Seletores de pontos de junção: um mecanismo de extensão para linguagens e arcabouços orientados a aspectos

Cristiano Malanga Breuel

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Área de Concentração: Sistemas de Software

Orientador: Prof. Dr. Francisco da Rocha Reverbel

São Paulo, fevereiro de 2008

Seletores de pontos de junção:
um mecanismo de extensão para linguagens e arcabouços orientados a aspectos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Cristiano Malanga Breuel
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. Francisco Carlos da Rocha Reverbel (orientador) – IME-USP.
- Prof. Dr. Fabio Kon – IME-USP.
- Prof. Dr. Marco Túlio de Oliveira Valente – PUC-MG.

Resumo

Uma das questões mais importantes nas linguagens e arcabouços orientados a aspectos atuais é a expressividade da linguagem ou mecanismo de definição de *pointcuts*. A expressividade de uma linguagem de *pointcuts* impacta diretamente a qualidade dos *pointcuts*, uma propriedade que pode ser decisiva para a eficácia das implementações de aspectos. Neste trabalho, propomos os seletores de pontos de junção como um mecanismo de extensão simples para enriquecer linguagens de *pointcut* atuais com elementos que fazem o papel de "novos *pointcuts* primitivos". Os seletores de pontos de junção permitem a criação de *pointcuts* com maior valor semântico. Apesar de existirem mecanismos similares em algumas abordagens existentes, o conceito subjacente não foi claramente definido ou completamente explorado.

Apresentamos também uma arquitetura simples para a adição de seletores de pontos de junção a um arcabouço orientado a aspectos existente, e mostramos exemplos do uso de seletores para melhorar a qualidade de *pointcuts* e facilitar o desenvolvimento de aspectos.

Palavras-chave: Programação Orientada a Aspectos, Linguagens de *Pointcut*, *Pointcuts* Semânticos, Extensibilidade, Seletores de Pontos de Junção.

Abstract

One of the main issues in modern aspect-oriented programming languages and frameworks is the expressiveness of the pointcut language or mechanism. The expressiveness of pointcut languages directly impacts pointcut quality, a property that can be decisive for the effectiveness of aspect implementations. In this work we propose join point selectors as a simple extension mechanism for enriching current pointcut languages with constructs that play the role of "new primitive pointcuts". Join point selectors allow the creation of pointcuts with greater semantic value. Although similar mechanisms can be found in some existing approaches, the underlying concept has not yet been clearly defined nor fully explored.

We also present a simple architecture for adding join point selectors to an existing aspect-oriented framework, and show examples of usage of join point selectors to enhance the quality of pointcuts and make aspect development easier.

Keywords: Aspect-Oriented Programming, Pointcut Languages, Semantic Pointcuts, Extensibility, Join Point Selectors.

Sumário

Lista de Figuras **ix**

Lista de Tabelas **xi**

Glossário **xiii**

1 Introdução **1**

1.1 Objetivos do trabalho 2

1.2 Organização do trabalho 2

2 Programação Orientada a Aspectos **3**

2.1 Terminologia 3

2.2 Conceitos 3

2.3 Pontos de junção e *pointcuts* 4

2.4 JBoss AOP 7

2.5 Qualidade de um *pointcut* 9

2.6 Estilos de definição de pointcuts 9

2.6.1 Enumeração 9

2.6.2 Anotações 10

2.6.3 Declarações semânticas 11

3 Seletores de pontos de junção **13**

3.1 Motivação . . . 13

3.1.1 Editor de figuras . . . 13

3.1.2 Segurança . . . 15

3.1.3 Aspectos para *Web Services* . . . 17

3.2 Conceito . . . 18

4 Uma implementação de seletores de pontos de junção **21**

4.1 Visão do programador de seletores . . . 23

4.1.1 Meta-informação . . . 24

4.1.2 Uso de um seletor . . . 25

4.2 Visão interna . . . 25

4.2.1 Mecanismo de interpretação . . . 26

4.2.2 O processo de combinação . . . 29

4.3 Desempenho . . . 30

5 Aplicações **33**

5.1 Tipo de parâmetro . . . 33

5.2 Editor de figuras . . . 35

5.3 Chamadas por reflexão . . . 35

5.4 Aspectos para *Web Services* . . . 37

5.5 Seletores específicos para arcabouços . . . 39

6 Trabalhos relacionados **47**

6.1 Programação funcional . . . 47

6.1.1 XQuery/BAT . . . 47

6.2 Programação lógica 48
6.2.1 Andrew 48
6.2.2 Gamma 50
6.2.3 Alpha 51
6.3 Programação imperativa 52
6.3.1 Josh 52
6.4 Avaliação comparativa 54
6.4.1 *Pointcuts* dinâmicos 54
6.4.2 Informação temporal 54
6.4.3 Resistência a mudanças 55
6.4.4 Clareza de intenção 55
6.4.5 Viabilidade prática 55

7 Considerações finais **57**
7.1 Principais contribuições 57
7.2 Trabalhos futuros 58
7.2.1 Otimização de expressões *pointcut* 58
7.2.2 Mudança do modelo de combinação 58
7.2.3 Agregar informação ao *pointcut* para uso em adendos 59
7.3 Conclusão 59

Referências Bibliográficas **61**

Lista de Figuras

3.1 Diagrama UML do editor de figuras 14
3.2 O funcionamento de um seletor 18

4.1 Classes que representam a árvore sintática da linguagem de *pointcuts* 26
4.2 A estrutura de *matchers* do JBoss AOP 28

5.1 Hierarquia de classes da biblioteca de seletores do Hibernate 42

Lista de Tabelas

2.1 Comparação de linguagens orientadas a aspectos 6

6.1 Comparação de novas abordagens em linguagens de *pointcuts* 54

Glossário

Terminologia de programação orientada a aspectos

adendo (*advice*) elemento que especifica o comportamento transversal a ser executado, semelhante a um método.

combinação (*weaving*) processo de junção dos aspectos aos módulos tradicionais.

declaração intertipos (*inter-type declarations*) mecanismo para adição de elementos estruturais a classes do programa base como parte da implementação de aspectos.

entrelaçamento (*entanglement*) presença de mais de um requisito (ou interesse) em uma mesma unidade (módulo).

espalhamento (*scattering*) divisão da implementação de um requisito (ou interesse) em diversas unidades (módulos).

interesse transversal (*crosscutting concern*) um requisito do software cuja implementação afeta a implementação de outros requisitos.

linguagem base a linguagem utilizada para implementação do programa base, sobre o qual serão aplicados os aspectos.

linguagem de *pointcuts* parte de uma linguagem orientada a aspectos usada para definir *pointcuts*.

linguagem orientada a aspectos linguagem (ou arcabouço) que permite a definição de requisitos transversais através da definição de *pointcuts*, adendos e declarações intertipos.

POA Programação Orientada a Aspectos.

pointcut conjunto de pontos de junção.

ponto de junção (*join point*) ponto de interesse na execução de um programa no qual é

possível inserir funcionalidades definidas por aspectos.

programa base implementação dos interesses (requisitos) não transversais de um programa, feita em um paradigma tradicional, como procedimental ou orientado a objetos.

sombra de ponto de junção (*join point shadow*) ponto na estrutura estática de um programa (que pode ser seu código-fonte, uma árvore sintática abstrata ou uma representação binária compilada) que corresponde a um ponto de junção.

Capítulo 1

Introdução

A complexidade do desenvolvimento de software é um problema que acompanha essa indústria há várias décadas. Por muito tempo, era lugar comum falar na existência de uma "crise do software" [10], que parecia nunca ter solução.

Grande parte da complexidade em projetos de software é causada pelo incessante aumento do grau de sofisticação dos sistemas produzidos, que sempre incorporam novas funcionalidades. Ou seja, muito da complexidade é intrínseca ao problema que buscamos resolver.

Existe, entretanto, outra fonte de complexidade, que é o próprio conjunto de ferramentas usadas para produzir software, como linguagens de programação e arcabouços. Se essas ferramentas obrigarem o programador a lidar com problemas que não estão diretamente ligados ao objetivo do software que ele pretende produzir, elas estarão introduzindo complexidade desnecessária. É na redução dessa complexidade extra que se podem obter grandes avanços no projeto de ferramentas de desenvolvimento.

No caso das linguagens e arcabouços, uma das maiores fontes de complexidade para o programador é a necessidade de trabalhar em um nível de abstração mais baixo do que o problema que ele busca resolver. Por isso, aumentar esse nível de abstração produz ganhos de produtividade.

A programação orientada a aspectos proporciona um aumento no nível de abstração pelo fato de permitir ao programador pensar em um determinado requisito transversal como uma unidade, o aspecto, e não mais como pedaços de código espalhados em meio a outras funcionalidades. Entretanto, para definir os pontos de intersecção entre requisitos transversais e convencionais, o programador ainda precisa lidar com conceitos que estão em um nível de abstração inferior. Além de aumentar a

complexidade para o programador, isso causa dificuldades para criar definições que mantenham suas propriedades diante de mudanças no programa.

Neste trabalho, procuramos continuar o esforço de elevação no nível de abstração, ao permitir que se usem conceitos de alto nível para definir os pontos de intersecção entre requisitos transversais e convencionais. Além disso, pretendemos aumentar a resistência dos aspectos a mudanças em outros módulos.

1.1 Objetivos do trabalho

Esta dissertação apresenta uma proposta para o avanço da Programação Orientada a Aspectos no sentido de reduzir a complexidade para o programador, permitindo que ele trabalhe em um nível de abstração mais alto do que o possível atualmente. Para atingir esse objetivo, acrescentamos a uma linguagem de definição de *pointcuts* a possibilidade de definir e usar novas unidades de abstração, semelhantes a funções, às quais demos o nome de "seletores de pontos de junção".

Os objetivos deste trabalho são:

- Apresentar e analisar os problemas encontrados nas linguagens de *pointcut* atuais;
- Definir o conceito proposto de seletores de pontos de junção;
- Descrever a implementação do protótipo que criamos como prova de conceito;
- Mostrar exemplos de aplicações em que os seletores produzem soluções mais simples e robustas;
- Apresentar outros trabalhos que possuem propostas similares, e comparar seus resultados entre si e em relação a esta proposta.

1.2 Organização do trabalho

Organizamos esta dissertação da seguinte forma: o Capítulo 2 introduz conceitos básicos sobre a programação orientada a aspectos, o Capítulo 3 introduz os problemas que buscamos solucionar e os conceitos principais da proposta, e o Capítulo 4 descreve a implementação que realizamos destes conceitos, o Capítulo 5 mostra alguns exemplos de aplicação do conceito apresentado, o Capítulo 6 apresenta trabalhos cuja proposta tem relação com este, e o Capítulo 7 contém algumas considerações finais e conclusões.

Capítulo 2

Programação Orientada a Aspectos

2.1 Terminologia

A terminologia original da Programação Orientada a Aspectos foi definida na língua inglesa. As traduções que utilizamos aqui foram em sua maior parte definidas no Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos [15].

Entretanto, duas exceções foram feitas. A tradução definida naquele workshop para o termo *pointcut* ("conjunto de pontos de junção"), além de muito extensa, em nossa opinião não expressa adequadamente o seu significado, o que poderia levar a imprecisões na interpretação do texto, e por isso mantivemos o termo em seu idioma original. O termo *advice* tinha sido originalmente traduzido como "comportamento transversal", mas discussões posteriores na comunidade favoreceram o termo "adendo", mais preciso e conciso, que utilizamos aqui.

Para o termo *join point shadow*, cuja tradução não foi incluída na lista do workshop, utilizamos "sombra de ponto de junção" ou apenas "sombra", quando o contexto o permitiu.

No primeiro uso de cada termo neste texto, indicamos também seu equivalente em inglês.

2.2 Conceitos

A Programação Orientada a Aspectos [25] (POA, ou AOP como é conhecida em inglês), é um paradigma de programação que surgiu com a intenção de modularizar características de um programa cuja implementação não pode ser claramente modularizada nos paradigmas tradicionais. Essas características são chamadas de interesses transversais (*crosscutting concerns*), e sua implementação

através de técnicas convencionais como a POO (Programação Orientada a Objetos) geram código não localizado, entrelaçado (*tangled*) ao restante do sistema.

A POA não pretende substituir a Programação Orientada a Objetos ou a procedimental, mas complementar essas abordagens. Por isso, as principais implementações de linguagens orientadas a aspectos atuais ou são extensões de linguagens existentes, ou são arcabouços escritos em uma linguagem existente[1] [6]. Em ambos os casos, há uma etapa em que o código orientado a aspectos é inserido em pontos específicos do código "base" (orientado a objetos ou procedimental), num processo chamado de combinação (*weaving*).

Para atingir o isolamento de interesses transversais, a POA introduz um novo tipo de abstração: o *aspecto*. Um aspecto consiste em partes que implementam o interesse e partes que definem onde essa implementação deve ser inserida no programa base.

A implementação dos interesses é composta por adendos (*advices*) e declarações intertipos (*inter-type declarations*). Essas partes normalmente são escritas na mesma linguagem do programa base, às vezes com pequenas extensões. As partes de um aspecto que definem onde ele deve se aplicar são chamadas de *pointcuts*. Na seção seguinte, examinaremos este conceito em mais detalhes.

2.3 Pontos de junção e *pointcuts*

O principal modelo para as linguagens orientadas a aspectos atuais é a linguagem AspectJ [23], criada como uma extensão para a linguagem orientada a objetos Java. Os conceitos aqui apresentados originaram-se nessa linguagem, mas hoje são usados na maioria das abordagens, sejam elas linguagens ou arcabouços.

Os pontos de junção (*join points* ou *joinpoints*) são os pontos de interesse na execução de um programa nos quais é possível inserir funcionalidades definidas por aspectos. Os tipos de pontos de junção mais comuns são: chamadas e execuções de métodos, leituras e escritas de campos, tratamentos de exceções e inicializações de classes. Os pontos na estrutura estática de um programa (que pode ser seu código-fonte, uma árvore sintática abstrata ou uma representação binária compilada) correspondentes aos pontos de junção são chamados de sombras de pontos de junção (*join point shadows* [29]). Por exemplo, no caso de uma instrução de chamada a método que aparece dentro de um laço, cada passagem por essa instrução é um ponto de junção distinto, enquanto que a própria

[1] No restante do texto, quando utilizamos os termos linguagens ou abordagens orientadas a aspectos, estamos nos referindo indistintamente a linguagens e arcabouços, exceto quando explicitado.

instrução é a sombra de todos esses pontos de junção.

Os *pointcuts* são expressões definidas pelo programador que, aplicadas sobre o conjunto de pontos de junção de um programa, selecionam um subconjunto deles. O conjunto gerado pelo *pointcut* servirá de base para a aplicação de aspectos.

