Combinação Dinâmica de Aspectos: Uma Abordagem Eficiente

Flavia Rainone

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Francisco Carlos da Rocha Reverbel**

São Paulo, agosto de 2008

Resumo

Os sistemas dinâmicos orientados a aspectos permitem a adição e a remoção de aspectos em tempo de execução, capacidade denominada *combinação dinâmica*. Tais sistemas são ditos adaptáveis, pois possuem a capacidade de se adaptar a novas situações no decorrer de sua execução.

Este trabalho visa avaliar o custo de desempenho envolvido na execução da combinação dinâmica, fator que pode afetar o desempenho do sistema durante a execução de operações de adição e remoção dinâmicas de aspectos. Mostraremos que esse custo é relativamente alto e iremos propor uma nova abordagem de implementação a fim de minimizá-lo.

A nossa abordagem é formada por duas partes. A primeira envolve a criação de uma estrutura de dados capaz de devolver informações sobre pontos de junção utilizando um *pointcut* como chave de busca. Essa estrutura de dados é chamada de grafo de sombras de junção e possui um desempenho médio superior, quando comparada com o processo de casamento de *pointcuts* tradicional.

A segunda parte da abordagem que propomos consiste em alterar os *bytecodes* da aplicação em tempo de execução, processo conhecido como *hot swap*, com o objetivo de garantir que o fluxo de controle de um ponto de junção se manterá inalterado quando não houver adendos a serem chamados durante sua execução.

A fim de realizarmos uma avaliação prática da abordagem proposta, implementamos a nossa solução na ferramenta de programação orientada a aspectos dinâmica JBoss AOP. Note, no entanto, que poderíamos ter implementado a abordagem proposta em qualquer ferramenta de programação orientada a aspectos dinâmica em Java, ou ainda, em outra linguagem que fornecesse suporte a *hot swap*.

Abstract

Dynamic aspect-oriented systems support the addition and removal of aspects at runtime, capability called *dynamic weaving*. Such systems are called adaptable, given they possess the ability to adapt to new situations during their execution.

This work targets the evaluation of the performance costs involved in the execution of dynamic weaving, which can impact the overall performance of the system during dynamic aspect-oriented operations. We will show that this cost is relatively high and we will propose a new approach to minimize it.

Our approach is composed of two parts. The first one involves the creation of a data structure capable of retrieving information regarding joinpoints using a pointcut as search key. This data structure is called the joinpoint shadow graph and provides an average better performance when compared with the traditional pointcut matching process.

The second part of our approach consists in changing the bytecodes of the application at runtime, a technique called hot swap, in order to enforce that we will always keep a joinpoint control flow unchanged when there are no advice to be run during that joinpoint execution.

In order to perform an evaluation of the proposed approach, we have implemented our solution at the JBoss AOP aspect-oriented programming tool. Note, however, that the proposed approach could have been implemented in any other dynamic aspect-oriented programming tool or, yet, in other language that offers support to hot swap.

Sumário

1. Introdução

A combinação dinâmica de aspectos consiste em adicionar e remover aspectos de um sistema em tempo de execução. Esse recurso, disponível somente em algumas ferramentas de programação orientada a aspectos, permite a adaptabilidade de um sistema a novas situações durante a sua execução.

Apesar do potencial que a combinação dinâmica de aspectos representa, ela envolve um custo de desempenho relativamente alto, se comparado com o impacto causado por aspectos adicionados em tempo de compilação. O motivo desse impacto está na própria natureza da combinação dinâmica de aspectos, que requer um algoritmo diferente do utilizado na combinação de aspectos em tempo de compilação. Durante a compilação, o tempo gasto pela combinação de aspectos não acarreta custos à execução do sistema, visto que toda a operação é realizada antes que essa execução tenha início. O mesmo não se aplica à combinação dinâmica de aspectos, que ocorre em tempo de execução e, portanto, oferece um impacto direto ao desempenho do sistema.

O principal objetivo deste trabalho é o estudo dos mecanismos envolvidos na combinação dinâmica de aspectos e do seu desempenho. Assim, começaremos avaliando os custos envolvidos na implementação da combinação dinâmica de aspectos. Os nossos argumentos serão embasados em um estudo detalhado de sua implementação nas ferramentas JBoss AOP, AspectWerkz, JasCO e PROSE.

Através dessa avaliação, identificamos dois pontos da combinação dinâmica que classificamos como críticos para a obtenção de um desempenho melhor. Esses pontos são: a atualização de cadeias de adendos, onde é preciso recalcular os adendos que interceptarão cada ponto de junção; e a necessidade de manter o fluxo de controle do sistema inalterado nos pontos de junção que não serão interceptador por adendos. Em relação a esse último item, observe que, com a adição e remoção de aspectos em tempo de execução, o estado de um ponto de junção pode mudar. Queremos garantir que, sempre que não houver interceptadores, esse ponto de junção executará sem nenhuma operação adicional.

Para cada um dos pontos identificados nessa avaliação da combinação dinâmica, propomos uma solução de implementação que resultará em uma melhora visível no seu desempenho.

Assim, para a atualização de cadeias, propomos uma solução que envolve a implementação de uma complexa estrutura de dados, o grafo de sombras. Esse grafo irá conter informações sobre os pontos de junção, fornecendo suporte a buscas de pontos tendo *pointcuts* como chave de busca. Assim, quando um aspecto for adicionado em tempo de execução, faremos uma busca no grafo, que devolverá de forma eficiente as cadeias de adendos a serem atualizadas. Propomos, também, uma solução que implementará a remoção de aspectos sem necessidade de realizar buscas ou casamentos de *pointcuts*. Para isso, iremos manter informações sobre a adição de aspectos em memória, utilizando essa informação para remover os adendos das cadeias de forma eficiente.

Quanto ao segundo ponto crítico que levantamos, propomos uma solução que consistirá na alteração dos *bytecodes* de um a sistema em tempo de execução. Essa alteração atualizará o código que executa os pontos de junção, garantindo que o código original será executado quando não houver aspectos. Faremos alterações nesse código somente quando houver adendos a serem chamados.

As soluções que propomos foram implementadas no JBoss AOP, ferramenta de programação orientada a aspectos do JBoss. Utilizamos essa implementação para avaliar o desempenho de nossas soluções, comprovando a sua eficiência em relação a outras soluções implementadas previamente pelo JBoss.

A implementação desse trabalho encontra-se disponível no repositório do JBoss AOP, no endereço:

`http://anonsvn.jboss.org/repos/jbossas/projects/aop/branches/joinpoint_graph/`

Nesse endereço, é possível encontrar tanto a implementação de nosso trabalho como é possível também executar os testes de unidade.

1.1. Organização

Esse texto está organizado da seguinte forma. No Capítulo 2, apresentamos os principais conceitos envolvidos na programação orientada a aspectos, essenciais para o entendimento de nosso trabalho. A seguir, no Capítulo 3, fazemos uma análise dos custos envolvidos na implementação da combinação dinâmica, utilizando como base a implementação de uma série de ferramentas. Antes de seguirmos para uma descrição detalhada das soluções que propomos, o Capítulo 4 apresenta uma visão geral da arquitetura do JBoss AOP, onde descrevemos as principais classes que compõem a sua arquitetura e as funcionalidades que elas implementam. Após isso, o Capítulo 5 descreve a abordagem que propomos para a implementação da atualização de cadeias de adendos. O principal componente dessa solução, o grafo de sombras, é o assunto do Capítulo 6. A segunda parte do nosso trabalho, a instrumentação em tempo de execução, é descrita na íntegra no Capítulo 7. Após, descrevemos os testes de desempenho que realizamos, e analisamos os resultados obtidos no Capítulo 8. A lista de trabalhos relacionados está

disponível no Capítulo 9. Finalmente, concluímos o texto algumas considerações finais, no Capítulo 10.

2. Programação Orientada a Aspectos

Este capítulo é uma introdução à programação orientada a aspectos e nos permitirá definir a terminologia necessária para a compreensão dos capítulos seguintes. Ilustraremos também os principais conceitos envolvidos através de uma análise da ferramenta de programação orientada a aspectos JBoss AOP.

A programação orientada a aspectos, ou POA, é um paradigma criado por Gregor Kiczales et al. [50] que visa o encapsulamento das funcionalidades ortogonais a um sistema. É interessante notar que o surgimento da POA é uma conseqüência natural do curso histórico tomado pela evolução das linguagens de programação. O que impulsionou a sua criação foi a necessidade crescente de modularização e reutilização de código, com o intuito de viabilizar sistemas cada vez mais complexos, bem como de facilitar a manutenção dos mesmos. Dessa forma, consideramos que o decorrer dos fatos históricos na busca por essas características é essencial para apontar o principal intuito da POA. Os recursos providos por esse novo paradigma estão associados a um mesmo objetivo comum, como veremos adiante.

2.1. Motivação Histórica

"O primeiro computador programável, a Máquina Analítica de Babbage, projetada na década de 1840, tinha a capacidade de reutilizar coleções de cartões de instrução em diversos lugares diferentes em um programa quando isso era conveniente." [71]

A reutilização de instruções tem sido um fator importante desde os primórdios da ciência da computação. Quando os primeiros computadores passaram a ser comercializados, no final da década de 1940 e início da década de 1950, a capacidade limitada de processamento aliada à dificuldade de programação das máquinas por conta da falta de software de apoio tornava o processo de desenvolvimento de programas bastante custoso. Por conta disso, uma grande importância foi dada à reutilização de instruções à medida em que esta possibilitava uma redução considerável do custo associado ao desenvolvimento de software.

Entretanto, no final da década de 1960 e início da década de 1970, os computadores passaram a apresentar um custo cada vez mais baixo, assim como adquiriram maior poder de processamento e mais memória [71]. Em contrapartida, o desenvolvimento de software não acompanhou tal evolução; permanecia relativamente lento, ao mesmo tempo em que o custo dos programadores aumentava. Uma causa desse contraste foi o próprio desenvolvimento das arquiteturas de máquinas computacionais, que permitia aumentar a complexidade dos problemas solúveis por tais máquinas e, portanto, a dos programas a serem escritos. Esse cenário estimulou pesquisas em linguagens de programação, com o objetivo de criar novas linguagens que facilitassem a compreensão dos programas e, assim, agilizariam o

desenvolvimento dos mesmos. Em paralelo, novas metodologias de desenvolvimento e novos paradigmas de programação foram surgindo, com o mesmo intuito de reduzir o custo e agilizar o processo de desenvolvimento.

Desde então, desenvolvedores e pesquisadores têm se empenhado em diminuir os custos de desenvolvimento e de manutenção de sistemas. Uma das principais conseqüências de tais esforços é a potencialização das capacidades de abstração e de encapsulamento de código. E, nesse contexto, desde a existência do primeiro computador programável, é possível notar a importância do conceito de reutilização de código, permitindo que os esforços de programadores pudessem ser reaproveitados.

