo como elemento candidato a receber a anotação `ann`. Da mesma forma, quando um método deixa de atender às condições impostas por determinada anotação, a informação relativa à remoção do ponto de junção será atualizada nos arquivos mantidos pela solução proposta. Entretanto, nos casos de eliminação dos pontos de junção, a ferramenta não faz nenhum questionamento aos desenvolvedores.

4.2.2 Novas *aspect-aware interfaces*

As *aspect-aware interfaces* geradas pelo Annotator correspondem a uma extensão das AAIs descritas no Capítulo 3, com a adição de informações sobre os elementos do código base que receberam anotações. O trecho de código a seguir ilustra métodos presentes nas AAIs das classes `Point` e `Line` que receberam a anotação `@DisplayStateChange`.

```
class Point {
  @DisplayStateChange public void setY(int);
  @DisplayStateChange public void setX(int);
}
class Line {
  @DisplayStateChange public void setP1(Point);
  @DisplayStateChange public void setP2(Point);
}
```

Os desenvolvedores têm a opção de editar o arquivo que contém as *aspect-aware interfaces* para inserir ou remover anotações diretamente. Intervenções desse tipo são necessárias quando o desenvolvedor percebe que respondeu de forma incorreta a uma pergunta feita pelo Annotator sobre anotar ou não determinado elemento do programa base. A inclusão direta de uma anotação no arquivo corresponde a uma resposta "sim" dada a uma pergunta feita pelo Annotator, da mesma forma, a remoção de uma anotação corresponde a uma resposta "não". Conforme foi mencionado no final da Seção 4.2.1, as respostas dadas às perguntas feitas pelo Annotator são gravadas em um arquivo interno da ferramenta. Esse arquivo não é o mesmo que armazena as *aspect-aware interfaces*. E quando, por exemplo, o desenvolvedor responde "sim" a uma pergunta e depois edita o arquivo de interfaces e remove a anotação, o Annotator detecta, na próxima compilação, que o arquivo interno de dados contém a resposta "sim" e a *aspect-aware interface* não contém a anotação. Nesses casos, irá prevalecer a informação presente nas *aspect-aware interfaces*. Ainda assim, o arquivo de dados é necessário porque, ao analisar apenas as *aspect-aware interfaces*, não é possível diferenciar respostas negativas de respostas ainda não fornecidas.

4.2.3 Aspectos anotadores

Na solução proposta, para cada anotação especificada, existe um aspecto responsável por introduzir a anotação no programa base de forma não invasiva. Aspectos que desempenham essa função, denominados de aspectos anotadores, são gerados e atualizados de forma semi-automática pela ferramenta Annotator sempre que um novo elemento candidato é encontrado no escopo léxico da anotação. A geração é considerada semi-automática porque os desenvolvedores responsáveis pela aplicação devem confirmar quais dos elementos selecionados irão ser realmente anotados. Nos casos em que o desenvolvedor edita manualmente o arquivo que contém as *aspect-aware interfaces*, os aspectos anotadores também são atualizados.

O trecho de código a seguir apresenta o aspecto anotador gerado para a anotação `DisplayStateChange`, definida no exemplo da Seção 4.2.1:

```
package Annotator;
aspect DisplayStateChangeAnnotator {
  declare @method: public void Point.setX(int): @DisplayStateChange;
  declare @method: public void Point.setY(int): @DisplayStateChange;
  declare @method: public void Line.setP1(Point): @DisplayStateChange;
  declare @method: public void Line.setP2(Point): @DisplayStateChange;
}
```

As declarações das anotações também são geradas juntamente com os aspectos anotadores. Anotações devem ser declaradas para que sejam reconhecidas pelo compilador Java. Na solução proposta, essas declarações ficam em arquivos separados, que têm o mesmo nome das anotações. A declaração da anotação `@DisplayStateChange` é apresentada a seguir:

```
package Annotator;
public @interface DisplayStateChange { }
```

Uma vez definidas as anotações, o desenvolvedor responsável pelo código dos aspectos pode utilizá-las na implementação de conjuntos de junção. O código a seguir apresenta uma nova versão para o aspecto `UpdateSignaling`, utilizando a anotação `@DisplayStateChange`:

```
aspect UpdateSignaling {
  pointcut change(): execution(@DisplayStateChange * *(..));
  after() returning: change() {
    Display.update();
  }
}
```

4.3 Exemplos

Esta seção descreve a aplicação da solução proposta para aumentar a robustez de alguns conjuntos de junção de dois sistemas reais: JAccounting e HealthWatcher.