A listagem 2.1 exemplifica um aspecto simples, em AspectJ, que mostra uma mensagem na saída padrão sempre que um determinado método é chamado. Vê-se, na linha 1, que um aspecto é declarado de forma muito semelhante a uma classe, substituindo-se a palavra-chave "class" por "aspect". Nas linhas 3 e 4 temos a declaração do *pointcut*, chamado de `tracedCall`, que captura todas as chamadas (`call`) ao método `draw` da classe `FigureElement` que recebe um argumento do tipo `GraphicsContext`. As linhas 6 a 8 contém a declaração do adendo, que será executado antes (`before()`) dos pontos selecionados pelo *pointcut* `tracedCall`. O corpo do adendo (entre as chaves) pode conter qualquer código válido em Java. Neste bloco, o programador tem à disposição algumas variáveis implicitamente declaradas pelo AspectJ, como `thisJoinPoint`, que representa o ponto de junção no qual o adendo está sendo executado. No exemplo, a descrição deste ponto de junção é impressa na saída padrão.

Listagem 2.1 Um aspecto de rastreamento

```
1  aspect SimpleTracing {
2
3      pointcut tracedCall():
4          call(void FigureElement.draw(GraphicsContext));
5
6      before(): tracedCall() {
7          System.out.println("Entering: " + thisJoinPoint);
8      }
9
10 }
```

Nas linguagens atuais, os *pointcuts* dividem-se em dois tipos: os primitivos e os definidos pelo usuário. Os primeiros são pré-definidos como palavras-chave na linguagem, os segundos são compostos pelo programador através da combinação lógica dos primeiros[2].

Como referência, utilizaremos os principais arcabouços e linguagens orientados a aspectos baseados na linguagem Java que estão disponíveis atualmente. Uma comparação entre seus modelos de *pointcuts* pode ser vista na tabela 2.1, extraída de [21]. Para cada tipo de recurso da linguagem, ela

[2]Utilizamos neste parágrafo a terminologia corrente na área. Entretanto, consideramos que ela não é a ideal, como será visto na seção 3.2.

	AspectJ	**AspectWerkz**	**JBoss AOP**	**Spring AOP**
invocation	{method, constructor, advice} x {call, execution}			method execution
initialization	instance, static, pre-init	instance, static	instance	-
access	field get/set			-
exception handling	handler		(via advice)	
control flow	cflow, cflowbelow		(via specified call stack)	cflow
containment	within, winthincode	within, withincode, has method/field	within, winthincode, has method/field, all	-
conditional	if	-	(via Dynamic cflow)	custom pointcut
pointcut matching	signature, type pattern, subtypes, wild card, annotation		signature, instanceof, wild	card, annotation regular expression
pointcut composition	&&, ||, !			&&, ||
extensibility	abstract pointcuts	overriding, advice bindings		

Tabela 2.1: Comparação de linguagens orientadas a aspectos

mostra o que cada linguagem/arcabouço implementa.

Como vemos, as diversas linguagens orientadas a aspectos em uso atualmente têm características muito semelhantes no que se refere à definição de *pointcuts*. Em geral, elas permitem:

- Capturar chamadas e execuções de métodos, leitura e escrita de campos, tratamento de exceções e inicialização de classes;
- Restringir a captura por fluxo de controle (exemplo: chamadas ao método *m1* efetuadas no fluxo de controle do método *m2*), tipos onde os eventos ocorrem (exemplo: chamadas a *m1* efetuadas pela classe *c1*) e métodos com parâmetros de tipos específicos (exemplo: chamadas a *m1* que recebam um objeto do tipo *c2* como único parâmetro);
- Combinar esses predicados através de lógica booleana.

Devido a esta semelhança de expressividade, no restante da discussão compararemos nossa proposta (e outras relacionadas) às "linguagens em uso atual", genericamente. Nos casos em que são feitas comparações explícitas com alguma linguagem mencionada nesta seção, a comparação pode ser generalizada para as outras.

2.4 JBoss AOP

Para a implementação do protótipo apresentado neste trabalho, decidimos estender o arcabouço JBoss AOP. Por isso, apresentamos aqui uma introdução mais detalhada a esse arcabouço em especial.

O JBoss AOP é um arcabouço para programação orientada a aspectos de propósito geral. Embora seja extensivamente usado no servidor de aplicações JBoss [14], ele pode ser usado em qualquer servidor de aplicação ou independentemente.

Um dos recursos do JBoss AOP é o suporte à AOP dinâmica, ou seja, ele permite que se altere o comportamento da aplicação em tempo de carregamento ou execução, sem necessidade de recompilação. Além disso, os aspectos são criados na linguagem Java, sem alterações, evitando assim a necessidade de substituição do compilador Java por outro mais específico.

Para criar um adendo em JBoss AOP, implementa-se um método que possui a seguinte assinatura:

```
Object methodName(Invocation object) throws Throwable
```

O adendo recebe como parâmetro um objeto da classe `Invocation`[3], que representa o ponto de junção em execução. Esse objeto contém informações como o método invocado, o objeto que recebe a chamada e os argumentos fornecidos. Além disso, ele possui uma referência para a cadeia de interceptadores, que contém outros adendos e o próprio ponto de junção. Para continuar com o fluxo do ponto de junção, invocando o próximo elemento da cadeia de interceptadores, o adendo pode chamar o método `invokeNext` do `Invocation`.

Além de implementar a classe que define o adendo, o programador precisa fornecer a meta-informação que permite ao JBoss AOP identificar esse adendo. Além disso, os *pointcuts* também precisam ser definidos através de meta-informação.

Há duas formas de especificar essa meta-informação: através de um arquivo XML ou de anotações no próprio código. A listagem 2.2 mostra a definição de um aspecto com anotações. A listagem 2.3 mostra o mesmo aspecto definido com XML. Neste último caso, apenas a meta-informação é mostrada, e a implementação do aspecto poderia ser igual à do exemplo com anotações.

O JBoss AOP possui um mecanismo de reaproveitamento de *pointcuts* chamado de *pointcut*

[3] O nome *Invocation* não é completamente preciso, pois nem sempre um ponto de junção é uma invocação de método. Ele pode ser também uma escrita ou leitura de um campo, por exemplo. A razão da escolha desse nome é provavelmente histórica, pois existe uma classe de mesmo nome e funcionalidade semelhante, embora não idêntica, no servidor de aplicações JBoss.

Listagem 2.2 Uma definição de aspecto do JBoss AOP através de anotações

```
1  package com.mypackage;
2
3  import org.jboss.aop.Bind;
4
5  @Aspect()
6  public class MyAspect {
7
8      @Bind(pointcut="execution(* com.acme.*->*(..))")
9      public Object myAdvice(Invocation invocation) {
10     }
11
12 }
```

Listagem 2.3 Uma definição de aspecto do JBoss AOP através de XML

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE aop SYSTEM "jboss-aop_1_0.dtd">
3
4  <aop>
5
6      <aspect class="com.acme.MyAspect" />
7
8      <bind pointcut="execution(* com.acme.*->*(..))">
9
10         <advice aspect="MyAspect" name="myAdvice" />
11
12     </bind>
13
14 </aop>
```

nomeado (*named pointcut*), que consiste em atribuir um nome a uma expressão *pointcut*. Isso permite que essa expressão seja referenciada como parte de outras expressões. A listagem 2.4 exemplifica um uso desse recurso, através de uma configuração em XML.

Listagem 2.4 Declaração de *pointcuts* nomeados no JBoss AOP

```
1  <pointcut name="publicMethods" expr="execution(public * *->*(..))" />
2  <pointcut name="staticMethods" expr="execution(static * *->*(..))" />
3
4  <bind pointcut="publicMethods AND staticMethods">
5        <interceptor-ref name="tracing" />
6  </bind>
```

2.5 Qualidade de um *pointcut*

Quando um programador define um *pointcut*, ele tem em mente um conjunto de pontos de junção que devem fazer parte dele. Esse conjunto normalmente é definido por características em comum entre os pontos de junção. Tais características identificam o conjunto de pontos de junção nos quais deseja-se aplicar um aspecto. Em uma situação ideal, a definição do *pointcut* seria a própria definição semântica dos critérios idealizados pelo programador.

Definimos a qualidade da definição de um *pointcut* como a sua capacidade de atender aos seguintes requisitos:

- **Resistência a mudanças**: mudanças no programa não devem afetar negativamente o conjunto gerado pelo *pointcut*. Mais especificamente, quando um novo ponto de junção for adicionado ao programa ou um existente for modificado, o ponto de junção deve ser incluído se e somente se ele atender às características desejadas pelo autor do *pointcut*.
- **Clareza de intenção**: a definição de um *pointcut* deve ser feita de forma simples e próxima ao domínio modelado. Ou seja, ela deve transmitir, tanto quanto possível, e de forma simples e clara, a intenção de quem o definiu, para facilitar o entendimento e modificação da expressão.

Utilizaremos esse conceito de qualidade de *pointcut* para comparar as linguagens orientadas a aspectos atuais com a nossa proposta e com outras relacionadas. Nosso objetivo é melhorar a qualidade da definição dos *pointcuts*, tanto em relação à clareza da definição como à resistência a mudanças.

2.6 Estilos de definição de pointcuts

Durante o desenvolvimento da programação orientada a aspectos e de programas escritos nela, observamos que emergiram algumas formas básicas de especificar *pointcuts*. A seguir, classificamos estas formas de declaração, analisando suas vantagens e desvantagens.

2.6.1 Enumeração

A forma mais básica de definir um *pointcut* é enumerando-se os pontos de junção que devem compô-lo. Esta enumeração pode ser feita listando pontos de junção individualmente ou através de alguma convenção de nomenclatura. Os *pointcuts* na listagem 2.5 (em AspectJ) ilustram essa forma.

Listagem 2.5 *Pointcuts* por enumeração

```
pointcut DBActivity() :
  call(void Person.updateName(String)) ||
  call(void Person.updateAge(int)) ||
  call(void Company.insertPerson(Person)) ||
  call(void Company.deletePerson(Person));
```

```
pointcut DBActivity() :
  call(void *.update*(..)) ||
  call(void *.insert*(..)) ||
  call(void *.delete*(..));
```

Podemos usar como metáfora para esta forma de definição um "recrutamento". Neste caso, o recrutador é o programador do aspecto (ou da integração do aspecto a um programa específico), que escolhe os pontos de junção que deseja incluir no *pointcut*.

A principal vantagem do primeiro caso (listagem de pontos de junção individuais) é que não serão incluídos pontos de junção indesejados, a menos que seja feita uma alteração funcional que faça com que um dos pontos de junção não satisfaça mais às condições desejadas. A maior desvantagem é que não haverá inclusão automática de novos pontos de junção desejáveis.

No segundo caso, há maior chance de que novos pontos de junção relevantes sejam incluídos (se os programadores seguirem a convenção de nomenclatura), mas por outro lado torna-se possível a inclusão de pontos de junção indesejados.

2.6.2 Anotações

Uma outra forma de determinar os pontos de junção de um *pointcut* é marcá-los com alguma forma de meta-informação, o que pode ser feito através de "anotações" de elementos na linguagem Java[4]. Neste caso, o programador do elemento é quem decide que ele tem uma determinada propriedade e o marca dessa forma, permitindo que o *pointcut* o selecione. Vemos um exemplo disso na listagem 2.6.

Seguindo com as metáforas, podemos dizer que o método em questão "alista-se" para fazer parte do *pointcut*, ao declarar-se possuidor de uma propriedade interessante para este. Em comparação com a primeira abordagem, esta é mais precisa. Não é mais necessário confiar em convenções de nomenclatura, que são mais frágeis que anotações, pois as anotações são checadas pelo compilador. Além disso, elimina-se a possibilidade de inclusão involuntária de pontos de junção em um *pointcut*, pois o programador deve adicionar a anotação explicitamente em cada ponto.

No lado negativo, ainda é necessário depender da aderência do programador a uma convenção.

[4] Recurso disponível a partir da versão 1.5 da linguagem.

Listagem 2.6 *Pointcuts* por anotação

```
public class Person {

  @DBActivity
  public void updateName(String newName) {
    DBManager.openConnection();
    ...
  }

}

aspect DBConnectionDisposal {

  pointcut DBActivity() :
    call(@DBActivity * *(..));

  after() : DBActivity() {
    DBManager.closeConnection();
  }

}
```

Além disso, esta técnica é mais intrusiva, pois pode exigir mudanças no programa base para acomodar novos aspectos, reduzindo o isolamento entre eles.

2.6.3 Declarações semânticas

Finalmente, os *pointcuts* podem ser declarados baseando-se na semântica da situação que se deseja capturar. A listagem 2.7 exemplifica este estilo.

Neste caso, a idéia do programador era capturar todas as execuções do método `Server.initiate Process()` que tenham aberto a conexão com o banco de dados, para garantir que, ao final, esta conexão será fechada.

O *pointcut* DBActivity captura execuções do método `initiateProcess` que tenham tido uma chamada do método `DBManager.openConnection()` em seu fluxo de controle previsto (`pcflow` [24]). Para isso, utiliza a cláusula hipotética `containsCall`, que capturaria tal condição, mas não existe em nenhuma linguagem orientada a aspectos atual.

Este *pointcut* é muito mais resistente a mudanças que os anteriores, pois não se baseia em convenções de nomenclatura nem em declarações de meta-informação. Não importando como o programa foi implementado, se uma conexão ao banco de dados foi aberta, ela será fechada, garantindo que a intenção do programador do aspecto foi realizada.

Listagem 2.7 *Pointcuts* semânticos

```
public class Person {

  public void updateName(String newName) {
    DBManager.openConnection();
    ...
  }