Até meados da década de 70, existia somente a abstração de procedimentos. A mesma consiste em reunir procedimentos programados em unidades, conhecidas como subprogramas, que podem ser invocados por procedimentos de outros programas e subprogramas. Isso permite a reutilização de código através dos subprogramas. A abstração de procedimentos sempre foi suportada pelas linguagens, inclusive pela Máquina de Babbage. Foi somente a partir do final da década de 70 que a abstração de dados ganhava importância significativa. Simultaneamente, o encapsulamento, conceito relacionado à abstração, passava a ser valorizado. O encapsulamento consiste na reunião de subprogramas e dados logicamente associados em unidades ou módulos e permite a estruturação de um sistema complexo, tornando-o intelectualmente administrável.

Anos mais tarde, com as linguagens Smalltalk e Simula, surgiu um novo paradigma de programação: a orientação a objetos (POO). Nesse paradigma, a abstração de dados é reafirmada através do encapsulamento de código e dados em construções denominadas *classes*, e é acrescida de um poderoso recurso, a herança de classes, que permite a reutilização de código através da extensão do mesmo. Porém, esses recursos não resolveram todos os problemas. A manutenção ainda era custosa e era difícil atingir o encapsulamento total de funcionalidades em classes reutilizáveis. Surgiram então os padrões de projeto [4, 15, 25, 27, 70]. Organizados em catálogos, esses padrões são soluções orientadas a objetos para problemas comuns. Os padrões têm o seu uso comprovado através da aplicação dos mesmos em diversos sistemas reais e auxiliam na criação de um projeto orientado a objetos flexível e reutilizável.

Apesar de eficazes na disseminação de boas práticas de desenho e encapsulamento, os padrões de projeto não resolvem todos os problemas de encapsulamento do código em classes. Essa questão nos remete ao problema da **separação de interesses** ou separação de responsabilidades [55]. Em um sistema, existem responsabilidades cuja implementação, até então, ficava inerentemente entrelaçada ao código que implementa a funcionalidade principal do mesmo. Alguns exemplos dessas responsabilidades incluem sincronização, transações, persistência de dados, tratamento de erros, controle de acesso e *logging*. O paradigmas de programação estruturada não permitiam a separação dessas responsabilidades em relação às demais. A consequência foi o surgimento de novas abordagens que permitissem o encapsulamento dessas responsabilidades em unidades distintas, bem como a composição das mesmas para a execução correta de um sistema. O objetivo de tais abordagens é obter um aumento da capacidade de reutilização do código, bem como da sua inteligibilidade [55]. Mais tarde, essas responsabilidades secundárias vieram a ser denominadas **funcionalidades ortogonais**.

Na década de 90, o problema da separação de responsabilidades ortogonais em linguagens POO começava a ser notado por desenvolvedores e pesquisadores da área. Tinha então início o surgimento

de novas abordagens para resolver essa questão. Dentre elas, podemos destacar os filtros de composição [2,3], a decomposição multi-dimensional de um sistema [74] e a programação orientada a aspectos [50]. Esta última foi a que mais se destacou na comunidade acadêmica e, desde 2002, há uma conferência anual dedicada ao tema, a AOSD (Aspect-Oriented Software Developement). Na indústria, a programação orientada a aspectos se manifesta principalmente através de ferramentas de *software* livre, como o AspectJ e o JBoss AOP.

2.2. Termos e Definições

A Programação Orientada a Aspectos (POA) é um paradigma de programação cujos princípios e conceitos nos remetem à questão do encapsulamento e do seu efeito na reutilização do código, bem como à questão da separação de interesses. A POA consiste em encapsular as funcionalidades ortogonais a um sistema em unidades distintas, os **aspectos**. Essas unidades ficam totalmente à parte daquelas que contêm o restante do código de um sistema, parcela denominada **sistema base** [1].

Para ilustrarmos melhor o conceito de aspectos, observe a classe abaixo:

```
1  public class WithdrawOperation
2  {
3     private double amount;
4     public synchronized void execute(double withdrawAmount)
5                                          throws OperationException
6     {
7        // start a transaction
8        transaction.begin();
9        // withdraw the amount
10       if (Double.compare(this.amount, drawAmount) < 0)
11       {
12          // rollback the transaction
13          transaction.rollback();
14          throw new OperationException(
15               "There is no such amount available for withdrawal");
16       }
17       this.amount -= drawAmount;
18       //  commit the transaction
19       transaction.commit();
20    }
```

[1] Note que as unidades de encapsulamento utilizadas na implementação de um sistema base variam de acordo com a linguagem na qual ele foi implementado, denominada **linguagem base**. Normalmente, essa linguagem é orientada a objetos e a unidade de encapsulamento utilizada para a sua implementação é a classe.

```
21 }
```

Listagem 2.1: Uma operação transacional

As linhas 8, 13 e 19 contêm operações de transação, que nada têm a ver com a funcionalidade que `WithdrawOperation` se propõe a implementar, a saber, saque de valores financeiros. Num sistema com operações transacionais, esse código tipicamente se repete ao longo de diversos métodos e classes, como mostra a Figura 2.1.

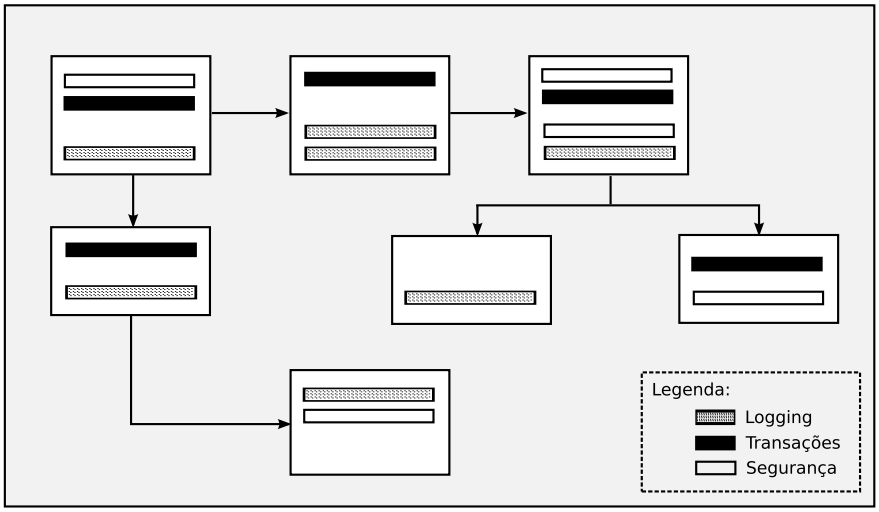


Figura 2.1.: Diagrama de um sistema orientado a objetos. Cada unidade representa uma classe, e as funcionalidades ortogonais, espalhadas ao longo dessas classes, estão em destaque conforme legenda.

Operações transacionais são um exemplo de funcionalidade ortogonal, assim denominada por "atravessar"diversas classes. Ao se implementar o mesmo sistema utilizando POA, temos como resultado a Figura 2.2 na página seguinte. Nela, vemos as funcionalidades ortogonais segurança, transação e *logging* encapsuladas em aspectos. O restante das funcionalidades permanecem implementadas em classes utilizando-se a mesma estrutura que antes. Tais classes compõem o denominado sistema base.

Um aspecto é composto, principalmente, por **adendos**(*advice* em inglês), que são blocos de código responsáveis por executar uma funcionalidade ortogonal. O suporte a transações pode ser implementado em um único adendo, como mostra a Listagem 2.2[2].

```
1 public class TransactionAspect
2 {
3     ...
```

[2]Utilizaremos a sintaxe do JBoss AOP para os exemplos desse texto. Detalhes dessa sintaxe serão vistos na a Seção 2.4.

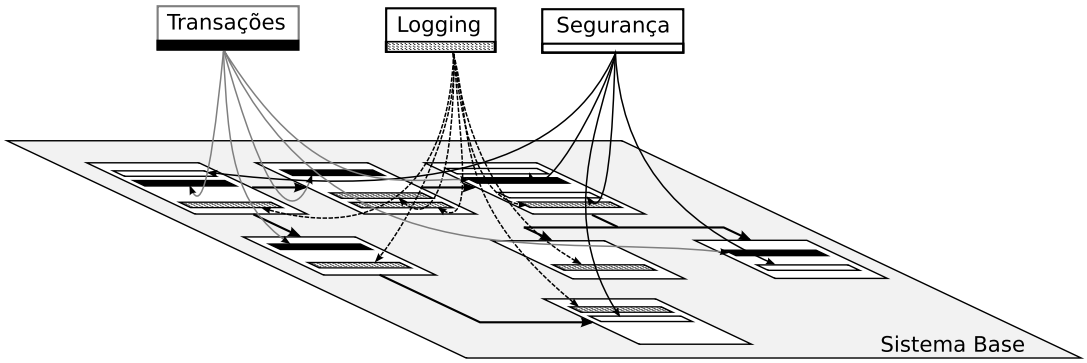


Figura 2.2.: Esse diagrama mostra como seria o sistema da Figura 2.1 se ele fosse orientado a aspectos. Cada funcionalidade ortogonal fica à parte do restante do sistema, encapsulada em uma unidade denominada aspecto. O restante do sistema, composto por classes comuns, é o que denominamos de sistema base.

```
4     public Object advice(Invocation invocation) throws Throwable
5     {
6         boolean thrown = false;
7         transaction.begin();
8         try
9         {
10            return invocation.invokeNext();
11        }
12        catch (Throwable t)
13        {
14            transaction.rollback();
15            thrown = true;
16        }
17        finally
18        {
19            if (!thrown)
20                transaction.commit();
21        }
22    }
23 }
```

Listagem 2.2: Suporte a transação encapsulado em um aspecto.

Naturalmente, é preciso que as funcionalidades encapsuladas em aspectos sejam executadas nos momentos apropriados, da mesma forma que no sistema original, da Figura 2.1. A POA é responsável por combinar (*weave*) de forma automática os aspectos com as unidades de um sistema base, garantindo

a reutilização de todas as partes envolvidas. Esse processo é conhecido como **combinação de aspectos** e é dito **dinâmico** quando ocorre em tempo de execução.

A combinação de aspectos a um sistema base resulta na execução dos adendos em pontos determinados do fluxo de controle do sistema base. Esses pontos são denominados **pontos de junção** (*join points*) [3]. Os pontos de junção podem ser a execução ou a chamada de métodos, construtores, a leitura ou escrita do valor de um campo, um bloco *catch*, e assim por diante. O conjunto de pontos de junção disponível depende da ferramenta de POA que se estiver utilizando. Note que a seleção dos pontos de junção a serem combinados com um determinado adendo não é realizada de forma arbitrária. Uma expressão, chamada ***pointcut***, é utilizada para identificar os pontos de junção afetados pelo adendo em questão. Essa expressão funciona como uma espécie de expressão regular de pontos de junção. Quando um *pointcut* identifica positivamente um ponto de junção, dizemos que ele casa (*match*) com aquele ponto de junção.