4.3.1 JAccounting

JAccounting[1] é um sistema web de contabilidade que utiliza o *framework* Hibernate para implementação de persistência e controle de transações. Binkley et al. realizaram uma refatoração da versão original do JAccounting criando uma versão que utiliza aspectos para implementação de controle de transações [BCH+06]. Essa nova versão utiliza o seguinte aspecto para modularizar esse interesse:

```
aspect Transaction {
  Transaction tx;
  pointcut p_0():
```

[1] https://jaccounting.dev.java.net

```
    call(Session SessionFactory.openSession() &&
    (( withincode(String ProductsPage.perform2())
    || withincode(String InvoicePage.perform2())
    || withincode(String RecurrencePage.perform2())
    || withincode(String PaymentPage.perform2())
    || withincode(String CustomerDetails.perform2())
    || withincode(String CustomerForm.perform2())
    || withincode(Account createInternalAccount(
       Session, Integer, int));
   after() returning: (Session sess) throws HibernateException: p_0 {
     tx = sess.beginTransaction();
   }
}
```

O conjunto de junção `p_0` define os pontos de junção que requerem controle de transações. Tais pontos de junção correspondem a chamadas do método `openSession`, realizadas no escopo léxico dos métodos enumerados na descrição do conjunto de junção. Implementações como essa são consideradas frágeis, pois podem conduzir a falhas no controle de transações se os responsáveis pela manutenção do sistema não atualizarem a definição do conjunto de junção `p_0` sempre que novos métodos que requerem o controle de transações forem adicionados.

A solução proposta nesta dissertação será utilizada para diminuir a fragilidade do conjunto de junção `p_0`. Para tal, a anotação `@Transactional` deve ser declarada da seguinte forma:

```
annotation
  name = @Transactional
  question = Is method %target transactional?
  target = methods && call(Session.save)
  scope = com
end
```

Os elementos candidatos a receberem esta anotação são métodos cujo corpo inclui, de maneira estática, pelo menos uma chamada ao método `save` da classe `Session`. Sabe-se que apenas tais métodos podem manipular transações em sistemas baseados no *framework* Hibernate. O escopo considerado será o pacote `com`, que é o pacote raiz na estrutura do JAccounting. Com a especificação da anotação `@Transactional`, o Annotator irá questionar os desenvolvedores sobre quais métodos, dentre os candidatos selecionados,

realmente necessitam de um controle de transações. O sistema JAccounting possui onze métodos que realizam chamadas a `Session.save`. Desses, sete efetivamente necessitam de controle de transações, conforme enumerado na implementação do conjunto de junção `p_0`. Os quatro métodos restantes, embora atualizem o banco de dados, não exigem controle de transações por razões distintas, incluindo, por exemplo, o fato deles compartilharem a sessão de banco de dados com outros métodos que são transacionais. Para os quatro métodos não transacionais, os desenvolvedores devem responder negativamente à pergunta feita pelo Annotator.

Utilizando a anotação que foi declarada, os desenvolvedores responsáveis por implementar o controle de transações podem utilizar uma nova implementação para o conjunto de junção `p_0`, conforme apresentado a seguir:

```
pointcut p_0():
  call(Session SessionFactory.openSession()) &&
  (withincode(@Transactional * * (..));
```

Quando o sistema JAccounting sofrer manutenções, de forma que métodos transacionais sejam adicionados, ou removidos, a nova implementação de `p_0` continuará válida. No caso da adição de novos métodos que realizam alterações no banco de dados, os desenvolvedores terão apenas que confirmar se os mesmos necessitam ou não de controle de transações.

4.3.2 HealthWatcher

HealthWatcher é uma aplicação *web* utilizada para registrar reclamações dos cidadãos (clientes) sobre problemas relacionados às condições sanitárias de restaurantes e lanchonetes. A primeira experiência utilizando AspectJ para modularizar os interesses transversais de distribuição e persistência do HealthWatcher foi conduzida por Soares, Borba e Laureano [SLB02]. Pelo menos dois aspectos presentes no sistema podem se beneficiar da solução proposta nesta dissertação: os aspectos de persistência e controle de transações.

Persistência: a versão orientada por aspectos do sistema utiliza o seguinte conjunto de junção para capturar chamadas de métodos que exigem sincronização do objeto alvo com o banco de dados:

```
pointcut remoteUpdate(PersistentObject o):
  this(HttpServlet) && target(o) && call(* set*(..));
```

A interface `PersistentObject` é utilizada para identificar classes cujos objetos devem ser atualizados no banco de dados logo após terem sido alterados. Tais classes são marcadas através de um aspecto que força que as mesmas implementem a interface `PersistentObject`, conforme exemplo a seguir:

```
declare parents: Complaint implements PersistentObject;
```

Declarações similares a essa são utilizadas também para as classes `Employee` e `HealthUnit`, que também demandam sincronização com o banco de dados. Durante evoluções do sistema HealthWatcher, quando, por exemplo, novas classes que requerem persistência forem implementadas, os desenvolvedores devem criar novas declarações inter-tipos que obriguem que essas novas classes implementem `PersistentObject`, conforme o exemplo mostrado para a classe `Complaint`. Essa exigência caracteriza uma fragilidade do conjunto de junção que implementa a persistência no HealthWatcher.