}

aspect DBConnectionDisposal {

  pointcut DBActivity(): pcflow(execution(void Server.initiateProcess()))
    && containsCall(* DBManager.openConnection());

  after DBActivity()(): {
    DBManager.closeConnection();
  }

}
```

Comparado às metáforas dos *pointcuts* anteriores, neste caso o que teríamos seria uma espécie de "filtragem", em que os pontos de junção selecionados seriam os que efetivamente possuem a propriedade de abrir uma conexão com o banco de dados, não importa quais sejam os seus nomes ou propriedades.

Com base nessas propriedades, vemos que esse tipo de *pointcut* permite menor acoplamento e declarações mais "intuitivas", que tornam a programação de aspectos mais fácil e eficiente. No entanto, as capacidades das linguagens orientadas a aspectos atuais nem sempre permitem a criação de *pointcuts* deste tipo. O exemplo acima está em pseudo-código, pois os *pointcuts* `pcflow` e `containsCall` não existem em nenhuma linguagem atualmente em uso.

Capítulo 3

Seletores de pontos de junção

3.1 Motivação

Os recursos para definição de *pointcuts* à disposição nas linguagens atuais atendem à maioria das necessidades. Muitas vezes, entretanto, eles não permitem criar *pointcuts* de boa qualidade, tanto no que se refere à resistência a mudanças quanto à clareza.

O problema da resistência a mudanças é bem conhecido e estudado, e recebeu o nome de problema do *pointcut* frágil (*fragile pointcut problem* [28]). A questão da clareza não recebeu um nome específico, talvez por ter menos trabalhos a seu respeito.

Apresentamos nesta seção alguns exemplos de problemas que evidenciam a necessidade de melhoramentos no modelo de *pointcuts* das linguagens orientadas a aspectos. Apresentamos também uma solução possível para cada um desses problemas na linguagem orientada a aspectos mais popular, o AspectJ.

3.1.1 Editor de figuras

Um dos exemplos mais utilizados em trabalhos sobre programação orientada a aspectos [7, 13, 23, 26, 30] é o de uma hipotética aplicação gráfica [2]. Nessa aplicação, existem diversos elementos gráficos, como linhas e retângulos, representados por classes. Quando é chamado um método que altera o estado de uma instância de alguma dessas classes, o método `redraw()` deve ser chamado para atualizar a exibição do elemento. Na figura 3.1, vemos o diagrama UML [32] desse editor de figuras.

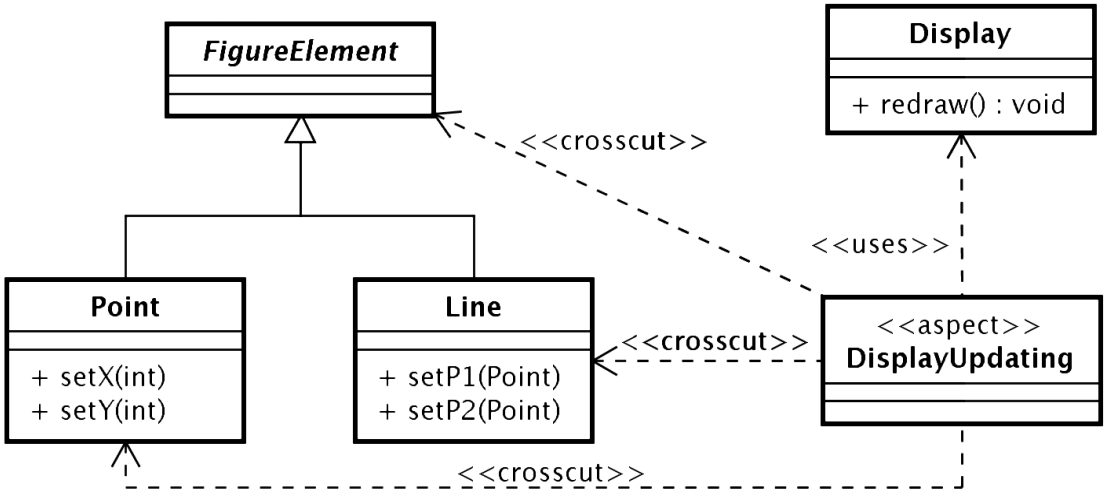

Figura 3.1: Diagrama UML do editor de figuras

Embora seja usado como exemplo de aplicação da Programação Orientada a Aspectos, este problema é também um bom exemplo das limitações dessas linguagens. A listagem 3.1 mostra duas possíveis soluções para o problema, em AspectJ.

Listagem 3.1 Soluções para o editor de figuras em AspectJ

```
aspect DisplayUpdating {
 pointcut move() :
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point)) ||
  call(void Point.setX(int)) ||
  call(void Point.setY(int));

 after() returning : move() {
  Display.redraw();
 }
}
```

```
aspect DisplayUpdating {
 pointcut move() :
  call(void FigureElement+.set*(..));

 after() returning : move() {
  Display.redraw();
 }
}
```

Na primeira solução, todos os métodos que afetam as figuras são enumerados no *pointcut*. Se um programador adicionar um novo método que altera o estado de uma figura e não alterar o *pointcut*, este deixará de ser completamente efetivo. O mesmo ocorre se for criada uma nova subclasse de `FigureElement`.

A segunda solução é mais genérica, pois inclui qualquer subtipo de `FigureElement`, e seleciona todos os métodos cujo nome se inicia por "set". Neste caso, a adição de um novo método já não quebra o *pointcut*, desde que o programador siga a convenção de iniciar o método por "set". Se o método, no entanto, for chamado, por exemplo, de `change()`, ele não será incluído no *pointcut*. Em nenhum dos casos, o *pointcut* transmite ou implementa diretamente a intenção do programador, que era capturar todas as mudanças de estado das figuras.Este problema, embora pareça específico, na verdade representa uma classe ampla de situações às quais a programação orientada a aspectos pode ser aplicada. Genericamente, podemos descrever essa classe como aquela dos problemas em que é necessário tomar ou não uma ação com base nas ações efetuadas em certo fluxo de controle.

Isso pode incluir a persistência de um objeto se o seu estado foi alterado, o fechamento de uma conexão de rede caso ela tenha sido aberta, ou a negação de acesso a um método que poderia fazer uma alteração não autorizada no sistema. Este último caso difere um pouco pelo fato de se basear em atos no futuro, ou seja, o adendo tem que agir antes que as alterações em questão efetivamente ocorram. A seção 3.1.2 ilustra essa situação.

3.1.2 Segurança

O exemplo da listagem 3.2, extraído (com modificações) de [28], ilustra um aspecto de controle de acesso a uma lista de compras. A intenção do programador é impedir que um cliente altere a lista de outro.

Suponha agora que foi adicionado um novo recurso à aplicação de compras, o qual permite a um cliente compartilhar a sua cesta de compras com seus amigos, como uma "lista de desejos". Esta funcionalidade é implementada por um novo método na classe `ShoppingCart`, chamado `showItems()`, que permite listar o seu conteúdo. Da forma como está implementado o aspecto de autorização, a nova funcionalidade não vai funcionar corretamente, pois ele vai impedir o acesso do amigo à cesta, devido ao nome do método.

Para resolver o problema, seria necessário alterar o *pointcut*, passando a enumerar todos os métodos que alteram a cesta. Nesse caso, entretanto, se fosse criado um novo método de alteração da cesta que não estivesse listado no *pointcut*, a cesta estaria desprotegida. No caso do *pointcut* com definição via coringa ("*"), esse risco também existe, mas é menor se as convenções forem seguidas.

Listagem 3.2 Um aspecto de controle de acesso a uma lista de compras

```
public class ShoppingCart {

  private Set items;
  private double total;
  public Customer customer;

  ShoppingCart(Customer customer) {
    this.customer = customer;
    items = new HashSet();
    total = 0.0;
  }

  public void addItem(Integer itemNr) {
    items.add(itemNr);
    total += Database.loadPrice(itemNr);
  }

  public void removeItem(Integer itemNr) {
    items.remove(itemNr);
    total -= Database.loadPrice(itemNr);
  }

}

public abstract aspect ItemChanges {

  pointcut itemChanges(Customer c, ShoppingCart s) :
    this(c) && target(s) && call(* ShoppingCart.*Item(..));

}

public aspect Authorization extends ItemChanges {

  before (Customer c, ShoppingCart s) : itemChanges(c, s) {
    if (!mayAccess(c, s)) {
      throw new AccessException("Illegal Access - denied.");
    }
  }

  private boolean mayAccess(Customer c, ShoppingCart s) {
    return c.equals(s.customer);
  }

}
```

3.1.3 Aspectos para *Web Services*

Aqui mostramos um exemplo de como um modelo de *pointcuts* mais rico pode ser benéfico, mesmo em problemas cuja solução seria possível nas linguagens orientadas a aspectos convencionais.

Em [33], os autores propõem uma linguagem orientada a aspectos específica (uma DSL, "*domain specific language*") para o processamento de XML em *Web Services*. Essa linguagem, denominada Doxpects, permite a definição de *pointcuts* que especifiquem elementos de XML trocados em operações remotas com Web Services, para facilitar operações sobre esses elementos através de adendos. Essencialmente, a linguagem define dois novos tipos de *pointcuts* primitivos: *header* e *body*, correspondendo ao cabeçalho e ao conteúdo de uma mensagem SOAP. Estes *pointcuts* podem receber como parâmetro uma consulta XPath [8], que seleciona um conjunto de elementos em um documento XML.

Além destes dois *pointcuts*, a linguagem Doxpects define dois qualificadores para os adendos: *request* e *response*. Embora os autores façam analogia entre esses qualificadores e os modificadores *before* e *after* do AspectJ, podemos considerar um qualificador *request* ou *response* como parte da definição do *pointcut*, pois a linguagem define que o adendo será executado sempre antes do envio da requisição ou recebimento de resposta. Esse qualificador, po rtanto, na verdade determina se será em um caso ou no outro.

Na listagem 3.3, vemos um exemplo de um *pointcut* na linguagem Doxpects.

Listagem 3.3 Um *pointcut* na linguagem Doxpects

```
request each(Address anAddress) : body(//Address, anAddress)
```

Esse *pointcut* seleciona todos os elementos XML do tipo "Address" presentes em mensagens enviadas como requisição (isso é feito pela da consulta XPath "`//Address`"). A seguir, cria objetos do tipo `Address` com as informações desses elementos e torna esses objetos disponíveis através da variável `anAddress`.

A implementação desse tipo de aspecto em uma linguagem orientada a aspectos tradicional seria mais complicada. Ela poderia depender de especificidades do arcabouço utilizado para a implementação dos *Web Services*, pois as linguagens de *pointcuts* tradicionais lidam com níveis mais baixos de abstração. Talvez mais importante do que isso é a elevação do nível de abstração que essa abordagem traz: o programador não precisa mais pensar nos métodos que implementam o protocolo

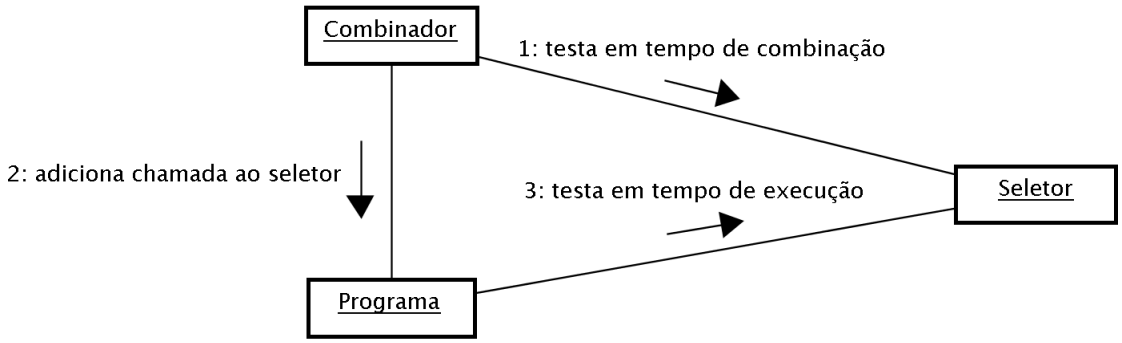

Figura 3.2: O funcionamento de um seletor

em um determinado arcabouço, mas sim nas mensagens que compõem esse protocolo.

3.2 Conceito

Como forma de melhorar a qualidade dos *pointcuts*, este trabalho propõe um novo mecanismo de extensão às linguagens de definição de *pointcuts*. Para explicar esse mecanismo, consideramos importante definir um conceito que chamamos de seletor de pontos de junção.

Um seletor de pontos de junção é uma função que, dado um ponto de junção e um conjunto de parâmetros, determina se o ponto de junção deve fazer parte de um *pointcut*. Os seletores de pontos de junção podem agir de duas formas: em tempo de combinação e em tempo de execução. O componente do seletor que age em tempo de combinação pode determinar que seja chamado também o componente de tempo de execução, caso a informação disponível em tempo de combinação não seja suficiente para completar a decisão. A figura 3.2 mostra o funcionamento básico de um seletor.

De certa forma, as linguagens e arcabouços orientados a aspectos atuais já possuem seletores de pontos de junção (como “call”, “execution” etc.), embora não os chamem por esse nome. Entretanto, o programador não pode definir novos seletores, pois os algoritmos de seleção de pontos de junção estão pré-definidos nos combinadores.

Consideramos importante a utilização da nova nomenclatura proposta, por deixar clara a distinção entre o algoritmo de seleção e as expressões que o utilizam. Estas últimas são os *pointcuts* propriamente ditos. O exemplo abaixo ilustra o conceito:

`call` é um seletor.

`call(void *->setSize(..))` é um *pointcut*.

As características principais que definem seletores de pontos de junção e os distinguem de mecanismos similares são as seguintes:

1. **Eles são parametrizáveis.** Quando usado em uma expressão *pointcut*, um seletor pode receber argumentos que são levados em conta por seu algoritmo na tomada de decisão.

2. **Eles podem ser combinados por expressões simples.** Seletores podem ser combinados por expressões booleanas simples para criar uma definição de *pointcut*.

3. **Eles operam tanto em tempo de combinação quanto em tempo de execução.** Um mecanismo simples e uniforme é utilizado para permitir que seletores utilizem informações de tempo de combinação, de execução, ou ambos, e complete a decisão no momento mais adequado.

Estas características fazem dos seletores de pontos de junção uma unidade básica de funcionalidade. Em expressões *pointcut*, eles desempenham papel análogo ao de métodos em objetos e adendos em aspectos.

Algumas destas características são encontradas em outros mecanismos, como os que serão vistos no Capítulo 6, mas a combinação de todas elas faz os seletores de pontos de junção mais expressivos e fáceis de usar.

Capítulo 4

Uma implementação de seletores de pontos de junção

Como prova de conceito, implementamos a funcionalidade de seletores de pontos de junção como uma extensão ao arcabouço orientado a aspectos JBoss AOP [19]. A escolha do JBoss AOP como base para o trabalho deveu-se a um conjunto de fatores práticos, não conceituais. Ou seja, a aplicação da proposta deste trabalho não está condicionada a esse arcabouço específico, e poderia ocorrer em qualquer outro arcabouço ou linguagem orientada a aspectos que possua conceitos semelhantes.

Um dos fatores para a escolha do JBoss AOP foi a maior facilidade de estender um arcabouço, em comparação com uma linguagem de programação. Foi necessário alterar levemente a sub-linguagem de definição de *pointcuts* do arcabouço, mas esta é uma linguagem de expressões simples e interpretada pelo próprio arcabouço. A alteração não gera interferências sobre o compilador da linguagem base, o que aconteceria em outras abordagens, como o AspectJ [1], que funcionam como extensões da linguagem base. Além disso, já havia conhecimento acumulado sobre esse arcabouço dentro do nosso grupo de pesquisa.

O mecanismo aqui descrito permite que se criem novos seletores, mas a implementação original dos seletores já existentes no JBoss AOP (como `call`, `within` etc.) foi mantida. A extensão ao arcabouço, do ponto de vista do usuário, foi feita em duas partes. Primeiro, foi modificada a linguagem de definição de *pointcuts* do JBoss AOP para que sejam reconhecidas chamadas a seletores. A seguir, foi definida uma nova sintaxe para declaração de seletores. Essa sintaxe tem duas representações: XML e anotações do Java 5.

Listagem 4.1 A interface `Selector`