Voltando ao nosso exemplo de implementação de suporte a transações, para combinar o adendo `advice`, do aspecto `TransactionAspect`, com o sistema base, poderíamos utilizar o *pointcut*:

```
"execution(public * *Operation->execute(..))"
```

A sintaxe de um *pointcut* é definida pela própria ferramenta de POA que se estiver utilizando. Os elementos disponibilizados por essa sintaxe determinam o conjunto dos pontos de junção que podem ser casados com um adendo, como construtores, métodos e acessos a campos. O exemplo acima segue a sintaxe do JBoss AOP e indica que o adendo deve ser executado quando os métodos `execute`, contidos nas classes cujo nome termina com o sufixo `Operation`, forem executados. Dizemos que o adendo **intercepta** esses pontos de junção, fenômeno conhecido como **interceptação**.

A interceptação não é o único recurso fornecido pela POA. Para implementar uma funcionalidade ortogonal, pode ser também necessária a existência de métodos auxiliares nas unidades do sistema base. Em linguagens base orientadas a objetos, é possível inserir métodos ou funções em classes (ao contrário dos adendos, esses métodos não estão associados a pontos de junção). Sempre que esses métodos forem invocados numa instância da classe alterada, a sua execução será delegada a um objeto específico, o ***mixin***. Esse recurso é denominado **introdução de *mixins***.

Um terceiro recurso comumente suportado pela maioria das linguagens e ferramentas orientadas a aspectos é a adição de meta-dados a métodos e classes, conhecido como **introdução de meta-dados**. Nas ferramentas Java, esse recurso é geralmente suportado na forma de **introdução de anotações**, onde o meta-dado a ser introduzido é uma anotação Java.

Na próxima seção, vamos conhecer as principais ferramentas de programação orientada a aspectos.

[3] A definição de pontos de junção depende do modelo adotado. No nosso trabalho, utilizamos o modelo de pontos de junção dinâmico do AspectJ, que diz que pontos de junção são a execução de uma linha de código, e não a linha de código em si. Escolhemos esse modelo por ser o mais comumente utilizado em textos sobre combinação dinâmica de aspectos.

2.3. Ferramentas e Linguagens

Existe uma vasta gama de ferramentas e linguagens de programação orientada a aspectos. Dentre elas, podemos citar DAO C++ [26], CaesarJ [16, 63], JAC [35, 64], AspectWerkz [7, 14], JasCO [73], AspectJ [6, 49], Prose [65, 68] e JBoss AOP [20, 38].

Dentre estas linguagens, merece destaque o AspectJ, por ser a primeira linguagem orientada a aspectos baseada em Java. O AspectJ adquiriu grande força acadêmica com o advento do abc [8], um compilador de AspectJ extensível. O abc foi desenvolvido especialmente para facilitar pesquisas na área.

Todavia, o AspectJ não provê combinação dinâmica de aspectos [4]. Dentre as ferramentas que suportam esse recurso, podemos destacar o PROSE, o AspectWerkz e o JBoss AOP. A primeira é totalmente voltada para a combinação dinâmica e é a única que não requer anotações ou arquivos XML para a declaração de aspectos e adendos. O nome PROSE é a sigla de *PROgrammable extenSions of sErvices*. Já o AspectWerkz foi uma ferramenta muito popular no início desta década. Porém, seu código não é editado há mais de três anos, desde que se iniciou o processo de incorporação do mesmo com o AspectJ. Finalmente, o JBoss AOP é uma ferramenta que, além de ser utilizada de forma independente, é amplamente empregada pelos diversos projetos do grupo JBoss, como por exemplo o JBoss Cache, o JBoss Messaging e o JBoss Microcontainer (esse último será o novo núcleo do servidor de aplicações JBoss, a partir da versão 5).

O JBoss AOP é a ferramenta que iremos utilizar como caso de teste para a nossa pesquisa. Suas funcionalidades e mecanismos serão detalhados no próximo capítulo.

2.4. JBoss AOP

Como já foi visto anteriormente (página 7), um aspecto no JBoss AOP é uma classe Java comum e, um adendo, um método dessa classe. Para que o JBoss AOP reconheça que uma classe é um aspecto, basta declará-la como tal em um arquivo denominado `jboss-aop.xml`, ou ainda anotá-la com `@org.jboss.aop.Aspect`. Os adendos e quaisquer outros recursos disponíveis pela ferramenta também devem ser configurados no arquivo XML ou através de anotações.

Detalhes como os que acabamos de descrever serão vistos nesta seção. Apresentaremos os elementos envolvidos na interceptação, principal recurso do JBoss AOP. Esses elementos são: pontos de junção, adendos, interceptadores e associações. Quanto aos *pointcuts*, sua sintaxe será descrita mais adiante, na Seção 6.1.1.1 na página 71. Daremos início com os pontos de junção suportados pelo JBoss AOP.

2.4.1. Pontos de Junção

Os pontos de junção que podem ser interceptados pelo JBoss AOP são:

- execuções de método e de construtores;

[4] O AspectJ prevê a adição de suporte à combinação dinâmica de aspectos em uma versão futura.

- chamadas de métodos e de construtores;
- leituras e escritas de valores de campos.

Observe que existe uma diferença sutil entre pontos de junção do tipo execução e pontos do tipo chamada. No primeiro tipo, a interceptação ocorre dentro do contexto do método ou construtor sendo executado, de forma que é possível ter acesso aos elementos desse contexto (como objeto alvo [5], por exemplo). Já os pontos de junção do tipo chamada são interceptados dentro do contexto onde é realizada a chamada e é possível acessar o objeto que faz a chamada (caso não seja um contexto estático). Apesar de sutil, essa diferença é importante e também se reflete no desempenho da combinação: é mais rápido combinar um adendo a um único ponto, a execução de um método ou construtor, do que combiná-lo a todos os pontos que fazem uma chamada àquele método ou construtor. Os pontos do tipo chamada devem utilizados quando: há interesse de restringir os pontos interceptados através do método ou construtor que faz a chamada (por exemplo, deseja-se interceptar somente as chamadas feitas por métodos que recebem um único argumento, do tipo `int`); é necessário ter acesso ao contexto da chamada; ou não é possível instrumentar a execução do método ou construtor chamado, já que ele está numa biblioteca e não poderá ser instrumentado (somente no caso de pré-compilação).

No JBoss AOP, os pontos de junção são representados por objetos, denominados *join point beans* e podem ser acessados pelos adendos. O tipo do objeto que representa um determinado ponto de junção depende do seu tipo e do tipo do seu adendo. Como é possível ver na tabela 2.1 na página seguinte, existem dois tipos de *join point beans*:

info beans: localizados no pacote `org.jboss.aop`, possuem como super classe o *bean* `org.jboss.aop.JoinPointInfo`. Eles contêm apenas informações reflexivas sobre o ponto de junção a ser interceptado. Esses *beans* estão disponíveis para os adendos do tipo *before*, *after*, *throwing* e *finally*, que serão detalhados na Seção 2.4.2 na próxima página.

invocation beans: esses beans estão situados no pacote `org.jboss.aop.joinpoint` e possuem como super classe comum `org.jboss.aop.joinpoint.Invocation`. São o tipo mais complexo de *bean* e, diferentemente do tipo anterior, eles contêm os meta-dados associados ao ponto de junção que representam. Os meta-dados contidos num *invocation bean* podem ter sido carregados de três formas diferentes, a saber: a partir do código fonte (*doclets* e anotações); através de introduções de meta-dados; ou adicionados pelos próprios adendos através da interface da classe `Invocation`. A utilização de *invocation beans* causa um impacto maior no desempenho se comparada com o uso de *info beans*, pois aqueles contêm mais dados e é necessário uma criação de um *invocation bean* a cada interceptação. Eles são utilizados somente para a interceptação de adendos do tipo *around*, que serão apresentados a seguir.

[5] O significado semântico do objeto alvo depende do ponto de junção em questão. No caso de execução de métodos, o objeto alvo é o objeto que está executando o método. Note que o objeto alvo só é valido em contextos não estáticos, de forma que a execução de métodos estáticos não está associada a um objeto alvo.

Ponto de Junção	***Bean* do tipo `Invocation`**	**Bean do tipo `Info`**
leitura de campos	`FieldReadInvocation`	`FieldAccess`
escrita de campos	`FieldWriteInvocation`	`FieldAccess`
execução de métodos	`MethodInvocation`	`MethodExecution`
execução de construtores	`ConstructorInvocation`	`ConstructorExecution`
chamada de métodos	`MethodCalledByMethodInvocation`, `MethodCalledByConstructor Invocation`	`MethodCallByMethod`, `MethodCallByConstructor`
chamada de construtores	`ConstructorCalledByMethod Invocation`, `ConstructorCalledByConstructor Invocation`	`ConstructorCallByMethod`, `ConstructorCallByConstructor`

Tabela 2.1.: *Beans* que representam pontos de junção no JBoss AOP.

2.4.2. Adendos

São cinco os tipos de adendos suportados:

before: executado antes do ponto de junção;

after: executado depois do ponto de junção, somente se esse finalizar sua execução sem lançar exceção;

throwing: executado após o ponto de junção, somente se esse lançar alguma exceção;

finally: executado após o ponto de junção, independe da forma como esse termina;

around: executado "ao redor"do ponto de junção, esse adendo engloba o ponto de junção e pode até mesmo substituí-lo no fluxo de controle do sistema.

A assinatura dos adendos é flexível e deve seguir a forma geral:

`public [`*`return type`*`] [`*`advice name`*`]([`*`annotated parameter`*`],...[`*`annotated parameter`*`])` (2.4.1)

Onde:

`return type` : o tipo de retorno pode ser não `void` somente para os adendos do tipo *around*, *after*, *finally*. Quando isso ocorre, o valor de retorno do adendo substituirá o valor de retorno do ponto de junção no sistema base. Nesse caso, o tipo do retorno pode ou seguir a forma genérica `Object` (serve para qualquer ponto de junção), ou ter tipo de retorno igual ao do ponto de junção interceptado.

`advice name` : o nome do adendo é livre e não influencia o funcionamento do JBoss AOP

`annotated parameter` : o adendo pode ter na sua assinatura um ou mais parâmetros anotados. As anotações dos parâmetros indicarão para o JBoss AOP quais valores devem ser passados para o adendo. Na tabela 2.2 na página 14 é possível ver as anotações que podem ser usadas nesse caso.

Observe que o tipo de um adendo não é especificado na sua assinatura. Do contrário, o tipo do adendo é especificado em associações, tópico que será abordado na Seção 2.4.5 na página 16.

Alguns exemplos de adendos podem ser vistos na Listagem 2.3:

```
public int advice1(@Arg int a, @Arg int b, @Return int returnedValue)
public void advice2(@Caller Object callerObject,
                                        @Target Object calledObject)
public void advice3(@JoinPoint MethodInvocation invocation)
public void advice4(@Args Object[] args,
                        @JoinPoint ConstructorExecution joinpointBean)
```

Listagem 2.3: Assinaturas de adendos.

O adendo `advice1` recebe os dois argumentos do ponto de junção, ambos do tipo `int`, e o valor que o ponto de junção devolveu. Como esse adendo devolve um valor do tipo `int`, sabemos que ele sobrescreve o valor de retorno do ponto de junção, que deve ser do mesmo tipo. Por ter um parâmetro com a anotação `@Return`, esse adendo tem que ser do tipo *after* ou do tipo *finally*, como mostra a tabela a tabela 2.2.