Na solução proposta, utilizando a ferramenta Annotator, a seguinte anotação é definida para identificar classes cujos objetos necessitam de persistência:

```
annotation
  name = @Persistent
  question = Is class %target persistent?
  target = classes && usedAs(param)
  scope = HealthWatcherFacade
end
```

O alvo da anotação `@Persistent` são classes utilizadas como parâmetro de métodos da classe `HealthWatcherFacade`, utilizada como fachada no sistema. Tais classes são naturalmente candidatas a persistência, uma vez a arquitetura do sistema define que a fachada é o único ponto de entrada do sistema, por meio do qual permite-se consulta e atualização dos objetos de negócio do HealthWatcher. Mesmo assim, para evitar falsos positivos, o Annotator irá solicitar a confirmação dos desenvolvedores sobre quais classes realmente demandam persistência no banco de dados. Existem, de fato, três classes utilizadas como parâmetros em métodos de `HealthWatcherFacade`. Todas representam objetos que exigem persistência.

Utilizando a anotação que foi declarada, os responsáveis pela implementação dos aspectos de controle de persistência podem utilizar a seguinte declaração inter-tipo para marcar as classes cujos objetos serão persistidos:

```
declare parents: (@Persistent *) implements PersistentObject;
```

Esta nova declaração determina que todas as classes anotadas com `@Persistent` devem implementar a interface `PersistentObject`, fazendo com que as mesmas sejam capturadas pelo conjunto de junção `remoteUpdate`. Com isso, durante manutenções e evoluções do sistema, quando o conjunto de classes que demandam persistência for alterado, não haverá necessidade de atualizar a declaração inter-tipo mostrada.

Controle de transações: a versão orientada por aspectos do sistema requer que métodos transacionais devam ser explicitamente declarados em uma interface denominada `ITransactionalMethods`. Essa interface é utilizada no seguinte conjunto de junção para capturar execuções de métodos que demandam controle de transações:

```
pointcut transactionalMethods(): execution(* ITransactionalMethods.*(..));
```

A maior desvantagem dessa implementação é que os desenvolvedores responsáveis pela aplicação base desconhecem a existência da interface `ITransactionalMethods`. Com isso, a adição de novos métodos transacionais no sistema pode conduzir a falhas na captura de pontos de junção. Para evitar esse problema, propõe-se a utilização da seguinte anotação:

```
annotation
  name = @Transactional
  question = Is method %target transactional?
  target = methods
  scope = HealthWatcherFacade
end
```

A declaração da anotação `@Transactional` requer que os desenvolvedores classifiquem os métodos da classe de fachada do sistema para identificar quais deles demandam controle de transações. É importante notar que a arquitetura do sistema HealthWatcher determina que apenas tais métodos são candidatos a controle de transações. Existem, de fato, quinze métodos na classe `HealthWatcherFacade`. Desses, cinco necessitam de controle de transações.

Utilizando a anotação `@Transactional`, o seguinte conjunto de junção pode ser utilizado para capturar execuções de métodos transacionais:

```
pointcut transactionalMethods(): execution(@Transactional * *(..));
```

Esta nova versão do conjunto de junção é totalmente independente da estrutura sintática da aplicação base. Além disso, a solução proposta automaticamente notifica os desenvolvedores sobre a adição de novos métodos na fachada do sistema, cabendo a

eles decidir se os novos métodos são ou não transacionais. Quando métodos transacionais são excluídos da fachada, a ferramenta detecta a eliminação do ponto de junção e atualiza automaticamente os aspectos anotadores e as *aspect-aware interfaces*.

4.4 Estudo de robustez

Esta seção baseia-se em um cenário de possíveis mudanças realizadas sobre o sistema clássico do Editor de Figuras. O objetivo é avaliar a robustez de conjuntos de junção implementados segundo a solução proposta nesta dissertação. O modelo utilizado é uma adaptação do cenário de alterações proposto por Ostermann, Mezini e Bockisch sobre o mesmo Editor de Figuras [OMB05]. O impacto das alterações propostas foi avaliado segundo quatro diferentes alternativas de implementação para o conjunto de junção `change`:

- Utilizando expressões regulares. Exemplo: `Figure+.set*(..)`.
- Utilizando enumerações. Exemplo: `Point.setX(..) || Point.setY(..)`.
- Utilizando anotações convencionais, aplicadas diretamente no código base.
- Utilizando anotações introduzidas pela ferramenta Annotator, conforme a solução proposta neste capítulo.

A Tabela 4.1 descreve cada uma das alterações utilizadas no estudo de robustez. O primeiro cenário (C1) corresponde à adição do campo `color` e do método `setColor` na classe `Point`. Quando o conjunto de junção `change` é construído com base em expressões regulares, exige-se que os nomes dos métodos que alteram atributos visuais começem com `set`. Embora isso represente uma regra baseada em nomes, essa regra geralmente é satisfeita por métodos utilizados para atualizar valores de campos (conhecidos como *setter methods*). Por essa razão, a implementação que utiliza expressões regulares é considerada robusta (+) para o cenário C1. A solução baseada em enumerações não é robusta (-) para C1, pois a enumeração deve ser alterada para adicionar o novo método (`setColor`). Da mesma forma, a solução que utiliza anotações convencionais não é robusta, pois requer que o novo método seja anotado. Por último, a solução que utiliza o Annotator é classificada como robusta, pois ela irá identificar a adição do novo método e questionar o desenvolvedor se esse método deve ou não alterar o estado do *display*.