```
public interface Selector {

        void setName(String name);

        String getName();

        /* Weaving-time selector methods */

        SelectionValue matchesExecution(Advisor advisor, CtMethod m,
                        List<SelectorParam> selectorParams);

        SelectionValue matchesExecution(Advisor advisor, CtConstructor c,
                        List<SelectorParam> selectorParams);

        SelectionValue matchesConstruction(Advisor advisor, CtConstructor c,
                        List<SelectorParam> selectorParams);

        SelectionValue matchesCall(Advisor callingAdvisor, MethodCall methodCall,
                        List<SelectorParam> selectorParams);

        SelectionValue matchesCall(Advisor callingAdvisor, NewExpr methodCall,
                        List<SelectorParam> selectorParams);

        SelectionValue matchesGet(Advisor advisor, CtField f,
                        List<SelectorParam> selectorParams);

        SelectionValue matchesSet(Advisor advisor, CtField f,
                        List<SelectorParam> selectorParams);

        /* Run-time selector methods */

        boolean matchesExecution(Advisor advisor, Method m, Object target,
                        Object[] args, List<SelectorParam> selectorParams);

        boolean matchesExecution(Advisor advisor, Constructor c, Object target,
                        Object[] args, List<SelectorParam> selectorParams);

        boolean matchesConstruction(Advisor advisor, Constructor c,
                        List<SelectorParam> selectorParams);

        boolean matchesCall(Advisor advisor, AccessibleObject within,
                        Class calledClass, Method calledMethod, Object target,
                        Object[] args, List<SelectorParam> selectorParams);

        boolean matchesCall(Advisor advisor, AccessibleObject within,
                        Class calledClass, Constructor calledCon,
                        Object[] args, List<SelectorParam> selectorParams);

        boolean matchesGet(Advisor advisor, Field f, Object target,
                        List<SelectorParam> selectorParams);

        boolean matchesSet(Advisor advisor, Field f, Object target, Object value,
                        List<SelectorParam> selectorParams);

}
```

4.1 Visão do programador de seletores

Para um usuário de seletores (ou seja, um programador de *pointcuts*), um seletor comporta-se como uma função sobre um ponto de junção. Entretanto, do ponto de vista de quem programa um seletor, ele é uma classe Java que implementa a interface `org.jboss.aop.selector.Selector` (mostrada na listagem 4.1). Esta interface foi criada com base na interface `org.jboss.aop.pointcut.Pointcut`, que define a implementação de um pointcut no JBoss AOP. Assim, foi possível integrar a funcionalidade necessária para o suporte a seletores de forma consistente e sem grandes alterações em relação ao funcionamento do arcabouço.

Há dois conjuntos de métodos na interface do seletor. Cada um desses conjuntos contém métodos para tratar todos os tipos de pontos de junção primitivos (chamadas, acesso a campos etc.) oferecidos pelo arcabouço[1]. Assim, um único seletor pode se aplicar a pontos de junção de diversos tipos, se isso for necessário.

Listagem 4.2 O tipo `SelectionValue`

```
public enum SelectionValue {
        TRUE,               // Matches
        FALSE,              // Does not match
        CHECK_AT_RUNTIME    // Needs runtime information to decide
}
```

O primeiro conjunto de métodos contém aqueles que analisam pontos de junção em tempo de combinação de aspectos. Esses métodos devolvem valores do tipo `SelectionValue`, cuja declaração é mostrada na listagem 4.2. Esse tipo enumerado define três constantes. A primeira (`TRUE`) deve ser devolvida pelo método caso o ponto de junção examinado satisfaça os critérios do seletor. A segunda (`FALSE`) indica o contrário, que o ponto de junção não deve pertencer ao *pointcut*. A terceira constante (`CHECK_AT_RUNTIME`) indica que o arcabouço deve incluir no ponto de junção uma chamada a um método de checagem em tempo de execução do seletor.

O segundo conjunto de métodos faz o processamento de pontos de junção em tempo de execução. Caso uma chamada feita a um método do primeiro grupo determine que a seleção do ponto de junção só pode ser decidida com informação disponível em tempo de execução, o método correspondente do segundo grupo será chamado em tempo de execução.

[1] Embora haja métodos para todos os tipos de pontos de junção na interface, na maioria dos casos apenas alguns deles são implementados. Há vantagens e desvantagens nessa abordagem, mas a principal motivação foi manter a citada coerência com o JBoss AOP.

4.1.1 Meta-informação

Para que o pré-compilador do JBoss-AOP reconheça uma classe como sendo um seletor, é preciso declarar este fato através de meta-informação, o que pode ser feito de duas formas: com um arquivo XML ou com anotação do Java 5. Estas duas formas são oferecidas por todas as outras funcionalidades do JBoss AOP, por isso as adotamos também na definição de seletores.

A listagem 4.3 mostra um exemplo de declaração no estilo anotação, enquanto que uma versão do mesmo *pointcut* em notação XML é mostrada na listagem 4.4.

Listagem 4.3 Definição de um seletor através de anotação

```
import org.jboss.aop.selector.SelectorDef;
import org.jboss.aop.selector.Selector;
...

@SelectorDef(name="parameterTypeIs")
public class ParameterTypeSelector extends SelectorBase {
...
```

Listagem 4.4 Definição de um seletor através de XML

```
<selector
        name="parameterTypeIs"
        class="org.jboss.test.aop.selector.examples.ParameterTypeSelector"/>
```

Note que a declaração via anotação é feita na própria classe que implementa o seletor, enquanto que a opção XML exige o uso de um arquivo separado. Este comportamento é idêntico ao funcionamento original do JBoss AOP. No caso da declaração via XML, a classe que implementa o seletor seria idêntica, exceto pela ausência da anotação inicial.

Em ambos os casos, existe um atributo chamado `name`. Este atributo determina o nome pelo qual o seletor será referenciado internamente e em expressões *pointcut*. No caso da declaração em XML, é preciso declarar também o nome da classe que implementa o seletor.

Para facilitar a implementação de seletores, existe uma classe auxiliar, `SelectorBase`, que provê uma implementação padrão de todos os métodos da interface, devolvendo `SelectionValue.FALSE` ou `false` em todos eles. Ao estender esta classe, o programador só precisa implementar os métodos que tratam os tipos de pontos de junção relevantes para o seu seletor.

4.1.2 Uso de um seletor

Um seletor é utilizado para formar cláusulas em uma expressão *pointcut*, assim como as cláusulas fixas na linguagem (como `call` e `within`, por exemplo). Diversos seletores podem ser livremente combinados através de expressões booleanas.

Nas listagens 4.5 e 4.6, vemos como utilizar o seletor definido na seção anterior utilizando anotações e XML, respectivamente. Nos exemplos, estamos selecionando todas as execuções de métodos da classe `BasePOJO` (trecho `execution(...)`) cujo primeiro parâmetro seja do tipo `Integer` (trecho `parameterTypeIs(...)`).

Listagem 4.5 Uso de um seletor através de anotação

```
@Bind(pointcut =
  " execution(* org.jboss.test.aop.selector.basic.BasePOJO->*(..)) " +
  " AND parameterTypeIs(\"0\",\"java.lang.Integer\") ")
public Object advice(Invocation invocation) throws Throwable {
  ...
```

Listagem 4.6 Uso de um seletor através de XML

```
<bind pointcut=
        "execution(* org.jboss.test.aop.selector.basic.BasePOJO->*(..))
         AND parameterTypeIs(&quot;0&quot;, &quot;java.lang.Integer&quot;)">
  <advice name="advice" aspect="TestAspect"/>
</bind >
```

Nesta prova de conceito, os parâmetros que um seletor pode receber devem ser cadeias de caracteres. Em uma implementação futura, poderia ser adicionado suporte a outros tipos de parâmetros. Note que um parâmetro de um seletor aparece dentro de um atributo de uma anotação ou elemento XML. Como tanto o atributo como o parâmetro (do tipo cadeia de caracteres) são delimitados por aspas, é necessário que as "aspas internas" sejam expressas de modo codificado, ou seja, como "`\"`" nas anotações e "`&quot;`" no XML.

4.2 Visão interna

As principais alterações para a inclusão da funcionalidade de seletores no JBoss AOP foram feitas nos mecanismos de instrumentação de código e interpretação de pointcuts.

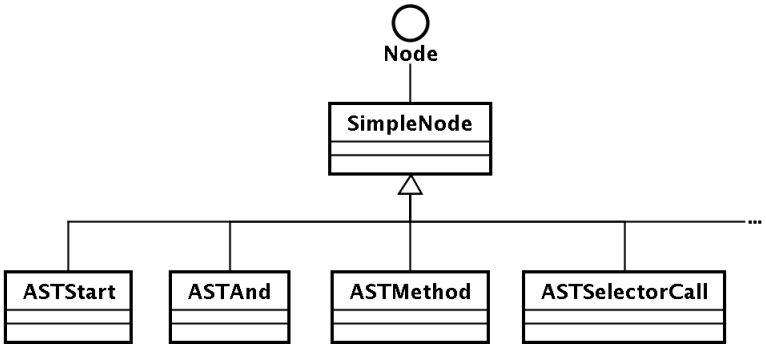

Figura 4.1: Classes que representam a árvore sintática da linguagem de *pointcuts*

4.2.1 Mecanismo de interpretação

Começamos por descrever o mecanismo de interpretação da linguagem de *pointcuts* e as modificações feitas nele para a implementação de seletores.

O JBoss AOP possui uma linguagem própria para a definição de expressões *pointcut*. Esta linguagem é definida em uma gramática, no formato do gerador de analisadores sintáticos JavaCC [18]. É utilizado também o pré-processador JJTree [20], que gera classes representando os elementos da árvore sintática da linguagem.

A Listagem 4.7 mostra as alterações (em negrito) feitas nessa gramática para incluir na linguagem de *pointcuts* o mecanismo de chamada a seletores. Um novo elemento sintático chamado `SelectorCall` foi criado para representar a chamada de um seletor em um *pointcut*. Este elemento consiste de um identificador (o nome do seletor), cujo formato é o mesmo usado para *pointcuts* nomeados, e uma lista de argumentos, cada um deles representado por um elemento sintático que recebeu o nome de `SelectorParam`. Os valores dos argumentos, por sua vez, são codificados como cadeias (*strings*) literais. A sintaxe para definição dessas cadeias foi também incluída na gramática, e é idêntica à utilizada pela linguagem Java.

A figura 4.1 mostra (parcialmente) a hierarquia de classes gerada pelo JJTree para representar a estrutura de uma expressão pointcut, já com a inclusão da classe que representa uma chamada a um seletor (`ASTSelectorCall`). Estas classes refletem a forma como os elementos sintáticos são agrupados na gramática e usadas no processo de interpretação das expressões *pointcut*.

Listagem 4.7 Adição de seletores à gramática da linguagem de *pointcuts* do JBoss AOP

```
...

| < WITHINCODE: "withincode(" > : BEHAVIOR
| < SELECTOR: <SELECTOR_IDENTIFIER> (<SELECTOR_DOT> <SELECTOR_IDENTIFIER>)*>
| < SELECTOR_IDENTIFIER: (<SELECTOR_WILD_LETTER>)+ >
| < #SELECTOR_WILD_LETTER: ["_","a"-"z","A"-"Z","0"-"9"] >
| < #SELECTOR_DOT: ["."] >
| < POINTCUT: <POINTCUT_IDENTIFIER> (<POINTCUT_DOT> <POINTCUT_IDENTIFIER>)*>
| < POINTCUT_IDENTIFIER: (<POINTCUT_WILD_LETTER>)+ >
| < #POINTCUT_WILD_LETTER: ["_","a"-"z","A"-"Z","0"-"9"] >
| < #POINTCUT_DOT: ["."] >
| <NOT: "!">
| < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

...

void Concrete() #void : {}
{
   ( LOOKAHEAD(4) Call() | LOOKAHEAD(4) Within() | LOOKAHEAD(4) Withincode()
     | LOOKAHEAD(4) Execution() | LOOKAHEAD(4) Construction()
     | LOOKAHEAD(4) Set() | LOOKAHEAD(4) Get()
     | LOOKAHEAD(4) FieldExecution() | LOOKAHEAD(4)SelectorCall()
     | LOOKAHEAD(4) Pointcut() | LOOKAHEAD(4) All()| LOOKAHEAD(4) Has()
     | LOOKAHEAD(4) HasField())
}

void Pointcut() #Pointcut :
{
  Token pointcut;
}
{
   pointcut=<POINTCUT>
   {
      jjtThis.setPointcutName(pointcut.image);
   }
}

void SelectorCall() #SelectorCall :
{
  Token selector;
}
{
   selector=<SELECTOR> "(" ( SelectorParams() )? ")"
   {
      jjtThis.setSelectorName(selector.image);
   }
}

void SelectorParams() #SelectorParams :
{}
{
    SelectorParam() ( "," SelectorParam() )*
}

void SelectorParam() #SelectorParam :
{
  Token value;
}
{
    value=<STRING_LITERAL>
   {
      jjtThis.setValue(value.image);
   }
}

...
```

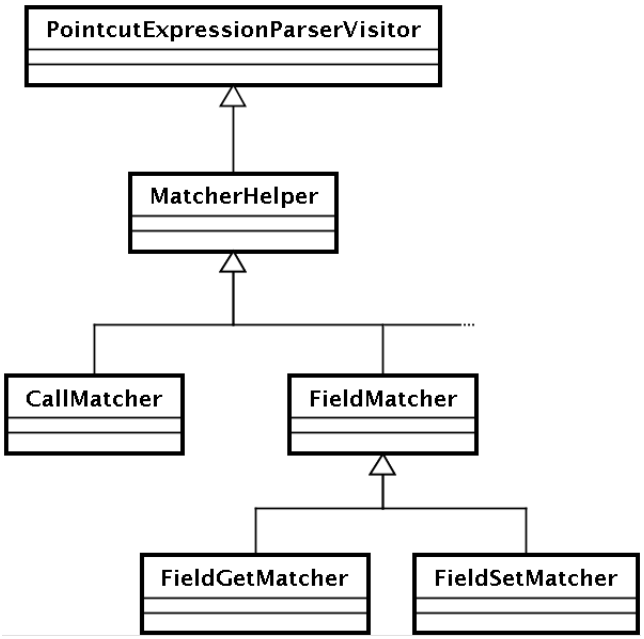

Figura 4.2: A estrutura de *matchers* do JBoss AOP

Além dos elementos da árvore sintática, o JJTree gera uma interface chamada `Pointcut-ExpressionParserVisitor`, que representa um visitante (padrão *Visitor* [12]) desta árvore. No JBoss AOP, esta interface é implementada pela classe `MatcherHelper`, que por sua vez serve de base a todas as classes que fazem casamento (*match*) entre *pointcuts* e pontos de junção. A figura 4.2 ilustra (parcialmente) a hierarquia de *matchers* do JBoss AOP.

Cada um destes *matchers* já possuía um método chamado `resolvePointcut()`. Este método é chamado sempre que é encontrada uma referência a um *pointcut* nomeado em uma expressão *pointcut*, e tem uma implementação específica em cada *matcher*, que chama o método adequado da interface `Pointcut`. De forma análoga, foi criado o método `resolveSelectorCall()`, cuja implementação específica em cada *matcher* chama o método adequado da interface `Selector`. A Listagem 4.8 exemplifica a implementação do método `resolveSelectorCall()` para um *matcher* de execução de métodos. Devido ao tipo devolvido por este método (`SelectionValue`), toda a cadeia de *matchers* teve que ser alterada para lidar com este tipo, já que ele é propagado até a raiz da árvore de interpretação. Anteriormente, o tipo devolvido pela interpretação de um *pointcut* era um booleano,

já que não existia a possibilidade de delegar uma parte da decisão para uma etapa posterior.

Listagem 4.8 Resolução de uma chamada a um seletor em um *matcher* de execução de método