No segundo caso, o `advice2` recebe o objeto chamador e o alvo dos pontos de junção que intercepta. Por receber o chamador, esse adendo só pode ser utilizado para interceptar pontos de junção do tipo chamada.

O terceiro adendo, `advice3`, recebe o *join point bean* do tipo `MethodInvocation`. Por se tratar de um *invocation bean*, esse adendo é válido somente para interceptações do tipo *around*.

Finalmente, o adendo `advice4` recebe os argumentos do ponto de junção no vetor `args`, e o *info bean* `ConstructorExecution` (novamente, podemos concluir pela assinatura do adendo algumas informações sobre o tipo de interceptação que ele executa; no caso, sabemos que ele intercepta execuções de construtores, e que ele não é do tipo *around*).

Os adendos do tipo *around* podem também ter uma assinatura simplificada, sem anotações, denominada **assinatura padrão de adendos**:

```
public Object [advice name]([invocation bean]) throws Throwable        (2.4.2)
```

Onde ***advice name*** pode ser escolhido de forma arbitrária e ***invocation bean*** deve ser um dos *beans* mostrados na tabela 2.1 na página anterior, ou a super classe comum `Invocation`.

Exemplos da assinatura padrão podem ser vistos na listagem a seguir.

```
public Object advice1(Invocation invocation) throws Throwable
public Object advice2(MethodInvocation invocation) throws Throwable
public Object advice3(FieldReadInvocation invocation) throws Throwable
```

Listagem 2.4: Assinaturas de adendos.

Veremos como o JBoss AOP identifica os adendos e seus tipos na Seção 2.4.5 na página 16.

Anotação	Semântica	Adendos				
		bf	**ar**	**af**	**th**	**fn**
`@JoinPoint`	o *join point bean* que representa o ponto de junção	•	•	•	•	•
`@Target`	o objeto alvo do ponto de junção	•	•	•	•	•
`@Caller`	o objeto chamador do ponto de junção	•	•	•	•	•
`@Thrown`	a exceção lançada pelo ponto de junção				•	•
`@Return`	o valor devolvido pelo ponto de junção			•		•
`@Arg`	um dos argumentos recebidos pelo ponto de junção	•	•	•	•	•
`@Args`	todos os argumentos recebidos pelo ponto de junção	•	•	•	•	•

Tabela 2.2.: Anotações de parâmetros de adendos. A primeira coluna mostra os nomes das anotações disponíveis (todas pertencem ao pacote `org.jboss.aop.advice.annotations`) e a segunda mostra o significado semântico das mesmas. Na terceira coluna é possível ver quais adendos suportam cada anotação (os tipos de adendos *before*, *around*, *after*, *throwing* e *finally* foram abreviados como *bf*, *ar*, *af*, *th* e *fn*, respectivamente).

2.4.3. Interceptadores

No JBoss AOP interceptadores são um tipo especial de aspecto, que contém somente um adendo, do tipo *around*.

Todo interceptador deve implementar a interface `org.jboss.aop.Interceptor`, que define a assinatura do adendo como sendo:

```
public Object invoke(Invocation invocation) throws Exception        (2.4.3)
```

Como um interceptador contém apenas um adendo, com assinatura fixa, o termo interceptador é também utilizado para se referenciar a um adendo na documentação e API do JBoss AOP.

Os interceptadores são largamente utilizados dentro do JBoss AS (*JBoss Application Server*) para a implementação de serviços como transação e segurança.

2.4.4. Interceptação do tipo `around`

A interceptação realizada por adendos do tipo `around` e interceptadores possui diversas peculiaridades, a começar pelo uso de `join point beans` do tipo `Invocation`.

Nesta seção nos dedicaremos a explicar os mecanismos envolvidos nesse tipo especial de interceptação.

Diferentemente dos outros tipos de interceptação, que ocorrem num único ponto definido (antes ou depois do ponto de junção), a interceptação do tipo *around* **engloba** o ponto de junção. Isso significa que ela executa antes e depois do ponto de junção, além de ser a única que pode substitui-lo completamente, impedindo sua execução.

Quando um ponto de junção é interceptado por um ou mais adendos do tipo *around*, ele está associado a uma pilha formada por esses adendos. Denominaremos essa pilha de **pilha *around***. Durante

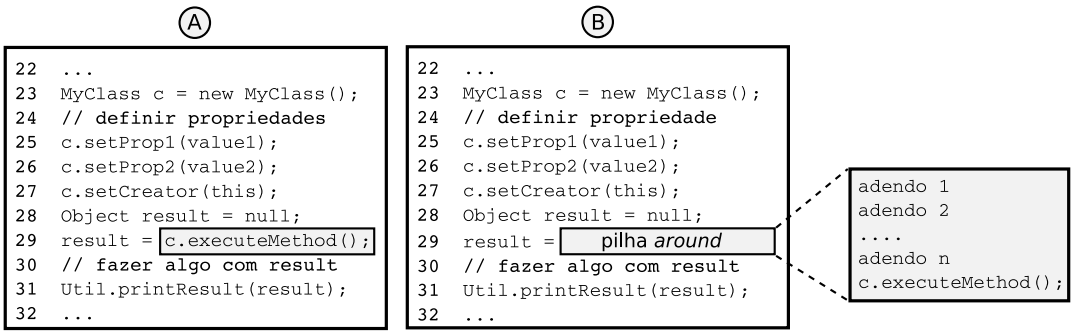


Figura 2.3.: Um exemplo de englobamento. Na parte A, um trecho de código é mostrado. Na parte B, a chamada realizada na linha 29 foi englobada por uma pilha *around*

.

a execução do sistema, os pontos de junção que estiverem associados a uma pilha *around* não vazia terão a sua execução substituída pela execução da própria pilha, fenômeno que denominamos de **englobamento**. Dizemos que uma pilha *around* engloba o ponto de junção ao qual ela está associada.

O englobamento é representado pela Figura 2.3. Na parte A, podemos ver um trecho de código do sistema base antes de ter sido transformado pelo JBoss AOP. Observe a chamada realizada na linha 29. Na parte B da figura, vemos que essa chamada foi englobada por uma pilha *around*, composta pelos adendos de `1` a `n`. Além de conter os adendos, a pilha contém o próprio ponto de junção englobado. Quando essa linha for executada, a pilha *around* será invocada. Sua execução é feita de forma a invocar cada um dos adendos, na mesma ordem em que eles seriam desempilhados (daí o nome pilha). Após a execução do último adendo, `adendo n`, o ponto de junção englobado é executado. A seguir, a execução retorna para os adendos na ordem inversa em que foram executados, como se eles estivessem sendo empilhados novamente. Durante esse processo, cada item da pilha tem acesso ao valor devolvido pelo item anterior (que pode ser o ponto de junção ou um outro adendo, dependendo da sua posição na pilha), e deve devolver como resultado um valor para o item seguinte. O valor devolvido pelo último adendo é repassado para o sistema base como sendo o valor de retorno do ponto de junção englobado. Isso permite que os adendos do tipo *around* substituam o valor que o ponto de junção devolveu, capacidade também disponível para os adendos do tipo *after*.

A pilha `around` fica contida num *bean invocation*. Esse *bean*, utilizado somente pelos adendos do tipo *around* e interceptadores, permite que se execute o próximo item da pilha, através do método `invokeNext()`. Esse método devolve exatamente o que o item invocado devolveu, e também repassa exceções. Veja o esqueleto de um adendo do tipo *around*:

```
1 public Object advice(Invocation invocation) throws Throwable
2 {
3     // fazer algo antes da execução do ponto de junção
4     ...
5     try
```

```
6       {
7           Object returnedValue = invocation.invokeNext();
8           // fazer algo depois da execução do ponto de junção
9           ...
10          // retorna valor de retorno
11          return returnedValue;
12      }
13      catch(Throwable t)
14      {
15          // fazer algo quando uma exceção foi lançada
16          ...
17      }
18      finally
19      {
20          // fazer algo após o ponto de junção, mesmo que
21          // uma exceção tenha sido lançada
22          ...
23      }
24  }
```

Listagem 2.5: Esqueleto de um adendo do tipo *around*.

No esqueleto da Listagem 2.5 fica claro que, com o adendo do tipo *around*, obtemos todas as funcionalidades dos adendos dos outros tipos. O bloco que executa antes da chamada a `Invocation.invokeNext()` equivale a um adendo do tipo *before*. O bloco que se seque, a um adendo do tipo *after*. Dentro do bloco `catch`, temos um adendo do tipo `throwing`, com exceção de que esse último não pode tratar exceções, ao contrário do adendo do tipo *around*. Por último, temos o bloco `finally`, que seria equivalente a um adendo do tipo `finally`. Para substituir a execução do ponto de junção englobado, basta omitir a chamada ao método `Invocation.invokeNext()`. Se houver outros adendos na pilha a serem executados, essa omissão impedirá também que esses adendos sejam executados.

Uma vez que a implementação dos adendos do tipo *around* é custosa (devido à necessidade de criação de um objeto `Invocation` a cada execução), aconselha-se o uso dos outros tipos de adendo sempre que possível. Porém, em alguns casos, não é possível substituir um código do tipo *around* por dois ou mais de outros tipos. Um exemplo seria um adendo que simplesmente coloca a chamada a `invokeNext()` dentro de um bloco `synchronized`. Além disso, a API de `Invocation` é mais rica, permitindo inclusive a passagem de metadados de um adendo para o outro. Para acessar esse recurso, é preciso utilizar um adendo do tipo *around*.

2.4.5. Associação de Adendos e Pointcuts

Até esse momento, nos contivemos em mostrar apenas a sintaxe dos adendos. De igual importância é a declaração de **associações** ou *bindings*, que permite a identificação dos adendos, e a sua associação

com *pointcuts*.

Uma associação conecta o adendo a um `pointcut`, indicando que o adendo em questão deve interceptar todos os pontos de junção que casarem com o `pointcut` associado. Uma associação pode ser declarada em um arquivo `jboss-aop.xml` ou através de anotações. Veja o trecho de um arquivo xml abaixo:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <aop>
3     ...
4     <bind pointcut="execution(POJO->new())">
5         <around name="constructorAdvice" aspect="MyAspect"/>
6         <before name="otherAdvice" aspect="MyAspect"/>
7     </bind>
8     ...
9 </aop>
```

Listagem 2.6: Exemplo de declaração de uma associação num arquivo `jboss-aop.xml`.

Esse trecho declara uma associação, conectando o *pointcut* `"execution(POJO->new())"` com os adendos `constructorAdvice` e `otherAdvice` do aspecto cujo nome é `MyAspect`. Essa associação indica que, sempre que o construtor *default* de `POJO` for executado, os adendos especificados deverão ser executados. O primeiro adendo é do tipo `around`, enquanto que o segundo, do tipo `before`. Similarmente, utiliza-se as *tags* `after`, `throwing` e `finally` para associar adendos do tipo equivalente. Agora veja o exemplo abaixo:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <aop>
3     ...
4     <bind pointcut="execution(POJO->new())">
5         <advice name="constructorAdvice" aspect="MyAspect"/>
6         <before name="otherAdvice" aspect="MyAspect"/>
7     </bind>
8     ...
9 </aop>
```

Listagem 2.7: Exemplo de declaração de associação com um adendo do tipo padrão.