No segundo cenário (C2), um campo que não altera nenhuma propriedade visual de figura é inserido na classe `Point`. Nesse caso, a solução baseada em expressões regulares não é robusta, pois irá gerar uma captura acidental do método `setDate`. A solução baseada

	Alteração	ER	Enum	Anotações	Annotator
C1	(Alteração em classe) Adicionar o campo `color` e o método `setColor` na classe `Point`	+	-	-	+
C2	(Alteração em classe) Adicionar o campo `date` e o método `setDate` na classe `Point`	-	+	+	+
C3	(Alteração em classe) Renomear método `setX` da classe `Point` para `changeX`	-	-	+	+
C4	(Alteração na hierarquia) Adicionar a classe `Circle` na hierarquia de `Figure`	+/-	-	-	+
C5	(Alteração na hierarquia) Renomear a interface `Figure` para `FigureElements`	-	-	+	-
C6	(Alteração no grafo de objetos) Utilizar um objeto do tipo `Pair` para armazenar as coordenadas de um `Point` e adicionar um método *setter* correspondente (`setPair`)	+/-	-	-	+/-
C7	(Alteração no fluxo de controle) Adicionar o campo `enable` na classe `Point` para controlar se determinado objeto deve ou não ser exibido no *display*	-	-	-	-

Tabela 4.1: Robustez dos conjuntos de junção baseados em expressões regulares (ER), enumerações (Enum), anotações Java e anotações adicionadas pela ferramenta Annotator, considerando sete possíveis cenários de mudança para o sistema Editor de Figuras.

em enumerações é robusta, pois não há necessidade de nenhuma alteração no conjunto de junção que contém as enumerações. A solução que utiliza anotações convencionais é também robusta porque não há necessidade de alterar os pontos de junção anotados. A solução que utiliza o Annotator é robusta, pois, assim que o método for inserido, a ferramenta irá questionar se aquele método altera ou não atributos visuais de figuras.

No terceiro cenário (C3), o método `setX` da classe `Point` é renomeado para `changeX`. Essa alteração representa uma falha na captura de ponto de junção, considerando os conjuntos de junção baseados em expressões regulares e enumerações. Por outro lado, a mudança do nome do método não afeta a solução baseada em anotações, pois a anotação continua presente no código base após a mudança. Também não afeta a solução que utiliza o Annotator, uma vez que os desenvolvedores irão confirmar que o método renomeado

continua alterando o estado do `Display`.

No quarto cenário (C4), a classe `Circle` é inserida na hierarquia de `Figure`. Nesse caso, o uso de expressões regulares é considerado parcialmente robusto (+/-), porque requer que os nomes dos métodos da nova classe que alterem o estado do `Display` comecem com `set`, sendo que aqueles que não alteram o `Display` não podem começar com esse prefixo. A solução baseada em enumerações não é robusta, pois o conjunto de junção com as enumerações não considera métodos da nova classe criada. Conjuntos de junção baseados em anotações convencionais também não são robustos, pois exigem que os métodos da classe `Circle` sejam anotados. Por último, a solução que utiliza o Annotator é robusta, pois irá questionar os desenvolvedores sobre a natureza de cada novo método que esteja no escopo definido no arquivo de configuração das anotações.

No quinto cenário (C5), a interface `Figure` é renomeada para `FigureElements`. Nesse caso, a solução que utiliza expressões regulares não é robusta, pois o conjunto de junção que contém a expressão regular baseia-se no nome `Figure`. Da mesma forma, a utilização de enumerações não é robusta, pois requer alterações na definição do conjunto de junção. A solução que utiliza o Annotator também não é robusta nesse caso, pois o escopo da anotação é o tipo `Figure`, conforme definido no arquivo de configuração (Seção 4.2.1). Por outro lado, a mudança não afeta em nada a solução baseada em anotações convencionais, pois as anotações permanecem válidas mesmo com a mudança no nome da interface.

No sexto cenário (C6), um campo do tipo `Pair` é utilizado para armazenar as coordenadas de um ponto, ao invés de utilizar os campos inteiros `x` e `y`. Nesse caso, conjuntos de junção baseados em expressões regulares são parcialmente robustos, pois irão capturar chamadas aos novos métodos que atualizam o campo `Pair`, desde que os nomes dos mesmos comecem com `set`. Contudo, não irão capturar chamadas a métodos da classe `Pair`, ainda que os mesmos alterem o estado do `Display`. A mesma situação ocorre na solução baseada no Annotator, pois a classe `Pair` não está no escopo definido para a anotação `@DisplayStateChange`. A solução que utiliza enumerações não é robusta, pois requer manutenção nos conjuntos de junção para adicionar os novos métodos criados. Da mesma forma, a solução baseada em anotações convencionais não é robusta, pois requer, de forma silenciosa, que os novos métodos que alteram o `Display` sejam anotados.