```
protected SelectionValue resolveSelectorCall(Selector s, List<SelectorParam> params) {
    return s.matchesExecution(advisor, ctMethod, params);
}
```

4.2.2 O processo de combinação

No JBoss AOP, as chamadas aos adendos de um ponto de junção não são inseridas diretamente no código da classe, pois elas são dinâmicas e podem ser modificadas em tempo de execução. Cada um desses pontos de junção possui uma pilha de interceptadores, que é populada em tempo de execução, em geral no momento em que as classes são carregadas em memória pela máquina virtual Java. Por isso, o processo de combinação de aspectos no JBoss AOP é dividido em duas etapas, que detalhamos a seguir.

Instrumentação

Na etapa de instrumentação, as classes potencialmente afetadas por aspectos são transformadas para a inclusão de código que chama a pilha de interceptadores nos pontos relevantes. Por exemplo, se existir um adendo para a execução de um método, o código original do método será substituído por outro que chama a pilha de interceptadores. O código original é colocado em um novo método, que faz parte da pilha de interceptadores, de forma que ele será executado (ou não) no momento determinado pelos adendos. Também é incluído, na classe, o código que iniciará a instanciação de um objeto *Advisor* associado a ela.

A etapa de instrumentação pode ser executada independentemente, modificando os arquivos das classes compiladas, ou no momento em que as classes são carregadas em memória, em tempo de execução.

Para determinar as classes e elementos que devem ser instrumentados, os *pointcuts* configurados são testados. Caso haja seletores nas expressões *pointcut*, serão chamados os métodos de tempo de combinação desses seletores.

Carregamento

O carregamento dos interceptadores na pilha de interceptação de uma classe previamente instrumentada ocorre quando ela é carregada em memória pela máquina virtual Java. O processo consiste na inserção dos devidos interceptadores na pilha de interceptação dessas classes, e é disparado por chamadas que foram inseridas no código da classe durante a instrumentação. A arquitetura do JBoss AOP também permite que sejam adicionados interceptadores e adendos durante a execução do programa, recurso conhecido como "POA dinâmica".

Durante o carregamento, os *pointcuts* são reavaliados para determinar quais interceptadores serão inseridos na pilha. Embora esta avaliação seja normalmente redundante em relação à feita na fase de instrumentação, ela é necessária devido ao modelo dinâmico do JBoss AOP, em que os adendos são definidos e podem ser trocados em tempo de execução. No caso dos *pointcuts* com seletores, são chamados na etapa de carregamento os mesmos métodos da etapa de combinação.

Caso um *pointcut* com seletores devolva o valor `CHECK_AT_RUNTIME`, será inserido na pilha de interceptação um interceptador especial, que vai avaliar a parte dinâmica do seletor para determinar se os adendos devem ser executados.

Execução

Quando um ponto de junção instrumentado é atingido, o JBoss AOP chama o primeiro interceptador da pilha associada ao ponto. Esse interceptador será responsável por chamar ou não o próximo interceptador da pilha. Terminando a pilha de interceptadores, o código original do ponto de junção é executado.

O interceptador para avaliação de seletores em tempo de execução, quando chamado, obtém um *matcher* adequado para o tipo de ponto de junção que está avaliando e o *pointcut* associado a si. A listagem 4.9 mostra um trecho desta classe, com o método que toma essa decisão.

4.3 Desempenho

O mecanismo de seletores, da forma como foi implementado, não causa por si só nenhum impacto significativo no desempenho do processo de combinação ou de carregamento de aspectos. O JBoss

Listagem 4.9 O interceptador para avaliação de seletores em tempo de execução

```
public class SelectorInterceptor implements Interceptor {

   // Chain of advice interceptors
   Interceptor[] adviceChain;

   // Pointcut for runtime evaluation
   Pointcut pointcut;

   /**
    * Handles an invocation, by testing the appropriate runtime selector and
    * invoking the advice chain if it matches, or the original code otherwise.
    *
    * @param joinPoint represents the original code of the current join point
    */
   public Object invoke(Invocation joinPoint) throws Throwable {

       if (getMatcher(joinPoint).matches(pointcut)) {
           return joinPoint.getWrapper(adviceChain).invokeNext();
       } else {
           return joinPoint.invokeNext();
       }
   }

...
```

AOP já percorre todos os pontos de junção, testando todos os *pointcuts* em cada um deles[2], e esse comportamento não foi alterado. Só haverá algum impacto de desempenho se o programador utilizar *pointcuts* com seletores. Este impacto será proporcional ao desempenho da implementação do seletor e ao número de pontos de junção testados por ele.

Os seletores podem ser combinados com seletores mais simples para restringir o número de classes testadas. Por exemplo, pode-se usar o seletor básico `within` para restringir a busca às classes localizadas em um pacote específico, em combinação com um seletor mais pesado. Neste caso, a ordem dos seletores na expressão *pointcut* é importante, pois a avaliação dos operadores booleanos é feita com curto-circuito, ou seja, se a primeira parte da expressão for suficiente para determinar o seu resultado, a segunda não será avaliada.

[2]Esta é uma limitação de desempenho da versão atual do JBoss AOP. Caso este mecanismo seja otimizado no futuro, os benefícios também se aplicarão aos *pointcuts* com seletores.

Por exemplo, considere os seguintes *pointcuts*:

1. `within(com.acme.*) AND getter()`
2. `getter() AND within(domain.*)`

Quando um método pertencente a uma classe que não faça parte do pacote `com.acme` for examinado para o *pointcut* 1, apenas o seletor `within` será testado, pois o fato de que ele devolve o valor `FALSE` é suficiente para determinar que a expressão inteira é falsa. No caso do *pointcut* 2, o seletor testado será o `getter`. Se o seletor `getter` for mais demorado que o `within`, o *pointcut* 2 será mais lento. Portanto, o impacto dos seletores sobre o desempenho da combinação pode ser controlado colocando-se sempre os seletores mais leves no início da expressão.

Capítulo 5

Aplicações

Este capítulo exibe alguns exemplos de como a nossa proposta pode melhorar a qualidade dos *pointcuts*. Nas listagens de código, alguns detalhes como tratamento de exceções e importação de classes foram omitidos para maior clareza.

5.1 Tipo de parâmetro

Este exemplo foi tirado do trabalho [7] (com uma pequena generalização). Ele consiste na implementação de um seletor que identifica métodos cujo parâmetro em uma certa posição seja compatível com um certo tipo, o qual também é especificado como argumento ao seletor.

A finalidade deste exemplo é simplesmente mostrar o uso mais simples possível do conceito de seletores, resolvendo um problema que não teria solução elegante nas linguagens orientadas a aspectos convencionais, embora não tenha aplicação prática imediata. Nossa solução é ligeiramente diferente da apresentada em [7] e, em nossa opinião, mais simples e elegante, devido à separação entre as partes estática e dinâmica do seletor. A listagem 5.1 mostra como seria declarado esse seletor.

O primeiro método é chamado em tempo de combinação e faz os testes possíveis com a informação disponível no momento. Ele pode selecionar ou descartar o ponto de junção imediatamente. Entretanto, se isso não for possível, ele devolve um valor que faz o combinador do arcabouço injetar no programa uma chamada ao segundo método do seletor. Essa chamada ocorrerá em tempo de execução, quando toda a informação necessária estiver disponível. Desta forma, só haverá o custo adicional da checagem em tempo de execução se ela for inevitável.

Nas listagens 4.5 e 4.6 (seção 4.1.2), foram apresentados exemplos de uso desse seletor.

Listagem 5.1 Um seletor por tipo de parâmetro

```
@SelectorDef(name="parameterTypeIs")
public class ParameterTypeSelector extends SelectorBase {

    /**
     * Weave-time method execution selector
     */
    public SelectionValue matchesExecution(Advisor advisor, CtMethod m,
            List<SelectorParam> params) {

        int paramIndex = Integer.parseInt(params.get(0).getValue());
        String paramTypeName = params.get(1).getValue();

        // Obtains type of wanted method parameter
        CtClass parType = m.getParameterTypes()[paramIndex];

        // Obtains CtClass whose name is the first selector parameter
        CtClass argType = ClassPool.getDefault().get(paramTypeName);

        // Tests for compatibility between types. The rule is the same as
        // for Java type casts.
        if (parType.subtypeOf(argType)) {
                return TRUE;
        } else if (argType.subtypeOf(parType)) {
                return CHECK_AT_RUNTIME;
        } else {
                return FALSE;
        }

    }

    /**
     * Run-time method execution selector
     */
    public boolean matchesExecution(Advisor advisor, Method m, Object target,
            Object[] args, List<SelectorParam> params) {

        int paramIndex = Integer.parseInt(params.get(0).getValue());
        String paramTypeName = params.get(1).getValue();

        // Obtains Class object for the wanted type
        Class argType = Class.forName(paramTypeName);

        Object arg = args[paramIndex];
        if (arg == null) {
            return true; // null matches any type
        } else {
            Class parType = arg.getClass();

            // Tests for runtime compatibility between types.
            return (argType.isAssignableFrom(parType));
        }
    }

}
```

5.2 Editor de figuras

Como vimos na seção 3.1.1, o exemplo mais clássico utilizado em explicações didáticas da programação orientada a aspectos, o do editor de figuras, não tem uma solução satisfatória nas linguagens orientadas a aspectos convencionais. A solução mais comum se baseia no uso de padrões de nomenclatura, o que torna os *pointcuts* frágeis e propensos a erros.

Aqui, propomos uma solução para o exemplo do editor de figuras com o uso de um seletor específico. O novo seletor determina se um método atualiza algum campo lido pelo método cujo nome é fornecido como parâmetro.

Na listagem 5.2, vemos a declaração deste seletor. Os métodos privados `getFieldsUpdatedBy-Method` e `getFieldsReadByMethod`, não mostrados, fazem a busca recursiva dos campos alterados e lidos por um método, respectivamente. Nas buscas recursivas devido a chamadas a outros métodos, deve-se levar em conta também as subclasses dos tipos cujos métodos são chamados. O uso desse seletor em um *pointcut* é bastante simples, como vemos na listagem 5.3.

Esta solução garante que todos os métodos que alteram o estado lido pelo método fornecido serão devidamente selecionados. Ela é imune a erros do programador e a mudanças de nomenclatura do programa.

Do ponto de vista da qualidade de *pointcuts*, esta solução é melhor do que as apresentadas na seção 3.1.1 (sem seletores) porque:

- Tem maior resistência a mudanças, por não depender de convenções de nomenclatura;
- Sua definição é bem mais clara que a original, baseando-se no comportamento que se deseja capturar.

O exemplo da seção 3.1.2 (segurança) também poderia ser resolvido por um seletor semelhante ao apresentado aqui.

5.3 Chamadas por reflexão

A chamada de métodos através de técnicas de reflexão é um mecanismo cada vez mais utilizado atualmente, principalmente em arcabouços e *middleware*. Atualmente, as linguagens orientadas a aspectos não oferecem uma forma simples de se inserir adendos no lado do chamador para esse tipo

Listagem 5.2 Um seletor para métodos que atualizam campos lidos por outro método

```
@SelectorDef(name="updatesStateReadBy")
public class StateUpdaterSelector extends SelectorBase {

        /**
         * Analyses compile-time method executions.
         */
        public SelectionValue matchesExecution(Advisor advisor, CtMethod m,
                        List<SelectorParam> params) {

                // Gets selector parameters
                String readerTypeName = params.get(0).getValue();
                String readerMethodName = params.get(1).getValue();

                // Obtains the reader method
                CtClass readerType;
                CtMethod readerMethod;
                readerType = ClassPool.getDefault().get(readerTypeName);
                readerMethod = readerType.getDeclaredMethod(readerMethodName);

                // Gets the sets of read and updated fields
                Set<CtField> readFields = getFieldsReadByMethod(readerMethod);
                Set<CtField> updatedFields = getFieldsUpdatedByMethod(m);

                // Compares sets
                boolean result = readFields.removeAll(updatedFields);

                return (result ? TRUE : FALSE);

        }

        /**
         * Finds all fields updated by the given method, including those made in
         * calls to other methods (serches recursively).
         *
         * @param m the method to examine.
         * @return a set of fields updated by m.
         */
        private Set<CtField> getFieldsUpdatedByMethod(CtMethod m) {
                ...
        }

        /**
         * Finds all fields read by the given method, including those made in
         * calls to other methods (serches recursively).
         *
         * @param m the method to examine.
         * @return a set of fields read by m.
         */
        private Set<CtField> getFieldsReadByMethod(CtMethod m) {
                ...
        }

}
```

Listagem 5.3 Uso do seletor `updatesStateReadBy`

```
updatesStateReadBy("Display","redraw")
```

de chamada, o que pode ser importante nos casos em que não é possível instrumentar o código do método chamado.

O JBoss AOP possui um aspecto para o tratamento deste tipo de chamada. Entretanto, cremos que uma solução baseada inteiramente em *pointcuts* seria mais natural, por analogia direta às chamadas convencionais, e por evitar a complexidade adicional de um aspecto específico. Por isso, propomos aqui um seletor específico para o tratamento desse tipo de caso. Sua implementação encontra-se na listagem 5.4.

Este seletor implementa os métodos da interface `Selector` que tratam pontos de junção de chamada de métodos, tanto em tempo de combinação quanto de execução. Ele funciona da seguinte forma:

1. Em tempo de combinação, todas as chamadas ao método `invoke()` da classe `java.lang.reflect.Method` (o qual executa a chamada de um método por reflexão) são marcadas para checagem em tempo de execução;
2. Em tempo de execução, essas chamadas são interceptadas. Caso o nome e classe do método chamado sejam iguais aos desejados, o seletor devolve o valor `true` para que o adendo possa ser executado (se as outras condições do *pointcut* forem satisfeitas).

Nesta versão, por simplicidade, não estamos levando em conta os argumentos do método na comparação.

5.4 Aspectos para *Web Services*

Na seção 3.1.3 nos referimos à linguagem orientada a aspectos Doxpects, específica para o domínio da manipulação de documentos XML trocados como mensagens em *Web Services*.

Acreditamos que, em muitos casos, é possível evitar a necessidade de linguagens de programação específicas de domínios através do uso de seletores. Esboçamos aqui uma solução semelhante à linguagem Doxpects, mas apenas com o uso de seletores.

Listagem 5.4 Um seletor de chamadas de métodos via reflexão