Ele difere do anterior na *tag* `advice`, que foi colocada no lugar da *tag* `around`. Essa *tag* declara um adendo sem especificar o seu tipo. Nesse caso, o JBoss AOP assume que o adendo é do tipo `around`, que é o tipo padrão de adendos.

Ambos os exemplos anteriores estão incompletos. `"MyAspect"` é o nome de um aspecto, que precisa ter sido declarado anteriormente numa `tag aspect`, como mostra a linha abaixo:

```
<aspect class="mypackage.MyAspect" name="MyAspect" scope="PER_VM"/>
```

No exemplo, vemos uma declaração de um aspecto cuja classe é `mypackage.MyAspect`, com o nome `"MyAspect"`. Esse nome deverá ser utilizado nas declarações de `binding` que utilizarem esse aspecto, como foi feito nos exemplos anteriores. O último atributo, `PER_VM`, define o escopo que cada instância do aspecto deverá abranger. Existem cinco tipos de escopo, a saber:

PER_VM : uma única instância do aspecto será criada para realizar todas as interceptações durante a execução de uma instância da máquina virtual;

PER_CLASS : será criada uma instância do aspecto para cada classe que possuir pontos de junção a serem interceptados;

PER_INSTANCE : o JBoss AOP criará uma instância do aspecto para cada objeto ou instância que possuir pontos de junção a serem interceptados. Note que, nesse escopo, pontos de junção estáticos não serão interceptados, por não estarem associados a uma instância.

PER_JOINPOINT : uma instância do aspecto será criada para cada ponto de junção que ele interceptará. Essa é a menor granularidade que podemos obter em termos de escopo de aspectos.

PER_CLASS_JOINPOINT : similar ao escopo anterior, uma instância do aspecto será criada para cada ponto de junção de uma classe.

Um escopo diferente de `PER_VM` deve ser utilizado somente quando houver necessidade de o aspecto conter informações referentes ao escopo que ele pertence. Por exemplo, é necessário utilizar o escopo `PER_INSTANCE` num aspecto que deve sincronizar, com um *lock*, o acesso às informações contidas em cada objeto a ser interceptado. Nesse caso, é preciso um *lock* exclusivo para cada objeto, o que pode ser facilmente obtido ao se utilizar o escopo `PER_INSTANCE` se o *lock* estiver contido num campo do aspecto.

Note, também, a diferença sutil que existe entre os escopos `PER_JOINPOINT` e `PER_CLASS_JOINPOINT`. Vamos ilustrá-la através de um exemplo. Considere o aspecto `MyAspect`, que será utilizado para interceptar a execução de um método, `void someMethod()`, contido na classe `Pojo`. Se o escopo de `MyAspect` for `PER_CLASS_JOINPOINT`, uma única instância de `MyAspect` será criada para interceptar a execução do método em questão, instância que será utilizada em todos os objetos da classe `Pojo`. Por outro lado, se o escopo de `MyAspect` for `PER_JOINPOINT`, haverá uma instância do aspecto para interceptar esse método para cada objeto de `Pojo`. Nesse caso, a interceptação do método `someMethod()` será realizada pelo aspecto associado à instância de `Pojo` que estiver sendo invocada.

O uso do atributo `scope` na *tag* `aspect` não é obrigatório. Quando esse atributo é omitido, o JBoss AOP assume que o escopo é o escopo padrão, `PER_VM`. Ademais, o atributo `name` também é opcional. Se omitido, o nome do aspecto será o nome de sua classe.

É possível também declarar um aspecto sem especificar a sua classe, e sim uma fábrica ou *factory*. Nesse caso, ao invés de declarar o valor do atributo `class`, utiliza-se o atributo `factory`, que deve conter o nome da classe da fábrica (essa classe precisa implementar a interface `org.jboss.aop.advice.AspectFactory`). Essa opção é interessante quando é preciso aplicar regras na criação do aspecto de acordo com o escopo para o qual ele será criado. É possível até mesmo criar aspectos de classes diferentes para cada classe, instância, ou ponto de junção a ser interceptado.

Além do XML, é possível declarar associações é utilizando-se anotações. Nesse caso, a anotações serão feitas nos próprios adendos, e conterão apenas a expressão *pointcut* e o tipo do adendo:

```
1  @Aspect
2  public class MyAspect
3  {
4      @Bind (pointcut="execution(POJO->new())",type=AdviceType.AROUND)
5      public Object constructorAdvice(ConstructorInvocation invocation)
6                                                          throws Throwable
7      {
8          ...
9      }
10 }
```

Listagem 2.8: Exemplo de declaração de uma associação num arquivo Java.

A linha 4 contém a anotação `@Bind`, declarando uma associação do adendo `constructorAdvice`, com o *pointcut* `"execution(POJO->new())"`. Se o tipo do adendo é omitido, o valor padrão, `AdviceType.AROUND`, será utilizado. Assim como é preciso declarar os aspectos no arquivo xml, é também preciso fazê-lo ao usar anotações. A anotação `@Aspect`, no código acima, declara que a classe `MyAspect` é um aspecto. Ela possui os atributos opcionais `name` e `scope`, correspondentes aos atributos de mesmo nome da *tag* xml `aspect`.

A associação de interceptadores também é declarada de forma semelhante à associação de adendos. Utilizando-se xml, ela segue a forma do exemplo a seguir:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aop>
3      ...
4      <bind pointcut="execution(POJO->new())">
5          <interceptor class="mypackage.MyInterceptor"/>
6      </bind>
7      ...
8  </aop>
```

Listagem 2.9: Exemplo xml de declaração de *binding* de interceptadores.

Note que a *tag* `interceptor` possui o atributo opcional `scope`. Esse atributo funciona da mesma forma que na declaração de aspectos. Como ele foi omitido no exemplo anterior, o escopo do interceptador declarado é o escopo padrão, `PER_VM`.

Diferente do aspecto, não é preciso declarar o interceptador antes de utilizá-lo em uma associação. O equivalente em anotações é mostrado no próximo exemplo:

```
1  @InterceptorDef
2  @Bind(pointcut="execution(POJO->new())")
```

```
3 public class MyInterceptor
4 {
5     public Object invoke(Invocation invocation) throws Throwable
6     {
7         ...
8     }
9 }
```

Listagem 2.10: Exemplo de declaração de *binding* de interceptadores através de anotações.

A anotação `@Bind` é declarada na própria classe do interceptador, já que ele possui apenas um adendo, de assinatura definida. É também preciso anotar a classe com `@InterceptorDef` para indicar ao JBoss AOP que se trata de um interceptador, e não um aspecto comum.

2.4.6. Outros recursos

Além da funcionalidade básica de interceptação, através da associação de *pointcuts* com adendos, existem outros recursos avançados disponíveis no JBoss AOP. Exemplos são expressões `cflow`, precedência de adendos, declaração de regras, introdução de interfaces, metadados, além de outros. A lista completa é apresentada de forma breve no Apêndice A.

2.5. Combinação de Aspectos

Ao analisar as ferramentas de POA em Java, nota-se que as linguagens *pointcut* e a sintaxe dos aspectos variam de uma ferramenta para outra. O AspectJ e o JasCO estendem a linguagem Java para permitir a declaração desses elementos. Já o JBoss AOP e o AspectWerkz se utilizam de XML e anotações para isso. No PROSE, por outro lado, chamadas a métodos de sua API são utilizadas para declarar *pointcuts*. Apesar dessa diferença de sintaxe, veremos que a implementação dessas ferramentas têm as suas similaridades.

Sabemos que, ao utilizarmos POA, as funcionalidades ortogonais deverão ficar à parte do sistema base, encapsuladas em aspectos. Mas, no momento da execução, essas funcionalidades têm de executar no sistema base da mesma forma que o fariam caso não estivessem encapsuladas em aspectos. Em outras palavras, o sistema da Figura 2.2 (página 8) tem que se comportar de forma igual ao sistema da Figura 2.1 (página 7).

De modo geral, podemos afirmar que o objetivo das ferramentas de POA é: prover meios de encapsular funcionalidades ortogonais em aspectos através de sua sintaxe; e combinar aspectos ao sistema base, processo que denominamos **combinação de aspectos**. A fim de garantir que os aspectos sejam executados nos momentos corretos, a maioria das ferramentas Java utiliza a **instrumentação**, uma técnica que consiste na manipulação de *bytecodes*. Todas essas ferramentas Java que citamos na seção anterior, como o AspectJ, JBoss AOP, AspectWerkz e JasCO, alteram o código do sistema base, inserindo as funcionalidades ortogonais nos pontos do código onde elas devem executar.

A instrumentação pode ser realizada em dois momentos diferentes:

- antes da execução: a ferramenta POA fornece um compilador que processa os `bytecodes` para realizar a instrumentação. Nesse caso, a instrumentação também é conhecida como **compilação de aspectos**.
- em tempo de carga: à medida que as classes são carregadas no sistema, o seu código é instrumentado. Essa instrumentação é feita no momento da carga, antes de os *bytecodes* serem processados pelo `ClassLoader`. Existem diversas técnicas que possibilitam esse tipo de instrumentação, como a adição de um gancho no código de `ClassLoader`, a utilização de um `ClassLoader` próprio, ou através da API `java.lang.instrument`. Mostraremos como as diversas ferramentas realizam instrumentação em tempo de carga no próximo capítulo.

Apesar de ser mais comumente utilizada, a instrumentação não é a única abordagem conhecida para a implementação de ferramentas POA em Java. O JBoss AOP, por exemplo, suporta também a combinação através da geração de *proxies* dinâmicos, o que será visto em mais detalhes na próxima seção. O PROSE também fornece uma alternativa à instrumentação. Essa ferramenta insere *breakpoints* nos pontos do código que deverão ser interceptados, dispensando a necessidade de instrumentação. Quando esses *breakpoints* são atingidos, durante a execução de um programa, o PROSE avalia quais *pointcuts* casam com o ponto de junção que será executado, e chama os adendos associados a esses *pointcuts*. Para inserir os *breakpoints*, o PROSE utiliza a JVMDI, *Java Virtual Machine Debug Interface*, API disponibilizada para a implementação de depuradores de programas escritos em Java [44]. Como conseqüência, é preciso executar o sistema em modo de depuração, acarretando perda de desempenho [65].

De modo geral, o termo **combinação** envolve também a combinação de outras funcionalidades ao sistema base (para exemplos, vide o Apêndice A). Nesse trabalho, no entanto, nos focaremos na combinação de aspectos apenas.

2.5.1. Combinação no JBoss AOP

A fim de mostrar com mais detalhes os mecanismos que envolvem a combinação de aspectos, essa seção fará uma análise da sua implementação no JBoss AOP. Como dito há pouco, a combinação no JBoss AOP pode ser realizada em dois momentos: compilação ou tempo de carga.

A combinação em tempo de compilação requer o uso de um compilador fornecido pela ferramenta, denominado `aopc`. Esse compilador processa os arquivos `.class`, transformando-o seu código de forma a combinar os aspectos e interceptadores. Os arquivos `.class` resultantes compõem o sistema orientado a aspectos resultante. Durante a sua execução, eles utilizarão classes da API do JBoss AOP para realizar a interceptação.

A outra possibilidade é a combinação em tempo de carga. Essa opção se utiliza da **JVMTI** (*Java Virtual Machine Tool Interface*) [45], uma interface de programação disponível para ferramentas de desenvolvimento e monitoração. A JVMTI foi adicionada à especificação da linguagem Java a partir da versão 1.5.

Para utilizar essa opção, é preciso executar o sistema base com a opção `-javaagent:aop50.jar`. Essa opção ativa o **agente** do JBoss AOP, contido no arquivo `aop50.jar`, fornecido como parte da distribuição do JBoss AOP. Um agente é um cliente da JVMTI. Ele é uma classe comum Java e deve implementar o método:

```
public static void premain(String, Instrumentation);
```

Através desse método, o agente é inicializado e tem acesso às funcionalidades fornecidas pela classe `java.lang.instrument.Instrumentation`. Essa classe é o componente central da JVMTI, e suporta, dentre outras coisas, o registro de objetos `java.lang.instrument.Class- FileTransformer`. Esses objetos são notificados pela JVMTI quando da carga de uma classe, e têm a oportunidade de alterar os *bytecodes* da mesma antes que ela seja carregada por um *class loader*. É exatamente isso que o JBoss AOP faz para implementar a instrumentação em tempo de carga, registra um objeto `ClassFileTransformer` que, quando notificado, delegará a transformação da classe ao `AspectManager`.

A combinação não é a única tarefa executada pelo JBoss AOP. Podemos dividir a execução do JBoss AOP em três fases distintas: **configuração**, **combinação** e **interceptação**. A primeira antecede a combinação e, assim como ela, pode ser realizada em tempo de compilação ou de carga. A terceira, interceptação, ocorre durante a execução do sistema orientado a aspectos e é resultante da combinação.

A configuração consiste no processamento do arquivo de configuração `jboss-aop.xml` e no processamento de anotações. Após essa leitura, o JBoss AOP armazena todas as informações lidas na memória (aspectos, adendos, *pointcuts*, introduções e outros).

A seguir, tem início a combinação, onde essas informações são utilizadas para que os aspectos sejam combinados ao sistema base. Na Seção 2.5, vimos que o JBoss AOP instrumenta os *bytecodes* para combinar aspectos ao sistema base. Isso é feito através do Javassist [37], uma ferramenta do JBoss que permite a instrumentação através de uma API de alto nível, sem necessidade de manipulação dos *bytecodes* em seu formato original. O Javassist recebe código Java dentro de `String`s e é responsável por compilá-lo para gerar *bytecodes*.

Durante essa fase, os *pointcuts* precisam ser casados com os pontos de junção, a fim de determinar quando os adendos devem ser executados. Na Seção 2.2, definimos os pontos de junção como sendo pontos no fluxo de controle. Isso significa que esses pontos só são conhecidos durante a execução do sistema. Tal afirmação nos leva à pergunta: como é possível casar os *pointcuts* com os pontos de junção antes da execução do programa? A resposta é que isso não é possível, mas é possível casá-los com as chamadas **sombras de junção** (*join point shadows*). A sombra de junção é uma linha de código cuja execução resultará em um ou mais pontos de junções. Ao analisarmos a sombra de junção, podemos inferir diversos dados sobre os pontos de junção que ela irá gerar. Por exemplo, abaixo vemos uma chamada ao construtor *default* de `StringBuffer`:

```
...
StringBuffer sb = new StringBuffer();
sb.append("new message");
```

...

Listagem 2.11: Um trecho de código do sistema base, onde uma chamada ao construtor de `StringBuffer` é realizada, seu resultado assinalado a uma variável denominada `sb` e, finalmente, o texto `"new message"` é adicionado a esse *buffer*.

Sabemos que esse trecho do código sempre irá gerar pontos de junção do tipo chamada, onde o objeto chamado é o construtor em questão. Se analisarmos o contexto onde é feita a chamada, podemos também descobrir quem é o chamador. Com base nessas informações, é possível casar a sombra de junção com expressões *pointcuts*. Porém, existem informações que não podem ser facilmente determinadas antes da execução. Por exemplo, não podemos inferir, por essa linha, quais métodos e construtores estarão na pilha de execução durante as várias vezes que essa linha provavelmente será executada. Logo, afirmamos que é possível apenas casar parcialmente um *pointcut* com essa linha. Se o *pointcut* determinar uma restrição que só puder ser avaliada durante a execução, será necessário efetuar uma checagem em tempo de execução. Denominamos de *pointcuts* estáticos aqueles que podem ser casados na sua integridade com uma sombra de junção. Esse é o caso de todas as expressões *pointcut* do JBoss AOP, exceto as do tipo `cflow`[6]. É por isso que essas últimas foram denominadas de *pointcuts* dinâmicos.

Dessa forma, a primeira etapa da combinação do JBoss AOP é o casamento de *pointcuts* com sombras de junção. Esse casamento será realizado ignorando-se restrições do tipo `cflow`, cuja presença resultará numa checagem em tempo de execução para verificar quais métodos e construtores estão na pilha de execução.

Uma vez determinadas quais sombras de junção casam com *pointcuts*, essas sombras são alteradas de forma a serem interceptadas pelos respectivos adendos. Isso significa que o JBoss AOP instrumenta o código do sistema base, de modo a executar os adendos associados a cada sombra no momento correto (adendos *before* são inseridos antes da sombra; adendos *after* depois da sombra, e assim por diante). Todas as chamadas a adendos ficam encapsuladas dentro de **ganchos**. Um gancho é nada mais do que um método responsável por executar os adendos associados a uma sombra de junção específica na ordem apropriada. É importante notar que um gancho é responsável não só por executar os adendos, mas por executar a sombra de junção englobada numa pilha *around*.

Após serem gerados, os ganchos precisam ser inseridos no fluxo de controle do sistema base de alguma forma. Isso nos leva à segunda parte da combinação: a **substituição de sombras**. Cada sombra deve ser substituída por uma chamada ao gancho associado a ela. Uma vez finalizada a substituição, a instrumentação está concluída.

No exemplo dado há pouco, a instrumentação da chamada ao construtor de `StringBuffer` consistiria na geração de um gancho e na substituição dessa chamada a uma chamada ao gancho gerado. Para que essa substituição seja possível, o gancho gerado deve, naturalmente, devolver uma instância de `StringBuffer`. Ademais, esse gancho deve chamar todos os adendos que deverão interceptar aquela sombra, além de ser responsável por executar a sombra a ser interceptada, `new StringBuffer()`. Veja a listagem abaixo:

[6] Expressões do tipo `cflow` definem quais métodos e construtores devem estar na pilha de execução de um ponto de junção. Para mais detalhes sobre esse tipo de expressão, vide o Apêndice A

```
    ...
    StringBuffer sb = callStringBuffer$aop();
    sb.append("new message");
    ...
}