No sétimo cenário (C7), o novo campo `enabled` é criado para controlar se o `Display` deve ser ou não atualizado. Apenas figuras com `enabled==true` serão atualizadas no `Display`. Este cenário requer duas alterações principais: que a especificação do conjunto de junção seja alterada para expor o objeto alvo da chamada (*target*) e que seja incluído na interface `Figure` o método `isEnabled`. Como resultado, nenhuma das quatro opções de implementação é robusta para este cenário. O trecho de código a seguir apresenta

uma possível implementação para este cenário no aspecto `UpdateSignaling` do Editor de Figuras:

```
aspect UpdateSignaling {
   pointcut change(Figure f): execution(void Figure+.set*(..)) &&
                              target(f);
   after(Figure f): change(f) {
     if (f.isEnabled()) {
        Display.update();
     }
   }
}
```

A Tabela 4.2 resume a robustez avaliada para cada uma das quatro opções de implementação diante dos cenários que foram apresentados. A implementação que utiliza enumerações apresentou o menor grau de robustez, sendo, de fato, robusta em apenas um dos sete cenários pesquisados. Por outro lado, a solução utilizando a ferramenta Annotator mostrou-se robusta para quatro cenários e parcialmente robusta para um deles.

Conjunto de junção	+	+/-	-
Expressões Regulares	1	2	4
Enumerações	1	0	6
Anotações	3	0	4
Annotator	4	1	2

Tabela 4.2: Resumo dos resultados para os sete cenários avaliados

4.5 Implementação

A ferramenta Annotator é uma extensão do *plug-in* apresentado no Capítulo 3 para geração e atualização das AAIs. A nova versão do *plug-in* continua gerando as *aspect-aware interfaces* e o arquivo de *log* que registra todas as alterações ocorridas nas capturas de pontos de junção. No caso das AAIs, as mesmas passaram a receber também informações relativas às anotações introduzidas em pontos de junção do sistema base. A Figura 4.4 mostra a alteração feita na estrutura da Figura 3.3 para associar logicamente as anotações aos pontos de junção[2].

[2]O nome completo da anotação (`@DisplayStateChange`) foi abreviado para facilitar a montagem da figura.

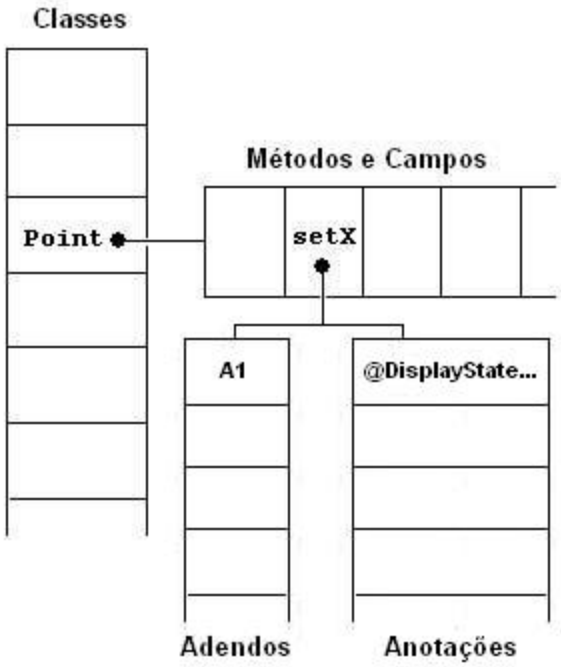

Figura 4.4: Estrutura de dados usada para armazenar *aspect-aware interfaces* com informações sobre anotações

Os aspectos anotadores e os arquivos utilizados para declarar as anotações são gerados em uma pasta chamada Annotator, que é criada dentro da pasta raiz do projeto no Eclipse. Portanto, para utilizar as anotações na implementação dos cojuntos de junção, os desenvolvedores devem importar o pacote Annotator nos arquivos que implementam os aspectos.

Os filtros utilizados no arquivo de configuração das anotações, `MethQualifiers` para métodos e `ClassQualifiers` para classes, são implementados com o auxílio da biblioteca ASM[3]. Com essa biblioteca, é possível obter informações estruturais do projeto através de leitura direta dos arquivos binários com extensão `.class`. Cada classe possui seu próprio arquivo `.class`, de onde podem ser obtidas informações sobre métodos, campos, parâmetros, exceções geradas, etc.

O Annotator mantém também um arquivo interno para armazenar as respostas dadas pelos desenvolvedores sobre quais pontos de junção devem ser anotados. São gravados no arquivo a identificação da anotação, o elemento alvo (campo, método ou classe) e a resposta S (Sim) ou N (Não). Esse arquivo não deve ser editado. Quando o desenvolvedor responder incorretamente a alguma pergunta feita pelo Annotator, deve-se editar e corrigir

[3]http://asm.objectweb.org/

a informação diretamente na AAI. A informação presente na AAI prevalece sobre o arquivo interno em caso de divergência. Se o elemento alvo for removido do programa base, as informações relativas a esse elemento são automaticamente removidas do arquivo interno.