```
@SelectorDef(name="reflectiveCall")
public class ReflectiveCallSelector extends SelectorBase {

        // Holds a reference to the Method.invoke() method object.
        CtMethod invoke;

        public ReflectiveCallSelector() {
                invoke = ClassPool.getDefault()
                        .get("java.lang.reflect.Method")
                        .getDeclaredMethod("invoke");
        }

        /**
         * Weave-time method call selector
         */
        public SelectionValue matchesCall(Advisor callingAdvisor,
                        MethodCall methodCall, List<SelectorParam> params)
                        throws NotFoundException {

                // Checks whether the called method is
                // java.lang.method.invoke(). If so, the joinpont must be
                // checked at runtime, else it can be discarded.
                if (methodCall.getMethod().equals(invoke)) {
                        return CHECK_AT_RUNTIME;
                } else {
                        return FALSE;
                }

        }

        /**
         * Run-time method call selector
         */
        public boolean matchesCall(Advisor advisor, AccessibleObject within,
                        Class calledClass, Method calledMethod, Object target,
                        Object[] args, List<SelectorParam> selectorParams) {

                // Pre-process selector parameters
                String methodClassName = selectorParams.get(0).getValue();
                String methodName = selectorParams.get(1).getValue();

                Method targetMethod = (Method) target;

                String targetMethodClassName =
                        targetMethod.getDeclaringClass().getName();
                String targetMethodName =
                        targetMethod.getName();

                return (targetMethodClassName.equals(methodClassName) &&
                        targetMethodName.equals(methodName));

        }

}
```

A implementação de um desses seletores pode ser vista na listagem 5.5. Este seletor identifica se o documento fornecido em uma chamada a um *Web Service* contém os elementos especificados pela expressão *XPath* [8] fornecida.

Esta solução utiliza dois métodos não implementados. O primeiro método (`isWsRequestMethod`) é responsável por selecionar os métodos do arcabouço de *Web Services* que tratam as mensagens. O segundo (`getWsDocument`) obtém o documento XML que representa a mensagem, tendo como parâmetros o objeto de destino da chamada sendo interceptada e os seus argumentos. Esses dois métodos encapsulam o conhecimento específico sobre o arcabouço de *Web Services* em uso. Tal conhecimento é necessário para o funcionamento do seletor.

Com base nesse seletor, é possível criar *pointcuts* que interceptem mensagem contendo um elemento XML específico. Isso pode simplificar e tornar mais legíveis as implementações de aspectos que lidem com mensagens de sistemas com *Web Services*.

5.5 Seletores específicos para arcabouços

Um domínio pouco explorado, no qual acreditamos que os seletores podem ter grande utilidade, é o uso de meta-informações de arcabouços na definição de *pointcuts*. Em muitos casos, não é possível fazer uso da meta-informação em *pointcuts* sem o uso de seletores, como nos casos em que ela não faz parte do código-fonte. Em outros, isso seria possível, mas exigiria que o programador dos aspectos usasse conhecimentos da estrutura interna do arcabouço, o que complicaria seu trabalho e geraria acoplamento em excesso.

Como solução, propomos a criação de seletores específicos para arcabouços. Esses seletores permitiriam ao programador utilizar conceitos e meta-informação do arcabouço para a construção de *pointcuts*, de forma modularizada. Para que essa abordagem faça sentido do ponto de vista de encapsulamento, esses seletores seriam fornecidos pelo próprio arcabouço, em uma biblioteca de seletores, assim como já são fornecidas bibliotecas de classes e procedimentos.

Para exemplificar esta abordagem, implementamos um conjunto de seletores para o arcabouço de mapeamento objeto-relacional Hibernate [3]. Os seletores implementados foram os seguintes:

- **getter**: seleciona métodos utilizados para obter valores de atributos persistentes;
- **setter**: seleciona métodos utilizados para atribuir valores a atributos persistentes;

Listagem 5.5 Um seletor para elementos em um documento XML recebido como requisição por um *Web Service*

```
@SelectorDef(name = "request")
public class WsRequestSelector extends SelectorBase {

        /**
         * Weave-time method execution selector
         */
        public SelectionValue matchesCall(Advisor callingAdvisor,
                        MethodCall methodCall, List<SelectorParam> params)
                        throws NotFoundException {

                // Calls a method to check for the apropriate static joinpoint
                if (isWsRequestMethod(methodCall)) {
                        return CHECK_AT_RUNTIME;
                } else {
                        return FALSE;
                }

        }

        /**
         * Run-time method execution selector
         */
        public boolean matchesCall(Advisor advisor, AccessibleObject within,
                        Class calledClass, Method calledMethod, Object target,
                        Object[] args, List<SelectorParam> selectorParams) {

                // Gets selector parameters
                String xpathExpression = selectorParams.get(0).getValue();

                // Gets the XML Document
                Document docroot = getWsDocument(target, args);

                // Matches the document to the desired elements, given by the XPath in
                // the selector parameters
                XPath xpath = XPathFactory.newInstance().newXPath();

                NodeSet resultNodes = (NodeSet) xpath.evaluate(xpathExpression,
                                docroot, XPathConstants.NODESET);

                if (resultNodes != null && resultNodes.getLength() > 0) {
                        return true;
                } else {
                        return false;
                }

        }

}
```

- **idGetter**: seleciona métodos utilizados para obter valores de atributos persistentes que são identificadores (chaves primárias);
- **idSetter**: seleciona métodos utilizados para atribuir valores a atributos persistentes que são identificadores (chaves primárias);
- **withinPersistent**: seleciona classes que representam entidades persistentes.

Assim como todo seletor, eles podem ser combinados a outros seletores, que podem ser parte da linguagem de *pointcuts* ou implementados pelo mecanismo de seletores aqui apresentado. Com isso, podem-se criar expressões *pointcut* como a da listagem 5.6, que seleciona métodos de atribuição de valores a atributos persistentes do tipo `Timestamp`, e a da listagem 5.7, que seleciona execuções de métodos que seguem a convenção de obtenção de atributos mas não obtém atributos persistentes, e estão em classes de entidades persistentes.

Listagem 5.6 Um *pointcut* que seleciona execuções de métodos que atribuem valores a atributos persistentes do tipo Date

```
execution(* *->*(java.sql.Timestamp)) AND setter()
```

Listagem 5.7 Um *pointcut* que seleciona execuções de métodos que seguem a convenção de obtenção de atributos mas não obtém atributos persistentes, e estão em classes de entidades persistentes

```
execution(* *->get*(..)) AND withinPersistent() AND !getter() AND !idGetter()
```

Todos os seletores dessa biblioteca são classes que estendem uma classe base abstrata, chamada `HibernateSelector`. O método `matchesExecution` dessa classe busca os objetos que contém a meta-informação necessária e delega para subclasses as decisões específicas de cada seletor usando o padrão *template method* [12]. No caso dos seletores que lidam com métodos ligados a atributos, existe uma classe intermediária na hierarquia que faz a iteração sobre os atributos, novamente delegando as decisões às subclasses por *template methods*. A figura 5.1 mostra essa hierarquia de classes. O código que implementa o seletor *getter* encontra-se nas listagens 5.8, 5.9 e 5.10. Os outros seletores têm implementações análogas.

Com o uso de alguns desses seletores, podemos criar um exemplo prático: um aspecto para fazer conversão automática de fusos horários em uma aplicação. Os requisitos dessa aplicação são:

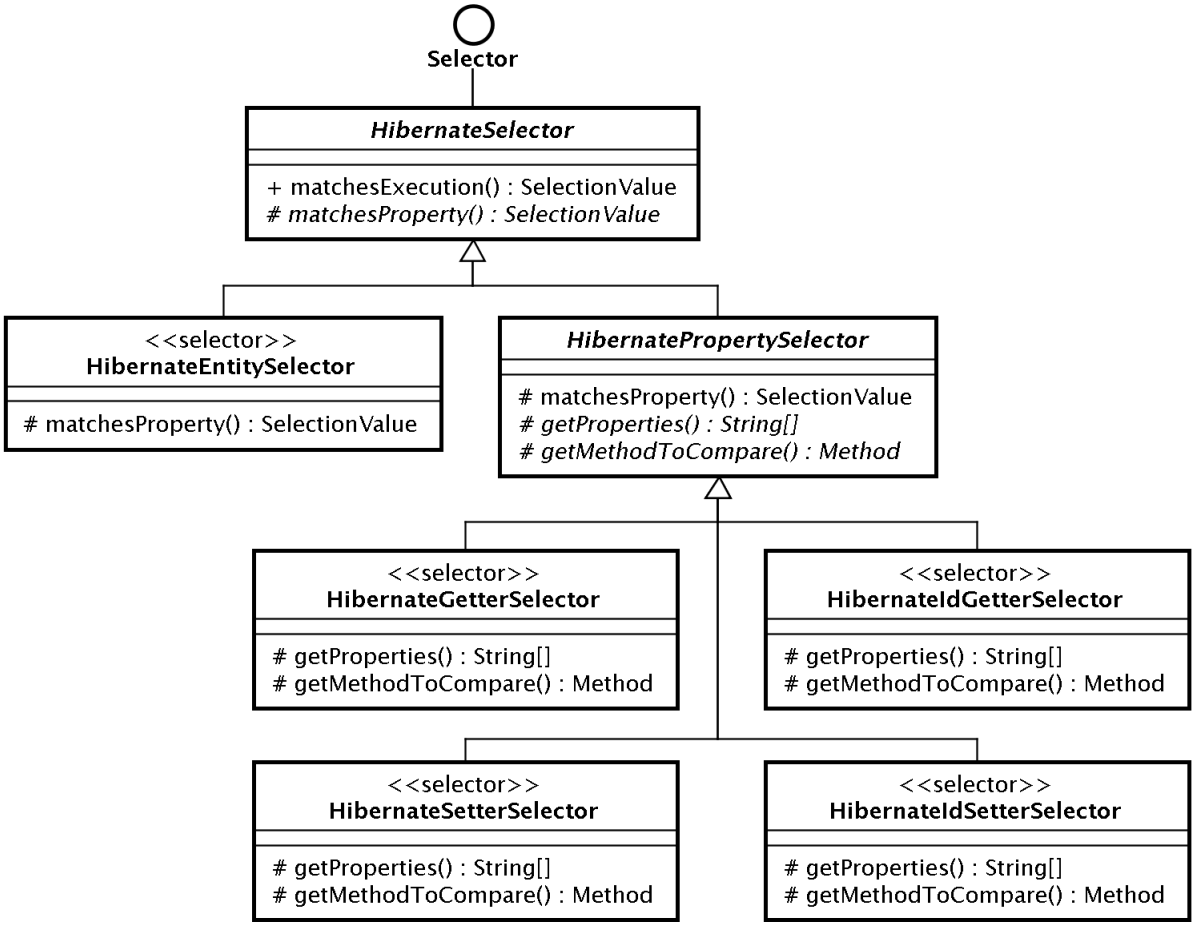

Figura 5.1: Hierarquia de classes da biblioteca de seletores do Hibernate

- Cada usuário tem um fuso horário associado;
- As operações e consultas de um usuário devem sempre receber e exibir as datas/horas em seu fuso;
- Os relatórios que mostram dados de diversos usuários devem exibir as datas/horas em um mesmo fuso de referência.

Para viabilizar esses requisitos, uma solução possível é armazenar todos os dados com as da-

Listagem 5.8 Classe base para todos os seletores do Hibernate

```
public abstract class HibernateSelector extends SelectorBase {

    private SessionFactory sessionFactory =
        new Configuration().configure().buildSessionFactory();

    public SelectionValue matchesExecution(Advisor advisor, CtMethod m,
            List<SelectorParam> selectorParams) {

        Class declaringClass = Class.forName(m.getDeclaringClass().getName());

        ClassMetadata metaData = sessionFactory.getClassMetadata(declaringClass);

        if (metaData == null) {
                // This is not a persistent class
                return FALSE;
        } else {
                return matchesProperty(m, declaringClass, metaData);
        }

    }

    abstract SelectionValue matchesProperty
        (CtMethod m, Class declaringClass, ClassMetadata cmd);

}
```

tas/horas no fuso horário de referência e convertê-los para o fuso do usuário em cada consulta ou operação. Para garantir que essas conversões sejam feitas de forma consistente e com o mínimo esforço, utilizamos um aspecto de conversão.

Como esse aspecto de conversão deve operar na transição entre o código de negócio da aplicação e o mecanismo de persistência, utilizamos os seletores desenvolvidos para selecionar esses pontos de forma precisa.

A listagem 5.11 contém o aspecto que efetua a conversão de fusos horários e a listagem 5.12 mostra os *pointcuts* que aplicam esse aspecto aos pontos de interesse.

Listagem 5.9 Classe base para os seletores que lidam com atributos (*properties*)

```
public abstract class HibernatePropertySelector extends HibernateSelector {

    SelectionValue matchesProperty(CtMethod m, Class declaringClass,
            ClassMetadata cmd) throws IntrospectionException {

        String[] persistentProperties = getProperties(cmd);

        for (String prop : persistentProperties) {

            // Gets the JavaBeans method used to set the property
            PropertyDescriptor pd = new PropertyDescriptor(prop, declaringClass);
            Method readMethod = getMethodToCompare(pd);

            // If the methods are the same, we found a match
            if (readMethod.getName().equals(m.getName())) {
                return TRUE;
            }
        }

        return FALSE;
    }

    abstract String[] getProperties(ClassMetadata cmd);

    abstract Method getMethodToCompare(PropertyDescriptor pd);

}
```

Listagem 5.10 Seletor de métodos de obtenção de atributos

```
@SelectorDef(name = "getter")
public class HibernateGetterSelector extends HibernatePropertySelector {

    String[] getProperties(ClassMetadata cmd) {
        return cmd.getPropertyNames();
    }

    Method getMethodToCompare(PropertyDescriptor pd) {
        return pd.getReadMethod();
    }
}
```

Listagem 5.11 Aspecto de conversão de fusos horários

```
public class TimeZoneConversionAspect {

    // Constante que define o fuso horário de referência
    private static final TimeZone REFERENCE_TIMEZONE = TimeZone.getTimeZone("GMT");