4.6 Avaliação

O uso de anotações para definição de conjuntos de junção desacopla os mesmos da árvore de sintaxe abstrata do programa base, tornando-os mais robustos e independentes. Contudo, o uso de anotações em AspectJ apresenta dois problemas principais: as mesmas representam interesses transversais e são frágeis. Por exemplo, o seguinte conjunto de junção é desacoplado da árvore de sintaxe abstrata, mas requer que os desenvolvedores anotem corretamente, no código fonte do programa base, os métodos que devem ser capturados por este conjunto de junção:

```
pointcut change(): execution(void @DisplayStateChange *.*(..));
```

A solução proposta nesta dissertação contribui exatamente para reduzir a fragilidade e eliminar o comportamento transversal proveniente do uso de anotações na definição de conjuntos de junção. Para reduzir a fragilidade, a solução baseia-se em uma linguagem simples para declaração das anotações. Foi desenvolvida uma ferramenta, denominada de Annotator, que lê as anotações declaradas e questiona os desenvolvedores sobre quais elementos do programa, dentro do escopo especificado, devem ser realmente anotados. Para reduzir o espalhamento e entrelaçamento de código, o Annotator gera aspectos anotadores, que são responsáveis por adicionar as anotações no programa base de forma não invasiva. Aspectos anotadores são atualizados quando os desenvolvedores respondem positivamente a uma pergunta feita pelo Annotator sobre anotar ou não determinado elemento selecionado e quando algum elemento do programa base deixa de atender às condições estabelecidas no arquivo de configuração da anotação.

A solução é particularmente recomendada quando uma das seguintes situações acontece:

- É possível associar anotações com propriedades que caracterizam os elementos selecionados no seu escopo. Por exemplo, no sistema JAccounting, apenas métodos que executam `Session.save` são candidatos a controle de transações. Nesses casos, a seleção de elementos candidatos feita pela ferramenta torna-se mais precisa, o que evita falsos positivos, e também mais robusta em relação às evoluções que ocorrem no programa base, evitando também os falsos negativos.

- É possível restringir o escopo de uma anotação, como é o caso dos exemplos apresentados para o Editor de Figuras e para o HealthWatcher. No HealthWatcher, por exemplo, classes que demandam persistência são sempre utilizadas como parâmetro de métodos da classe de fachada do sistema. Nesses casos, reduz-se o número de vezes que os desenvolvedores são interrompidos com questionamentos sobre quais elementos selecionados devem ser anotados, diminuindo também as chances de ocorrência dos falsos positivos e falsos negativos.

Quando o escopo de uma anotação se baseia em elementos do programa base, espera-se, na solução proposta, que sejam utilizados nomes estáveis na especificação do escopo. Por exemplo, no sistema JAccounting o método `Session.save` é um dos métodos mais importantes fornecidos pelo *framework* de persistência Hibernate, utilizado pelo sistema. Portanto, é improvável que o nome desse método seja alterado. O mesmo acontece com o nome da classe de fachada utilizada no HealthWatcher. A utilização de nomes considerados não estáveis pode comprometer a declaração das anotações, conforme foi demonstrado no quinto cenário (C5) da análise de robustez realizada na Seção 4.4.

De certo modo, a solução proposta está coerente com uma tendência recente em Engenharia de Software que advoga que as linguagens de programação e ferramentas atuais não são flexíveis na manipulação da maioria das abstrações do mundo real, uma vez que tais abstrações são geralmente provisórias, parciais e estão em constante evolução [Kic07]. Assim, ao invés de projetar uma nova linguagem para definição de conjuntos de junção, mais sofisticada, formal e com abstrações específicas, decidiu-se reconhecer as limitações das linguagens orientadas por aspectos atuais e propor uma alternativa simples e compatível com o atual modelo de captura de pontos de junção de AspectJ. Por fim, sabe-se que a solução proposta não resolve todos os casos de fragilidade de conjuntos de junção, conforme foi demonstrado no estudo de caso da Seção 4.4. Conforme mencionado anteriormente, a eficácia da solução depende da capacidade de definir com precisão o tipo de objeto alvo (*target*) e o escopo (*scope*) no arquivo de configuração das anotações.

4.7 Comentários finais

Neste capítulo, apresentou-se o resultado principal desta dissertação de mestrado: uma proposta para diminuir a fragilidade dos conjuntos de junção em AspectJ, a qual se baseia no conceito de aspectos anotadores. Tais aspectos são responsáveis pela introdução de anotações no programa base de forma não invasiva. Uma ferramenta foi desenvolvida para viabilizar a geração das AAIs e dos aspectos anotadores, de forma semi-automática. A ferramenta irá interagir com os desenvolvedores para selecionar quais elementos do

programa base devem ser anotados. Como benefício, conjuntos de junção implementados com base nas anotações criadas tornam-se mais robustos com relações a futuras manutenções no programa base. No próximo capítulo, são apresentadas as conclusões finais deste trabalho de dissertação.

Capítulo 5

Conclusão

O problema de fragilidade dos conjuntos de junção representa uma séria limitação à utilização de linguagens orientadas por aspectos em sistemas de software de maior porte e complexidade. Esse problema se manifesta quando, em razão de uma modificação no programa base, conjuntos de junção passam a capturar pontos de junção indesejados ou deixam de capturar determinados pontos de junção essenciais ao correto funcionamento de um sistema. Nesta dissertação, foi proposta uma solução para atenuar o problema dos conjuntos de junção frágeis de AspectJ por meio da combinação dos conceitos de *aspect-aware interfaces* e aspectos anotadores.

Aspect-aware interfaces foram originalmente propostas para serem utilizadas no apoio a raciocínio modular (isto é, a capacidade de se raciocinar localmente sobre um módulo, entendendo suas funções e interfaces, sem a necessidade de investigar os demais módulos do sistema) [KM05a]. As AAIs possuem informações sobre todas as classes, campos e métodos impactados por conjuntos de junção. Além de preservarem a característica de apoio a raciocínio modular, as AAIs constituem um instrumento útil para detecção e validação de fragilidades em conjuntos de junção. Nesta dissertação, foi desenvolvida uma ferramenta que gera AAIs sempre que um projeto AspectJ é compilado. Além disso, a ferramenta monitora alterações no programa base para atualizar as AAIs automaticamente.

Todas as manutenções que afetam as AAIs são registradas em um arquivo de *log*. No Capítulo 3, foi apresentado um estudo de caso que ilustrou o impacto, no funcionamento do código dos aspectos, decorrente da realização de algumas alterações no programa base. O estudo mostrou como o *log* gerado pelas manutenções realizadas nas AAIs pode auxiliar na identificação de falhas na captura e capturas acidentais de pontos de junção. O estudo de caso mostrou também que a solução, no entanto, não é capaz de auxiliar na detecção de novos pontos de junção adicionados a um programa que não seguem as convenções de

nome pressupostas por um aspecto (conforme é o caso da adição do método `changeColor`, no estudo de caso). Uma possível solução para este problema consiste em adicionar informações semânticas no modelo de captura de pontos de junção, o que foi conseguido com a combinação do uso de AAIs e aspectos anotadores.

Aspectos anotadores. Na solução proposta, anotações são utilizadas para classificar elementos do programa base de acordo com suas características semânticas. Aspectos anotadores são responsáveis por introduzir anotações no programa base de forma não invasiva. As AAIs foram adaptadas para armazenar também informações sobre os pontos de junção que foram anotados. Além de aumentar a robustez dos conjuntos de junção, o uso de aspectos anotadores evita dois problemas decorrentes do uso de anotações convencionais em Java: entrelaçamento e espalhamento do código relativo às anotações.

As anotações utilizadas são definidas por meio de uma linguagem declarativa. Essa linguagem permite a especificações de padrões de código que serão utilizados para identificar os elementos do programa base que são candidatos a receberem anotações. A linguagem não define, entretanto, quais elementos serão anotados. Para tal, os desenvolvedores da aplicação são questionados sobre quais desses elementos candidatos devem realmente ser anotados. Uma das vantagens dessa abordagem é que os padrões de código utilizados na linguagem declarativa de anotações são menos susceptíveis a alterações que podem ocorrer durante a evolução do sistema. Por outro lado, se o escopo definido para uma anotação for muito amplo, o número de questionamentos feitos aos desenvolvedores pode se tornar um entrave à produtividade no desenvolvimento do sistema. Considera-se que, ao aplicar a solução proposta em um sistema orientado por aspectos, nem todos os conjuntos de junção poderão se beneficiar do uso de aspectos anotadores, devendo, portanto, ser implementados da forma tradicional. Para esses casos, o *log* que registra alterações nas AAIs continua sendo útil para detecção de falhas na captura e capturas acidentais de pontos de junção.

A solução proposta foi capaz de reduzir a fragilidade de alguns pontos de junção em dois sistemas reais orientados por aspectos, JAccounting e HealthWatcher. Além disso, foi realizado um estudo de caso com o sistema do Editor de Figuras que demonstrou que a solução proposta é mais robusta que as soluções baseadas em expressões regulares, enumerações e anotações convencionais.

5.1 Contribuições

As contribuições obtidas com este trabalho de mestrado estão listadas a seguir:

- Apresentação de uma proposta para atenuar o problema de fragilidade dos conjuntos de junção em AspectJ, combinando os conceitos de *aspect-aware interfaces* e aspectos anotadores.

- Utilização de anotações para definição de conjuntos de junção sem, no entanto, apresentar os problemas de entrelaçamento e espalhamento de código que caracterizam anotações convencionais de Java. Na solução proposta, anotações são introduzidas no programa base de forma não invasiva, por meio de aspectos anotadores.

- Desenvolvimento de uma ferramenta que implementa a solução proposta e gera as *aspect-aware interfaces* e os aspectos anotadores, de forma semi-automática. A ferramenta interage com os desenvolvedores para determinar quais pontos de junção devem ser efetivamente anotados.

- Demonstração do uso de aspectos anotadores em dois sistemas orientados por aspectos (HealthWatcher e JAccounting). Mostrou-se também que a solução proposta foi capaz de reduzir a fragilidade de alguns conjuntos de junção desses sistemas.