    // Adendo que trata conversões de fuso horário nas execuções de métodos de
    // atribuição de valor
    public Object aroundSetTimestamp(MethodInvocation invocation) throws Throwable {

        Object[] args = invocation.getArguments();
        Timestamp t = (Timestamp) args[0];

        User user = Session.getUser();
        TimeZone tz = user.getTimeZone();
        Timestamp t2 = convertTimeZone(t, tz, REFERENCE_TIMEZONE);

        args[0] = t2;
        invocation.setArguments(args);
        return invocation.invokeNext();
    }

    // Adendo que trata conversões de fuso horário nas execuções de métodos de
    // obtenção de valor
    public Object aroundGetTimestamp(MethodInvocation invocation) throws Throwable {

        User user = Session.getUser();
        TimeZone tz = user.getTimeZone();
        Timestamp t = (Timestamp)invocation.invokeNext();
        Timestamp t2 = convertTimeZone(t, REFERENCE_TIMEZONE, tz);
        return t2;
    }

    // Método utilitário para conversão de timestamps entre fusos
    private Timestamp convertTimeZone(Timestamp t, TimeZone fromZone, TimeZone toZone) {

        Calendar result = Calendar.getInstance();
        result.setTime(t);
        int fromZoneOffset = fromZone.getOffset(t.getTime());
        int toZoneOffset = toZone.getOffset(t.getTime());
        int gap = toZoneOffset - fromZoneOffset;
        result.add(Calendar.MILLISECOND, gap);
        return new Timestamp(result.getTime().getTime());
    }

}
```

Listagem 5.12 *Pointcuts* do aspecto de conversão de fusos horários

```
// Pointcut para atribuição de valores persistentes do tipo Timestamp
execution(* domain.*->*(java.sql.Timestamp)) AND setter()

// Pointcut para obtenção de valores persistentes do tipo Timestamp
execution(java.sql.Timestamp domain.*->*(..)) AND getter()
```

Capítulo 6

Trabalhos relacionados

Os problemas mostrados na seção 3.1 motivaram diversos trabalhos anteriores a este. Neste capítulo, os mais relevantes são apresentados.

Todas as abordagens existentes no sentido de linguagens de *pointcuts* mais flexíveis têm em comum o fato de transformarem as definições de *pointcuts* em meta-programas que atuam sobre um modelo do programa no qual o aspecto deve ser aplicado.

Uma possível classificação dessas propostas é pelo paradigma da linguagem usada como base. Esses paradigmas correspondem aos principais existentes na programação de forma geral. Por isso, agrupamos os trabalhos desta forma.

6.1 Programação funcional

6.1.1 XQuery/BAT

A proposta de Eichberg et. al. [13] é expressar definições de *pointcuts* como consultas na linguagem XQuery [9]. A XQuery é uma linguagem funcional, usada para criar consultas em um meta-modelo da linguagem de marcação XML.

Para permitir o uso do XQuery na programação de *pointcuts*, uma representação em XML dos *bytecodes* de um programa em Java é gerada pelo arcabouço BAT, criado pela equipe dos próprios autores. Esta transformação gera uma imagem do programa que é estática e de baixo nível, mas adequada como base para a execução de consultas XQuery.

Segundo os autores, a programação funcional permite que as declarações de designadores de

pointcuts sejam curtas, precisas e declarativas. Em contraste, implementações de *pointcuts* em linguagens como AspectJ são complicadas e, por isso, é necessário existir uma especificação formal da sua semântica para que se compreenda seu funcionamento.

Outra vantagem desse tipo de declaração seria a facilidade de composição de *pointcuts*. O XQuery permite que o resultado da aplicação de uma função seja passado para outra função, criando assim uma composição de filtros que torna a linguagem de *pointcuts* mais expressiva.

A listagem 6.1 mostra uma função declarada em XQuery que pode atuar como um novo *pointcut* primitivo, neste caso o `pcflow` (fluxo de controle previsto). Podemos ver que a declaração é bastante compacta, porém não muito legível para quem não conhece a linguagem.

Listagem 6.1 Um *pointcut* primitivo em XQuery

```
declare function pcflow($all as element()*, $m as element()*) as element()* {
        let $pcfl1 := $all//method[@name = $m//invoke/@method] except $m//method
        return if (empty($pcfl1)) then $m else pcflow($all, $m union $pcfl1)
}
```

Uma limitação desta abordagem é sua atuação apenas sobre a estrutura estática do programa, sem acesso a informações de tempo de execução. Isso a torna mais limitada do que os seletores de pontos de junção, que podem também levar em conta informação de tempo de execução.

Pode-se argumentar também que o uso da linguagem XQuery é uma desvantagem, pois ela não é adequada para programação de propósito geral. Isso implica que o programador terá que utilizar uma linguagem diferente daquela a que está habituado.

6.2 Programação lógica

Diversos trabalhos foram feitos no sentido de utilizar programação lógica como forma de especificação de *pointcuts*. A sugestão parece surgir naturalmente da semelhança sintática e semântica entre o uso de *pointcuts* e de fatos em uma base Prolog.

6.2.1 Andrew

Gybels [16] propõe uma linguagem de definição de *pointcuts* baseada na linguagem de programação lógica QSOUL. Esta linguagem é uma variante simplificada de Prolog, desenhada para permitir meta-programação sobre programas em Smalltalk. A linguagem orientada a objetos Smalltalk, por sua vez, serve de base para a implementação da parte funcional da linguagem orientada a

aspectos proposta pelos autores, denominada Andrew.

O modelo de pontos de junção de Andrew é composto de predicados que refletem as poucas construções primitivas de Smalltalk, como envio de mensagens e atribuições de valores a variáveis. A partir desses pontos de junção primitivos, os autores constroem *pointcuts* mais complexos aproveitando-se de convenções que são utilizadas por programadores daquela linguagem, já que ela não provê construções específicas para algumas funcionalidades como tratamento de exceções e inicialização de instâncias. Esta possibilidade de construir *pointcuts* para eventos que não correspondem a construções primitivas da linguagem é citada pelos autores como uma de suas motivações.

Gybels e Brichau elencam as características da sua linguagem (derivadas principalmente do paradigma lógico) que permitem a definição de *pointcuts* mais expressivos [17]. Estas características são:

a. Unificação de predicados: permite que sejam feitos casamentos ("matching") entre as propriedades dos pontos de junção, além de produzir resultados que podem ser usados em outras comparações;

b. Processamento sobre propriedades: provê a capacidade de obter valores derivados de propriedades, por exemplo, através de cálculos algébricos;

c. Ligação com sombras de pontos de junção: A linguagem Andrew possui um modelo dinâmico (pontos de junção) e um estático (sombras), permitindo o uso de ambos na composição de *pointcuts*.

d. Regras parametrizadas reutilizáveis: as regras das linguagens lógicas possuem variáveis que podem ser usadas como restrições ou como resultados, facilitando a reutilização das regras de forma genérica;

e. Recursão: a recursão torna esta linguagem computacionalmente completa, permitindo capturar padrões recursivos.

As características a, b, d e e estão ligadas diretamente ao paradigma lógico escolhido, enquanto a característica c faz parte do modelo de pontos de junção desenvolvido pelos autores.

6.2.2 Gamma

Um outro trabalho, mais recente, que utiliza uma linguagem lógica para definição de *pointcuts* é apresentado por Klose e Ostermann [27]. Os autores propõem uma linguagem denominada Gamma, uma combinação da linguagem OO "de brinquedo" L2 [11] (um pequeno subconjunto de Java) para a implementação da parte imperativa com Prolog para a definição de *pointcuts*.

A principal característica da linguagem proposta é o modelo de dados sobre o qual são definidos os *pointcuts*. Esse modelo baseia-se em um rastreamento (*trace*) da execução do programa (portanto um modelo dinâmico) em que cada evento está associado a um número seqüencial, chamado de *timestamp*. Todos os predicados utilizados têm como primeiro argumento um *timestamp*. Isto permite que se façam relações temporais entre pontos de junção para compor os *pointcuts*, envolvendo inclusive eventos futuros de uma determinada linha de execução.

Esta flexibilidade permite definir *pointcuts* complexos, como o `cflow` do AspectJ, de forma sucinta, como vemos na listagem 6.2.

Listagem 6.2 O *pointcut* cflow definido em Gamma

```
1 % T2 está no fluxo de controle da chamada em T1
2 cflow(T1, T2) :-
3   isbefore(T1,T2),
4   calls(T1,_,_,_,_),
5   endcall(T3,T1,_),
6   isbefore(T2,T3).
```

Essa definição pode ser lida como: dados dois instantes T1 e T2, o evento ocorrido em T2 está no fluxo de controle do evento em T1 (linha 2) se T1 é anterior a T2 (linha 3), o evento em T1 é uma chamada de método (linha 4), e a finalização da chamada iniciada em T1 ocorre em um momento T3 (linha 5), posterior a T2 (linha 6).

O exemplo da listagem 6.3 mostra o uso de eventos no futuro para a definição de um *pointcut*. A intenção é que, antes que sejam chamados métodos protegidos, seja forçada uma autenticação do usuário. Esta situação é semelhante à apresentada na seção 3.1.2.

Os autores chamam a atenção para a possibilidade de criar aspectos paradoxais. Como um *pointcut* pode depender de eventos posteriores à aplicação do adendo associado a ele, o adendo pode alterar as condições que o dispararam, fazendo com que ele não devesse ter sido disparado.

Listagem 6.3 Um *pointcut* com eventos no futuro aplicado a um adendo

```
1  before calls(Now,server,_,execute,_),
2       cflow(Now,T),
3       calls(T,database,_,protected,_) {
4
5           this.db.authenticate(true);
6
7  }
```

6.2.3 Alpha

A linguagem Alpha, proposta por Ostermann et. al. [30], busca tornar mais prática a linguagem Gamma, restringindo condições e detalhando a sua implementação. Essa linguagem é similar a Gamma, exceto por não permitir referências a eventos no futuro, além de dispor de mais modelos de dados sobre os quais podem ser compostos *pointcuts*.

Os autores propõem o uso de quatro diferentes e complementares fontes de informação: uma representação da árvore sintática abstrata, uma representação do armazenamento de objetos (*heap*), uma representação do tipo estático de cada expressão do programa e uma representação do rastreamento da execução do programa.

O trabalho apresenta uma comparação entre diversas formas de definir o mesmo *pointcut*, para um exemplo de editor de figuras semelhante ao visto na seção 3.1.1. A comparação é feita em termos da capacidade das definições de resistir a mudanças no programa-base. Em ordem crescente de qualidade, os *pointcuts* definidos seguem as seguintes abordagens:

a. Enumeração de pontos de junção

b. Padrão de nomenclatura

c. Previsão de fluxo de controle

d. Fluxo de controle em tempo de execução

e. Acessibilidade de objetos no fluxo de execução

As duas primeiras definições seguem o estilo de definição por enumeração. No primeiro caso, são enumerados todos os pontos de junção. No segundo, usam-se máscaras para selecionar um conjunto.

As três últimas buscam definições semânticas do *pointcut*, sempre partindo da idéia de capturar atribuições a campos que sejam lidos durante a execução do método `redraw`, que faz a atualização da exibição das figuras.

No caso do *pointcut* c, é utilizada uma estimativa do fluxo de controle da chamada ao método `redraw` com base na estrutura estática do programa. O *pointcut* d utiliza a informação de tempo de execução dessa chamada, melhorando a precisão do *pointcut*. Por fim, o *pointcut* e busca as atribuições feitas dentro do fluxo de controle da chamada a `redraw`, a campos de objetos aos quais o `FigureElement` em questão tenha acesso por meio de referências.

Os autores argumentam que esta última forma de definição é a mais resistente a mudanças, pois seria capaz de resistir às seguintes classes de alterações: refatoração que coloca parte do estado de um derivado de `FigureElement` fora da própria classe; adição de um novo derivado de `FigureElement` à hierarquia; mudança das condições em que um redesenho de tela é necessário; adição ou remoção de um campo cuja alteração torna um redesenho necessário; e adição ou remoção de um campo à classe que não afeta a operação de redesenho.

Na seção de trabalho futuro, os autores afirmam que pretendem adaptar sua tecnologia a uma linguagem de programação completa e superar os obstáculos de desempenho para tornar a sua abordagem utilizável na prática.

6.3 Programação imperativa

6.3.1 Josh

Em [7], é introduzida uma nova linguagem de programação chama Josh. Esta linguagem possui os mesmos recursos do AspectJ, exceto pela habilidade de definir novos *pointcuts* primitivos. A definição desses *pointcuts* é feita na mesma linguagem que serve de base para a parte orientada a objetos e para a implementação dos adendos, Java.

A base para a implementação e também para a filosofia de Josh é um arcabouço de metaprogramação e manipulação de *bytecodes* Java chamado Javassist. O Javassist serve como API para a análise da estrutura do programa por parte das implementações de *pointcuts* e também como ferramenta para introdução dos adendos em meio ao programa orientado a objetos.

Graças a esta última característica, Josh oferece um recurso curioso, não encontrado em outras linguagens orientadas a aspectos: construir dinamicamente (no momento da combinação dos aspec-

tos) trechos de código Java, utilizando valores obtidos por meta-programação. Esta característica, entretanto, não está relacionada à definição de *pointcuts* e, portanto, não é objeto desta dissertação.

A listagem 6.4 (tirada diretamente de [7]) exemplifica a definição de um novo *pointcut* primitivo em Josh, criado para resolver o problema da seção 3.1.1 (o editor de figuras). Uma expressão *pointcut* com o uso deste *pointcut* primitivo ficaria semelhante ao pseudo-código da listagem 6.5.

Listagem 6.4 Um novo *pointcut* definido em Josh

```
static boolean updater(MethodCall mc,
    String[] args, JoshContext jc) {

  CtClass root = jc.getCtClass(args[0]);
  String mname = args[1];
  CtMethod mth = mc.getMethod();

  // skip if the method is redraw().
  if (root.getName().equals(mname))
    return false;

  Hashtable fields = enumerateFields(jc, root, mname);
  updated = false;
  mth.instrument (new ExprEditor() {
    public void edit(FieldAccess expr) {
      String name = expr.getFieldName();
      if (expr.isWriter() && fields.get(name) == expr.getCtClass())
        updated = true;
    }
  });
  return updated;
}
```

Listagem 6.5 Uso de um *pointcut* definido em Josh