- Apresentação de um estudo de robustez que compara a solução proposta com soluções baseadas em expressões regulares, enumerações e anotações convencionais.

- Publicação do artigo "Controlando a Evolução de Sistemas Orientados por Aspectos por meio de Aspect-Aware Interfaces" no LA-WASP 2007 [SV07]. Este artigo apresenta a utilização de *aspect-aware interfaces* para auxiliar na detecção de falhas na captura e capturas acidentais de pontos de junção.

- Artigo a ser publicado "Non-invasive and Non-scattered Annotations for More Robust Pointcuts" na 24th IEEE International Conference on Software Maintenance (ICSM) 2008 [SVD08]. Este artigo apresenta a utilização de aspectos anotadores e *aspect-aware interfaces* para definição de conjuntos de junção mais robustos.

5.2 Trabalhos futuros

Como trabalho futuro, pretende-se investigar a aplicação da solução proposta em outros conjuntos de junção dos sistemas já estudados e também em outros sistemas reais orientados por aspectos. Outros estudos de caso serão úteis, principalmente, para:

- Validar a abrangência da linguagem declarativa para definição das anotações, no que se refere à definição dos elementos candidatos (*target*) e do escopo (*scope*) considerados.

- Verificar novas situações nas quais a solução proposta reduz a fragilidade dos conjuntos de junção.
- Verificar novas situações nas quais a solução proposta não melhora a robustez do código.

Além do estudo envolvendo sistemas já existentes, seria interessante realizar um estudo de caso que avaliasse o desenvolvimento completo de um sistema (desde a fase de análise até o produto final) utilizando a solução proposta. Outra linha de trabalho futuro consiste em investigar o comportamento da ferramenta proposta em um ambiente de desenvolvimento multi-usuário, ou seja, com vários desenvolvedores trabalhando no mesmo projeto. Em um ambiente assim, é interessante que a resposta dada por um desenvolvedor seja válida para todos os outros, para que não haja repetição das perguntas.

Bibliografia

[AC04] R. Altman and Alan Cyment. Setpoint: A semantic approach for the pointcut resolution in AOP, 2004.

[Ald05] Jonathan Aldrich. Open modules: modular reasoning about advice. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer Verlag, 2005.

[BCH+06] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions Software Engineering*, 32(9):698–717, 2006.

[CKAA06] Alan Cyment, Nicolas Kicillof, Rubén Altman, and Fernando Asteasuain. Improving AOP systems evolvability by decoupling advices from base code. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 9–21, 2006.

[CPA06] Walter Cazzola, Sonia Pini, and Massimo Ancona. Design-based pointcuts robustness against software evolution. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 35–45, 2006.

[DNBS07] Marcos Dósea, Alberto Costa Neto, Paulo Borba, and Sérgio Soares. Specifying design rules in aspect-oriented systems. In *I Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*, pages 67–78, October 2007.

[EA06] Marc Eaddy and Alfred V. Aho. Statement annotations for fine-grained advising. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 89–99, 2006.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOSPLA Workshop on Advanced Separation of Concerns*, October 2000.

[Foua] Eclipse Foundation. AJDT: Aspectj development tools. http://www.eclipse.org/ajdt/.

[Foub] Eclipse Foundation. Join point matching based on annotations. http://www.eclipse.org/aspectj /doc/released/adk15notebook/annotations-pointcuts-and-advice.html.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 60–69, 2003.

[GL03] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[GSS+06] William G. Griswold, Kevin J. Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[HNBA06] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 214–225, 2006.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355. Springer Verlag, 2001.

[Kic07] Gregor Kiczales. Effectiveness sans formality. In *Keynote talk at OOPSLA*, 2007.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[KM05a] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *27th International Conference on Software Engineering (ICSE)*, pages 49–58, 2005.

[KM05b] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *19th European Conference Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer Verlag, 2005.

[KMBG06a] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *20th European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 501–525. Springer Verlag, 2006.

[KMBG06b] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. A model-driven pointcut language for more robust pointcuts. In *AOSD Workshop on Software Engineering Properties of Languages for Aspect Technology*, 2006.

[KS04] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[Lad03] Ramnivas Laddad. *AspectJ in Action*. Manning Publications, 2003.

[Mic] Sun MicroSystems. Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer-Verlag, 2005.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[SG05] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, 2005.

[SGS+05] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *13th International Symposium on Foundations of Software Engineering (FSE)*, pages 166–175, 2005.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *17th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 174–190, 2002.

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *21st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 481–497, 2006.

[SV07] Leonardo Humberto Guimarães Silva and Marco Túlio Oliveira Valente. Controlando a evolução de sistemas orientados por aspectos por meio de aspect-aware interfaces. In *I Latin-American Workshop on Aspect-Oriented Software Development (LA-WASP)*, pages 81–92, 2007.

[SVD08] Leonardo Humberto Guimarães Silva, Marco Túlio Oliveira Valente, and Samuel Domingues. Non-invasive and non-scattered annotations for more robust pointcuts. In *24th IEEE International Conference on Software Maintenance (ICSM)*, 2008.