```
aspect DisplayUpdating {
 pointcut move() :
  call(updater(void FigureElement+.*(..)));

 after() returning :
   move() && target(fe) {
  fe.redraw();
 }
}
```

Embora o uso do novo *pointcut* seja simples, a sua definição é muito mais complicada do que seria nas abordagens que utilizam linguagens lógicas ou funcionais. Além disso, Josh leva em conta apenas informação estática, disponível no momento da combinação dos aspectos. Isto significa que uma

atualização da exibição, no exemplo, poderia ser disparada sem que uma variável relevante tivesse sido alterada (por exemplo, caso haja um operador condicional no fluxo).

Do ponto de vista prático, esta abordagem parece ser a mais próxima do uso real. A implementação do protótipo teve desempenho muito próximo ao de linguagens mais maduras, como o AspectJ, e causou pequeno sobrepeso em comparação com o programa sem a aplicação de aspectos.

6.4 Avaliação comparativa

Nesta seção, apresentamos um resumo comparativo entre o nosso trabalho e as outras propostas analisadas.

A tabela 6.1 é uma tentativa de classificação crítica das linguagens e abordagens propostas. A seguir, detalharemos as informações sumarizadas por essa tabela.

Linguagem / Abordagem	**Paradigma**	***Pointcuts* dinâmicos?**	**Informação temporal?**	**Resistência a mudanças**	**Clareza de intenção**	**Viabilidade prática**
XQuery / BAT	Funcional	não	não	média	alta	média
Andrew	Lógico	sim	não	alta	alta	média
Gamma	Lógico	sim	sim	alta	alta	baixa
Alpha	Lógico	sim	sim	alta	alta	média
Josh	Imperativo	não	não	média	alta	alta
Seletores	Imperativo	sim	não	alta	alta	alta

Tabela 6.1: Comparação de novas abordagens em linguagens de *pointcuts*

6.4.1 *Pointcuts* dinâmicos

Por *pointcuts* dinâmicos, entende-se a capacidade de definir *pointcuts* com base em informações que estão disponíveis apenas em tempo de execução do programa.

As linguagens XQuery/BAT e Josh não provêm esta possibilidade, considerando-a como trabalho futuro. As outras propostas, inclusive a apresentada aqui, possuem esse recurso.

6.4.2 Informação temporal

Indica-se aqui se a linguagem permite que *pointcuts* sejam compostos com base em informações dinâmicas (portanto implicando um sim ao item 6.4.1) sobre eventos ocorridos no passado ou que ocorrerão na execução futura do programa. Apenas as linguagens Gamma e Alpha provêem este recurso. No caso de Alpha, apenas para eventos no passado, e para Gamma também no futuro.

Esse recurso, entretanto, é bastante experimental e os próprios autores admitem que ele precisa ser bastante aperfeiçoado do ponto de vista do desempenho para que possa ser usado na prática.

6.4.3 Resistência a mudanças

Neste item é feita uma avaliação da capacidade de resistência a mudanças dos *pointcuts* que podem ser escritos na linguagem. A base de comparação, considerada baixa (que todos pretendem superar), é a linguagem AspectJ e outras semelhantes que estão disponíveis atualmente (conforme exposto na seção 2.3).

As linguagens XQuery/BAT e Josh foram consideradas médias neste quesito. No caso de Josh e do XQuery/BAT, isto se deve à ausência de informação de tempo de execução, que limita a expressividade.

Nossa abordagem foi considerada de alta resistência a mudanças, por ter as capacidades das anteriores mais a possibilidade de definição com base em informações de tempo de execução. As linguagens Andrew, Gamma e Alpha também foram consideradas capazes de gerar *pointcuts* com alta resistência a mudanças, tendo em vista as características dessas linguagens (vide seções 6.2.2 e 6.2.3).

6.4.4 Clareza de intenção

A clareza de intenção procura avaliar o quanto o *pointcut* captura e transmite a real intenção do programador que o criou.

Todas as abordagens avaliadas, além da nossa proposta, foram consideradas de alta clareza. Isso se deve à possibilidade de definição de *pointcuts* (e, no nosso caso, seletores) semânticos, que transmitem mais claramente a intenção do programador.

6.4.5 Viabilidade prática

Tanto a nossa abordagem quanto a linguagem Josh são viáveis na prática, sem que sejam necessárias grandes rupturas de paradigmas na programação orientada a aspectos atual. Em nosso caso, implementamos a solução como uma extensão plenamente funcional a um arcabouço já consolidado na prática.

A linguagem Gamma é considerada de baixa viabilidade, devido aos desafios apresentados por

sua arquitetura que permite analisar eventos futuros. Isto foi apontado pelos próprios autores, que procuraram simplificar sua proposta na linguagem Alpha.

Capítulo 7

Considerações finais

7.1 Principais contribuições

A programação orientada a aspectos é uma idéia relativamente jovem, que apenas recentemente tem sido adotada na prática por um conjunto mais amplo de desenvolvedores. A história mostra que novos paradigmas de programação costumam demorar algum tempo [31] para amadurecer e serem aceitos pela maioria. Durante esse processo, é natural que não existam consensos e muitas idéias novas sejam testadas.

Neste trabalho, propusemos um mecanismo de extensão para as linguagens orientadas a aspectos. Este mecanismo pode melhorar a utilidade dessas linguagens de duas formas: permitindo que seu uso seja mais simples e eficaz e aumentando sua gama de aplicações. Acreditamos que esse pode ser mais um passo em direção à maturidade e adoção da POA.

As linguagens de *pointcuts* atuais são pobres em mecanismos que permitam a construção de novas abstrações. A maioria delas consiste de um conjunto fixo de primitivas que podem ser combinadas apenas por operadores booleanos. Com a abertura ao programador da possibilidade de criar novas primitivas com o poder de expressão de uma linguagem de propósito geral, níveis mais altos de abstração podem ser conseguidos sem sacrificar a simplicidade na criação de *pointcuts*.

Este trabalho mostra que mecanismos de extensão para linguagens de *pointcuts* podem melhorar a qualidade dos *pointcuts*. Em particular, o mecanismo aqui proposto permite a criação de novos tipos de *pointcuts* que antes não eram possíveis, como os que utilizam meta informação localizada fora do código-fonte. Isso é importante devido à forte tendência de utilização de arcabouços que empregam meta-informação.

O trabalho apresentado nesta dissertação gerou a seguinte publicação:

- *Join Point Selectors* [5]: artigo publicado no *5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '07)* [4], realizado em conjunto com a conferência *Aspect-Oriented Software Development 2007 (AOSD '07)*. Este artigo apresenta o conceito de seletores de pontos de junção e o protótipo implementado.

7.2 Trabalhos futuros

7.2.1 Otimização de expressões *pointcut*

Na versão atual do protótipo de seletores de pontos de junção, há espaço para otimização no processo de avaliação de expressões *pointcut* em tempo de execução. Conforme descrito na seção 4.2.2, nos casos em que existe necessidade de avaliação de um seletor em tempo de execução, todo o *pointcut* associado a ele precisa ser avaliado. Entretanto, podem haver cláusulas dessa expressão cujo valor já foi decidido em tempo de combinação, e portanto a expressão poderia ser simplificada com a eliminação dessas cláusulas.

Por exemplo, no *pointcut* "`within(com.acme.*) AND getter()`", usado como exemplo na seção 4.3, a cláusula `within` poderia ser removida em tempo de combinação, mesmo que a cláusula `getter` exigisse teste em tempo de execução.

Para implementar essa otimização, seria necessário encontrar uma forma satisfatória de armazenar as expressões otimizadas de forma que elas possam ser carregadas em tempo de execução, seja nas próprias classes compiladas e combinadas ou em um arquivo externo. Uma solução intermediária seria efetuar a otimização não em tempo de combinação, mas em tempo de carregamento do programa para execução.

7.2.2 Mudança do modelo de combinação

A combinação de seletores de pontos de junção em expressões *pointcut* poderia tornar-se mais flexível se adotássemos as seguintes mudanças na sua semântica:

- Em vez de receber um único ponto de junção como argumento, um seletor receberia uma referência para um conjunto deles;

- Em vez de devolver um valor indicativo, o seletor devolveria um subconjunto do conjunto de seletores recebido como argumento.

Estas mudanças permitiriam o uso do resultado de um seletor como argumento para outro seletor. Por exemplo, o seletor `updatesStateReadBy` da seção 5.2 poderia ser dividido em dois seletores: um deles selecionaria os campos lidos por um determinado método e o outro selecionaria os métodos que atualizam algum dos campos de um conjunto fornecido. Para formar o equivalente ao `updatesStateReadBy`, portanto, o resultado do primeiro seletor seria fornecido como argumento ao segundo. Dessa forma, ambas as partes poderiam ser reutilizadas independentemente.

A implementação dessas mudanças, entretanto, exigiria uma mudança completa na arquitetura do combinador do JBoss AOP. Faz-se necessário um maior esforço para determinar a viabilidade dessa reformulação, tanto do ponto de vista funcional como do desempenho.

7.2.3 Agregar informação ao *pointcut* para uso em adendos

A linguagem Doxpects, descrita brevemente na seção 3.1.3, oferece um recurso que não pode ser obtido através dos seletores em sua forma atual. Ela permite que o programador do adendo tenha acesso a objetos criados pelo *pointcut* com base em informação do ponto de junção, em tempo de execução. Naquele caso específico, os objetos são instâncias de classes especificadas pelo programador, com informações obtidas dos elementos do XML selecionado pelo *pointcut*.

Essa funcionalidade, se acrescentada de forma genérica ao mecanismo dos seletores de pontos de junção, poderia enriquecer a programação de aspectos em diversos casos, especialmente aqueles que envolvem meta-informação. O principal benefício dessa funcionalidade seria permitir que o adendo tivesse acesso a informação em um nível de abstração mais alto do que a API de reflexão disponibilizada pelo JBoss AOP.

7.3 Conclusão

Nesta dissertação, exploramos a necessidade e a viabilidade de estender as linguagens de *pointcuts* com a criação de novos elementos primitivos pelo programador. Foram expostas as limitações das linguagens de *pointcuts* atuais, mostrando que muitas vezes elas não permitem criar pointcuts de boa qualidade, o que motiva nossa tentativa de buscar novas soluções.

Para melhor caracterizar a abordagem utilizada, julgamos necessário definir o conceito de seletores

de pontos de junção. Além de dar suporte conceitual à solução apresentada, consideramos que esta definição tem o valor intrínseco de separar os algoritmos de seleção de pontos de junção de seus usos em *pointcuts* específicos. Essa separação pode tornar mais claras as discussões sobre mecanismos de extensão de linguagens de *pointcuts*.

A viabilidade da proposta foi demonstrada pela implementação de um protótipo funcional. Embora seja ainda passível de melhorias, o protótipo mostrou-se capaz de ser usado em situações práticas, como as mostradas nos exemplos. Em nosso protótipo, tanto as informações disponíveis em tempo de combinação como as disponíveis apenas em tempo de execução podem ser utilizadas de forma simples por seletores de pontos de junção. Essa característica do arcabouço faz com que a implementação de seletores seja um trabalho pouco complexo.

Os diversos exemplos apresentados demonstram a aplicabilidade prática do conceito de seletor de pontos de junção. Certamente existem muitas outras aplicações, além das abordadas no Capítulo 5. Nossos exemplos demonstraram também que é possível melhorar a qualidade dos *pointcuts* com o uso de seletores. Os *pointcuts* aperfeiçoados têm melhor legibilidade e resistência a mudanças do que os que seriam possíveis nas linguagens de *pointcut* tradicionais.

Finalmente, consideramos que o mecanismo de seletores de pontos de junção ainda pode ser bastante estendido para lidar com mais casos de uso, com melhor desempenho e qualidade de *pointcuts*. A seção de trabalhos futuros forneceu alguns exemplos de como isso pode ser feito.

Referências Bibliográficas

[1] AspectJ Project. *AspectJ website*. http://eclipse.org/aspectj.

[2] AspectJ Team, Xerox Corporation. *AspectJ Programming Guide*, 2005. http://eclipse.org/aspectj/doc/released/progguide.

[3] Christian Bauer and Gavin King. *Hibernate in Action*. Manning, 2005.

[4] Lodewijk Bergmans, Johan Brichau, Erik Ernst, and Kris Gybels, editors. *SPLAT '07: Proceedings of the 5th Workshop on Engineering Properties of Languages and Aspect Technologies*, New York, NY, USA, 2007. ACM.

[5] Cristiano Breuel and Francisco Reverbel. Join point selectors. In *SPLAT '07: Proceedings of the 5th Workshop on Engineering Properties of Languages and Aspect Technologies*, pages 14–21, New York, NY, USA, 2007. ACM.

[6] Johan Brichau and Michael Haupt. Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD-Europe, May 2005.

[7] Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 102–111. ACM Press, March 2004.

[8] World Wide Web Consortium. XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath.

[9] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation 23 January 2007. http://www.w3.org/TR/xquery.

[10] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[11] Sophia Drossopoulou. Lecture notes on the L2 calculus. http://www.doc.ic.ac.uk/~scd/Teaching/L1L2.pdf.

[12] R. Johnson E. Gamma, Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2004.

[14] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*.

[15] Alessandro Garcia, Christina Chavez, Sergio Soares, Eduardo Piveta, Rosângela Penteado, Valter Vieira de Camargo, and Fabrício Fernandes. 1o. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos – WASP 2004 – Relatório do Workshop. 2004.

[16] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In Pascal Costanza, Günter Kniesel, Katharina Mehner, Elke Pulvermüller, and Andreas Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, February 2002. Technical report IAI-TR-2002-1.

[17] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Mehmet Akşit, editor, *Proc. 2nd Intl' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 60–69. ACM Press, March 2003.

[18] JavaCC Project. *JavaCC website*. https://javacc.dev.java.net.

[19] JBoss Inc. *JBoss AOP Reference Documentation (v1.3)*. http://docs.jboss.com/aop/1.3/aspect-framework/.

[20] JJTree project. *JJTree website*. https://javacc.dev.java.net/doc/JJTree.html.

[21] Mik Kersten. Aop@work: AOP tools comparison, part 1: Language mechanisms. Technical report, IBM Developer Works, February 2005.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Comm. ACM*, 44(10):59–65, October 2001.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[24] Gregor Kiczales. The fun has just begun. keynote. In *AOSD 2003, Boston*, March 2003.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[26] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, 2005. ACM Press.

[27] Karl Klose and Klaus Ostermann. Back to the future: Pointcuts as predicates over traces. In Gary T. Leavens, Curtis Clifton, and Ralf Lämmel, editors, *Foundations of Aspect-Oriented Languages*, March 2005.

[28] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[29] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Ron Cytron and Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 17–26, March 2002.

[30] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

[31] Tim Rentsch. Object oriented programming. *SIGPLAN Not.*, 17(9):51–57, 1982.

[32] Dominik Stein, Stefan Hanenberg, and Rainer Unland. An UML-based aspect-oriented design notation. In Gregor Kiczales, editor, *Proc. 1st Intl' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 106–112. ACM Press, April 2002.

[33] Eric Wohlstadter and Kris De Volder. Doxpects: aspects supporting XML transformation interfaces. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-oriented Software Development*, pages 99–108, New York, NY, USA, 2006. ACM Press.