StringBuffer callStringBuffer$aop()
{
    StringBuffer result;
    try {
        // chamar adendos do tipo before
        ...
        // executar a sombra de junção
        result = new StringBuffer();
        // chamar adendos do tipo after
        ...
    } catch(Exception e) {
        // executar adendos do tipo throwing
        ...
        throw e;
    } finally {
        // executar adendos do tipo finally
        ...
    }
    return result;
}
```

Listagem 2.12: O trecho contido em a Listagem 2.11 instrumentado pelo JBoss AOP: Um gancho denominado `callStringBuffer$aop` foi gerado: Esse gancho faz chamadas aos adendos e à própria sombra de junção original: No local onde havia a chamada ao construtor de `StringBuffer`, existe agora uma chamada ao gancho gerado:

Essa listagem mostra de forma simplista o corpo do gancho gerado. Note que as chamadas aos adendos estão ocultas, assim como não inserimos nesse gancho a chamada à pilha *around*, que seria necessária caso houvessem adendos do tipo *around*. Os adendos invocados por um gancho compõe o que chamamos de **cadeia de adendos** .

No JBoss AOP, a etapa de combinação é conhecida como **transformação**. Classes que foram instrumentadas pelo JBoss AOP são denominadas de **classes transformadas**.

Finalmente, após a combinação, a interceptação dos pontos de junção ocorre durante a execução do sistema. A interceptação consiste na execução dos adendos nos pontos de junção conforme foi especificado por associações. Essa etapa ocorre como uma conseqüência da etapa anterior. Quando os

bytecodes instrumentados são executados, os adendos são executados nos pontos de junção definidos pelo usuário.

Contudo, essa não é a única implementação de combinação existente no JBoss AOP. Na integração com o JBoss AS, o servidor de aplicações do grupo JBoss, *proxies* dinâmicos são utilizados para fazer a combinação dos serviços do servidor, implementados na forma de interceptadores. Esses *proxies* possuem os ganchos que farão a chamada aos adendos, da mesma forma que a Listagem 2.12. No momento de executar o ponto de junção, eles invocam o objeto representado pelo *proxy*, cujos *bytecodes* não foram alterados. Essa implementação é utilizada apenas na integração do JBoss AOP com o servidor, e atualmente não está disponibilizada para aplicações *standalone*.

3. Combinação Dinâmica de Aspectos

No capítulo anterior introduzimos os principais conceitos de programação orientada a aspectos e mostramos como eles se traduzem em funcionalidades numa ferramenta de *software* livre, o JBoss AOP. Apesar de longa, essa introdução se faz necessária para o entendimento do que falaremos de agora em diante.

O tema desse capítulo é a combinação dinâmica de aspectos. Definimos a **combinação dinâmica** como sendo o mecanismo que possibilita a **programação orientada a aspectos dinâmica**. A POA é dita dinâmica quando aspectos são adicionados e removidos do sistema em tempo de execução. O potencial desse recurso se deve à união da programação orientada a aspectos à capacidade de alteração de um sistema em tempo de execução.

Na Seção 2.1 na página 4, descrevemos a necessidade de reutilização de código, fator que acreditamos ter sido o principal impulsionador da criação da programação orientada a aspectos. Outros fatores impulsionaram o desenvolvimento das Ciências da Computação nas mais diversas áreas ao longo dos últimos anos. Dentre eles, podemos destacar a necessidade de adaptabilidade de um sistema, que começou a ser estudada no início da década de 80.

A **reflexão** foi uma das primeiras abordagens apresentadas para a obtenção de sistemas adaptáveis. Podemos defini-la como sendo uma forma de se estruturar e organizar procedimentos auto-modificáveis. Com esse recurso, um sistema pode responder a inquisições sobre ele mesmo (inquisições reflexivas) e suportar ações nele mesmo (ações reflexivas). Na década seguinte, vários trabalhos de reflexão orientada a objetos foram realizados [34, 52, 77, 78], mostrando o grande potencial dessa técnica. Até hoje, ela é largamente aplicada em sistemas adaptáveis.

Apesar disso, a reflexão não provê meios de encapsular as novas funcionalidades a serem adicionadas dinamicamente a um sistema, capacidade que encontramos na programação orientada a aspectos. Por outro lado, na sua forma tradicional, a POA é considerada **estática**, uma vez que é necessário realizar a combinação dos aspectos aos componentes antes que se inicie a execução do sistema (esse procedimento é denominado **combinação estática**).

Felizmente, não demorou muito para que, a partir da criação da POA, começassem a surgir trabalhos e ferramentas de **POA dinâmica** [47, 57, 65]. Esta une o melhor das duas soluções, permitindo o encapsulamento de funcionalidades ortogonais em sistemas adaptáveis, de forma que aspectos sejam combinados a um sistema em tempo de execução. Além disso, tais aspectos não precisam ser previamente conhecidos pelo sistema, permitindo mudanças de requisitos não previstas em tempo de compilação. O processo de adicionar aspectos a um sistema base em tempo de execução, bem como o de removê-los, é o que denominamos **combinação dinâmica**.

No decorrer dos últimos anos, diversos autores têm comprovado o potencial da combinação dinâmica

através de estudos de caso [18, 19, 26, 28, 30, 32, 33, 59].

Esse capítulo se propõe a avaliar o desempenho da combinação dinâmica de aspectos em Java da forma mais genérica possível. Para isso, consideramos importante analisar como ela é implementada em algumas ferramentas. Na próxima seção, veremos como funciona a combinação dinâmica em algumas ferramentas de POA dinâmica em Java. Após isso, avaliaremos os impactos que a implementação desse tipo de combinação podem causar no desempenho de um sistema.

3.1. Estudos de Caso

Nessa seção, faremos um estudo de caso com as ferramentas JBoss AOP, AspectWerkz, JasCO e PROSE, onde conheceremos os principais mecanismos utilizados para a sua implementação.

Começaremos ilustrando como a combinação dinâmica pode ser realizada no JBoss AOP e mostrando os principais mecanismos envolvidos na sua implementação. Após, falaremos sobre AspectWerkz, JasCO e PROSE, onde o foco principal será a implementação da combinação dinâmica.

3.1.1. JBoss AOP

Antes desse trabalho ser concluído, o JBoss AOP realizava toda a instrumentação de forma estática, isto é, a instrumentação ocorria exclusivamente ou em tempo de compilação ou em tempo de carga (dependendo de como o JBoss AOP é utilizado). Na Seção 2.5.1 vimos que, nesse momento, ganchos são adicionados nas sombras de junção para executar uma cadeia de adendos naquele ponto. O resultado dessa instrumentação é, conforme esperado, a interceptação, que ocorre durante a execução dessas sombras, momento no qual esses ganchos são executados.

A implementação inicial da combinação dinâmica no JBoss AOP era simples: consistia na **atualização das cadeias de adendos**. À medida que adendos e interceptadores eram adicionados e removidos em tempo de execução, bastava atualizar as cadeias de adendos das sombras de junção afetadas. Uma vez que os ganchos já estavam inseridos nessas sombras, a atualização das cadeias se refletia de forma automática durante a execução das sombras interceptadas.

O JBoss AOP suporta a combinação dinâmica através de dois recursos, a saber: operações dinâmicas e *hot deployment* de aspectos. Por ser a ferramenta onde testaremos nossas idéias, implementando-as, julgamos conveniente mostrar em detalhes esses recursos e suas restrições.

3.1.1.1. API de Operações Dinâmicas

No modo *standalone*, a combinação dinâmica no JBoss AOP ocorre exclusivamente através das denominadas operações de combinação dinâmica ou, simplesmente, **operações dinâmicas**. Essas são disponibilizadas através de uma API, que iremos detalhar.

As principais operações dinâmicas no JBoss AOP são fornecidas por alguns métodos da sua fachada principal, a classe `org.jboss.aop.AspectManager`:

- `void addBinding(org.jboss.aop.advice.AdviceBinding)`,

- `void removeBinding(String)`,
- e `void removeBindings(ArrayList)`.

Esses métodos são utilizados internamente durante a configuração, para armazenar as associações declaradas no arquivo `jboss-aop.xml` e em anotações. Quando invocados durante a execução do sistema, eles servem como ponto de entrada para a execução das operações dinâmicas.

Os métodos `addBinding` e `removeBinding` executam a adição e a remoção de associações de *pointcuts*, respectivamente. Para efetuar uma adição, basta criar uma associação dos novos interceptadores e adendos a um *pointcut*, e adicionar essa associação através do método `addBinding`. O resultado dessa operação é a combinação dos novos adendos aos pontos de junção apropriados. É possível também utilizar o mesmo método para redefinir associações previamente registradas (mesmo as associações declaradas na configuração inicial do sistema). O uso de `removeBinding` requer o nome de uma associação presente no sistema, algo que pode ser obtido através do método `AdviceBinding.getName()`. Quanto ao método `removeBindings`, se trata de um atalho para a remoção de mais de uma associação. Seu argumento é uma lista com os nomes das associações a serem removidas.

Veja o exemplo abaixo:

```
1   // obtenção da instância singleton de AspectManager
2   AspectManager aspectManager = AspectManager.instance();
3
4   // criação de uma associação de adendos
5   AdviceBinding binding =
6                 new AdviceBinding("execution(POJO->new(..))", null);
7
8   // adição de um interceptador
9   binding.addInterceptor(SimpleInterceptor.class);
10
11  // adição de um adendo do tipo around
12  AspectFactory aspectFactory =
13                      new GenericAspectFactory(MyAspect.class, null);
14  AspectDefinition aspectDefinition = new AspectDefinition
15                        ("meu aspecto", Scope.PER_VM, aspectFactory);
16  aspectManager.addAspectDefinition(aspectDefinition);
17  AdviceFactory aroundAdviceFactory =
18                    new AdviceFactory(aspectFactory, "myAroundAdvice");
19  binding.addInterceptorFactory(aroundAdviceFactory);
20
21  // adição de um adendo do tipo after
22  AdviceFactory afterAdviceFactory =
23                      new AfterFactory(aspectFactory, "myAfterAdvice");
24  binding.addInterceptorFactory(afterAdviceFactory);
```

```
25
26      // a adição da associção ao aspect manager
27      // executa a combinação dinâmica
28      aspectManager.addBinding(binding);
```

Listagem 3.1: Exemplo de operação dinâmica.

Nesse exemplo, uma associação (`AdviceBinding`) é criada (linhas 5-6) e adicionada ao `AspectManager` (linha 28). O *pointcut* que ela contém é `"execution(POJO->new())"`, passado como primeiro argumento para o seu construtor. Note que o segundo argumento é nulo. Esse argumento deve receber o nome de um `cflow`. O fato de ele ser nulo indica que não há uma expressão `cflow` nessa associação. No intervalo que compreende as linhas de 9 a 24, são adicionados um interceptador e dois adendos a essa associação. O interceptador é o único que pode ser adicionado utilizando-se o método `addInterceptor(Class)`, mais simples na sua assinatura. Já os adendos podem ser adicionados apenas através do método `addInterceptorFactory(InterceptorFactory)`, o que requer a utilização da API descrita na Seção 4.1.2.

O equivalente dessa associação no formato XML é mostrada a seguir:

```
1     <bind pointcut="execution(POJO->new())">
2        <interceptor class="SimpleInterceptor"/>
3        <around name="myAroundAdvice" aspect="MyAspect"/>
4        <after name="myAfterAdvice" aspect="MyAspect"/>
5     </bind>
```

Listagem 3.2: Declaração de associação equivalente à associação que foi adicionada dinamicamente na Listagem 3.1.

Essas operações podem ser acessadas também através de domínios AOP. No JBoss AOP, existem **domínios**, que contêm subconjuntos das classes instrumentadas. Um domínio, representado pela classe `org.jboss.aop.Domain`, é uma subclasse de `AspectManager` e, portanto, herda todas as características da super classe, incluindo as operações dinâmicas. Operações realizadas nesses domínios afetarão somente as classes daquele domínio, mantendo o resto das classes instrumentadas inalteradas. É interessante notar que a instância única de `AspectManager` é conhecida como **domínio principal**, por incluir as classes de todos os domínios. Mostraremos mais detalhes sobre a estrutura de domínios no capítulo seguinte.

Além dessas operações, é possível aplicar um ou mais interceptadores exclusivamente a um único objeto. Todos os interceptadores associados a um objeto serão invocados sempre que qualquer método de tal objeto for executado e sempre que qualquer campo tiver o seu valor lido ou redefinido [1].

Esse recurso está disponível na interface de `InstanceAdvisor`. Existe uma instância dessa interface associada a cada instância de uma classe transformada. O `InstanceAdvisor` contém pilhas *around* a

[1] Existem restrições nos pontos de junção passíveis de operações dinâmicas. Veja a Seção 3.1.1.3 para mais detalhes.

serem executadas somente nos pontos de junção da instância à qual está associado. Através dele, é possível adicionar interceptadores a um objeto. Os métodos disponibilizados pelo `InstanceAdvisor` são:

- `void insertInterceptor(Interceptor interceptor)`,
- `void removeInterceptor(String name)`,
- `void appendInterceptor(Interceptor interceptor)`,
- `void insertInterceptorStack(String stackName)`,
- `void removeInterceptorStack(String stackName)`,
- e `void appendInterceptorStack(String stackName)`.

Observe que existem dois tipos de métodos para adição de interceptadores: os que começam com a palavra *insert* e os que começam com a palavra *append*. O motivo disso é que o `InstanceAdvisor` possui, internamente, duas pilhas de interceptadores. Uma é a **pilha inserida**, e será adicionada ao topo da pilha `around` de cada ponto de junção daquela instância, de forma ser a primeira parte da pilha a ser executada. A outra é a **pilha anexada**, que será adicionada à base da pilha `around` de cada ponto de junção, de forma a conter os últimos interceptadores que serão invocados. Os métodos cujos nomes começam com *insert* adicionam um interceptador à pilha inserida. Os métodos do tipo *append*, à pilha anexada.

O acesso ao `InstanceAdvisor` de uma instância se dá através da interface `org.jboss.aop.Advised`. Essa interface é implementada por toda classe que tiver sido transformada pelo JBoss AOP:

```
1  public interface Advised extends InstanceAdvised
2  {
3     // retorna o class advisor
4     public Advisor _getAdvisor();
5  }
6
7  public interface InstanceAdvised
8  {
9     // retorna o instance advisor
10    public InstanceAdvisor _getInstanceAdvisor();
11
12    // define o instance advisor
13    // esse método deve ser chamado apenas internamente
14    public void _setInstanceAdvisor(InstanceAdvisor advisor);
15 }
```

Listagem 3.3: A interface `org.jboss.aop.Advised`. Essa interface associa um objeto transformado a um `Advisor` e a um `InstanceAdvisor`.

Reveremos essa interface com mais detalhes no próximo capítulo. O exemplo a seguir utiliza a interface `Advised` para obter o `InstanceAdvisor` de um objeto e adicionar um interceptador:

```
1   // criação de duas instâncias de POJO
2   POJO pojo1 = new POJO();
3   POJO pojo2 = new POJO();
4
5   // obtenção do instance advisor associado a pojo1
6   InstanceAdvisor advisor = ((Advised) pojo1).getInstanceAdvisor();
7
8   // inserção de um interceptador em pojo1
9   advisor.insertInterceptor(new MyInterceptor());
10
11  // execução do método someMethod em pojo1:
12  // será automaticamente interceptada por MyInterceptor
13  pojo1.someMethod();
14
15  // execução do método someMethod em pojo2:
16  // não será interceptada pelo interceptador inserido
17  pojo2.someMethod();
```

Listagem 3.4: Exemplo de inserção de interceptadores através do `InstanceAdvisor`.

Esse exemplo cria duas instâncias de `POJO`, mas insere um interceptador apenas no `InstanceAdvisor` da instância `pojo1` (linha 9). Como resultado, quando o método `someMethod` de `pojo1` é executado, o interceptador inserido também será executado. No entanto, a operação realizada na linha 9 não alterou `pojo2`, que não é interceptado quando o seu método `someMethod` executa.

3.1.1.2. Hot deployment de Aspectos no JBoss AS

Até agora não entramos em detalhes na utilização do JBoss AOP dentro do servidor JBoss AS. O JBoss AS suporta unidades de implantação com a extensão `.aop`. Esse tipo de unidade deve conter um ou mais aspectos e, opcionalmente, um arquivo `jboss-aop.xml` (se esse arquivo estiver ausente, os aspectos têm que estar anotados). Quando o arquivo é implantado no servidor, o JBoss AOP realiza uma combinação dinâmica dos seus aspectos com todos os componentes de usuário que já tiverem sido implantados no servidor.

O uso desses arquivos não é a única opção de implantação de aspectos no JBoss AS. Outra alternativa é implantar apenas um arquivo `jboss-aop.xml` no servidor. Nesse caso, os aspectos que o arquivo declara devem estar disponíveis em algum outro módulo que já tenha sido implantado no servidor. Assim, quando um arquivo `jboss-aop.xml` é implantado no servidor, o JBoss AOP também realiza uma combinação dinâmica dos elementos que ele declara com as outras unidades implantadas.

Em ambos os casos, novas aplicações que forem implantadas no servidor após uma operação de combinação dinâmica serão automaticamente combinadas com todos os aspectos implantados no servidor, inclusive os que tiverem sido adicionados dinamicamente.

É possível também implantar aspectos dentro de outras unidades de implantação, como as do tipo `war` e `ear`. Nesse caso, porém, o *hot deployment* dos aspectos é feito juntamente com as classes que eles alteram, o que resulta na execução de uma combinação estática e não dinâmica. Uma exceção ocorre quando a unidade que engloba os aspectos não está dentro de um escopo. Nesse cenário, os aspectos que ela contém afetarão também as outras unidades que já estiverem implantadas no servidor, da mesma forma que ocorreria com uma unidade `aop`, o que classificamos como sendo uma combinação dinâmica.

Para implementar o *hot deployment*, o JBoss AOP utiliza internamente as operações dinâmicas do `AspectManager` vistas na seção anterior.

3.1.1.3. Sombras de Junção Preparadas

Vimos que é através das operações das classes `InstanceAdvisor` e `AspectManager` que a combinação dinâmica é executada no JBoss AOP.

Porém, essas operações apresentam restrições, já que não é possível adicionar aspectos em todos os pontos de junção do sistema.

Os aspectos podem ser adicionados somente nos pontos onde existem ganchos para a sua execução. Vimos, na Seção 2.5.1, que os ganchos podem ser adicionados ou em tempo de compilação ou em tempo de carga. Portanto, somente nas sombras de junção que casarem com um ou mais *pointcuts* configurados inicialmente no sistema será possível adicionar aspectos. Esses ganchos apontam para a cadeia de adendos que contém todos os adendos e interceptadores a serem executados naquele ponto do código, inclusive os que formarão a pilha *around*.

Isso significa que, no JBoss AOP, a combinação dinâmica está restrita às sombras onde foram inseridos os ganchos, chamadas de **sombras de junção preparadas** ou, simplesmente, sombras preparadas.

Existe uma forma de forçar a preparação de sombras que, inicialmente, não serão interceptadas por nenhum adendo. O JBoss AOP disponibiliza, para isso, as **expressões de preparação**. A sintaxe dessas expressões é a mesma dos *pointcuts*, porém elas são declaradas com o exclusivo intuito de forçar a preparação de todas as sombras que casarem com essas expressões. Internamente, as expressões de preparação são armazenadas como *pointcuts* comuns no `AspectManager`. O resultado disso é que, de forma transparente, os transformadores instrumentarão as sombras identificadas por essas expressões, da mesma forma que eles o fariam com as identificadas por *pointcuts* comuns, associados a adendos e interceptadores. A diferença é que o gancho inserido pelos transformadores apontará para uma cadeia de adendos vazia.

As expressões de preparação são declaradas no arquivo `jboss-aop.xml`. Veja o exemplo a seguir:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <aop>
3     ...
4     <prepare expr="all(*POJO)"/>
```

```
5       ...
6  </aop>
```

Listagem 3.5: Declaração de uma expressão de preparação.

No exemplo, serão preparados para a combinação dinâmica todos os métodos e construtores, leituras e escritas de campos de todas as classes cujo nome termina com `"POJO"`.

Outra forma de preparar sombras de junção não associadas a adendos é o uso da anotação `@Prepare`. As classes que possuírem essa anotação terão as suas sombras de junção (exceto sombras de chamadas) preparadas para a combinação dinâmica.

O motivo da necessidade de preparação de pontos de junção será detalhado na Seção 3.2.

3.1.2. AspectWerkz

O AspectWerkz é uma ferramenta dinâmica que fez muito sucesso no começo dessa década. Porém ela está descontinuada, já que foi adquirida pelo AspectJ (atualmente, o AspectJ está em processo de integração com as funcionalidades dinâmicas do AspectWerkz).

No AspectWerkz, aspectos e adendos são classes e métodos comuns, assim como no JBoss AOP. Eles podem ser configurados através de anotações ou de um arquivo XML. Como já vimos anteriormente, essa ferramenta utiliza instrumentação. A manipulação de *bytecodes* no AspectWerkz é realizada através da ferramenta ASM [5], própria para esse tipo de manipulação.

As semelhanças não param por aí. A combinação dinâmica no AspectWerkz é muito similar à do JBoss AOP.

No AspectWerkz também existe uma classe que disponibiliza operações dinâmicas, a classe `org.codehaus.aspectwerkz.transform.inlining.deployer.Deployer`. Os métodos que ela disponibiliza são os seguintes:

- `DeploymentHandle deploy(Class)`,
- `DeploymentHandle deploy(Class, ClassLoader)`,
- `DeploymentHandle deploy(Class, DeploymentScope, ClassLoader)`,
- `DeploymentHandle deploy(Class, String)`,
- `DeploymentHandle deploy(Class, String, ClassLoader)`,
- `DeploymentHandle deploy(Class, String, DeploymentScope)`,
- `DeploymentHandle deploy(Class, String, DeploymentScope, ClassLoader)`,
- `void undeploy(Class)`,
- `void undeploy(Class, ClassLoader)`,
- e `void undeploy(DeploymentHandle)`

Com os métodos `deploy`, é possível adicionar aspectos através de classes. É possível especificar: configurações no formato XML (o argumento do tipo `String` nos métodos acima deve conter uma especificação do aspecto em *tags* XML); o `ClassLoader` cujas classes carregadas deverão ser afetadas pela operação; e até uma restrição de escopo `DeploymentScope`.

Um objeto `DeploymentScope` é um escopo formado por pontos de junção que foram preparados para a combinação dinâmica. Assim como no JBoss AOP, não são todos os pontos que podem ser alvo de operações dinâmicas. No AspectWerkz, o *deployment scope* é uma expressão *pointcut* não associada a adendos, equivalente às expressões de preparação do JBoss AOP. Eles podem ser declarados no arquivo XML ou através de anotações. Para realizar uma operação dinâmica com o uso de um objeto `DeploymentScope`, é preciso obter um dos *deployment scopes* que foram previamente configurados. Essa obtenção é feita a partir da classe `org.codehaus.aspectwerkz.definition.SystemDefinition` que, de forma equivalente ao `AspectManager` do JBoss AOP, armazena todas as informações configuradas no arquivo XML e em anotações.

É possível também remover aspectos através dos métodos `undeploy`. Observe o último método `undeploy` da listagem de operações dinâmicas acima. Diferente dos demais, cuja assinatura especifica um aspecto a ser removido, esse método desfaz uma operação `deploy` que foi executada anteriormente, utilizando-se do `DeploymentHandle` que ela devolveu.

A grande diferença do AspectWerkz em relação ao JBoss AOP está na preparação das sombras casadas por expressões `DeploymentScope`. No JBoss AOP, a preparação consiste na adição de ganchos para a execução de cadeias de adendos vazias. No AspectWerkz, ela consiste na adição de ganchos (no caso, métodos) que não serão executados, mantendo o controle de fluxo dos pontos correspondentes inalterado.

Quando uma operação `deploy` afeta sombras que foram preparadas, o código delas é instrumentado em tempo de execução, para alterar o seu fluxo de controle de forma a executar os ganchos previamente adicionados. Esses ganchos serão responsáveis por invocar os adendos que foram implantados através da operação `deploy`. A instrumentação em tempo de execução resulta em novos *bytecodes* para aquela classe. O AspectWerkz força uma recarga dessa classe pelo `ClassLoader`. Para que essa recarga seja possível, o AspectWerkz se utiliza, basicamente, de três opções:

JVMDI: a *Java Virtual Machine Debugger Interface* [44] fornece infra-estrutura de depuração a aplicações Java. O AspectWerkz utiliza essa ferramenta para poder redefinir classes através de um gancho inserido no método `defineClass` de `ClassLoader`. Essa API requer o uso de código nativo, e só pode ser utilizada quando o sistema for executado em modo de depuração.

bootclasspath: é possível executar a máquina virtual redefinindo-se o `bootclasspath` com o parâmetro `-Xbootclasspath`. O AspectWerkz provê uma versão instrumentada de `ClassLoader`, que também possui um gancho para redefinir classes. É possível executar o sistema orientado a aspectos com essa versão do `ClassLoader`, garantindo o suporte à redefinição de classes em tempo de execução.

JVMTI: a *Java Virtual Machine Tools Interface* [45] substituiu a JVMDI a partir do Java 1.5. Essa API permite a redefinição de classes sem a necessidade de executar o sistema em modo de

depuração ou de utilizar código nativo.

Além dessas opções, existem outras que são pequenas variações das que descrevemos. A redefinição de `bytecodes` em tempo de execução é denominada ***hot swap***. Com isso, podemos resumir a implementação da combinação dinâmica no AspectWerkz nos seguintes passos:

1. durante a inicialização, as sombras que casam com expressões `DeploymentHandle` são preparadas "*a la* AspectWerkz", sem alterar o fluxo dos mesmos; por outro lado, as sombras que casam com *pointcuts* comuns são instrumentadas totalmente, com a adição de ganchos para executar os adendos associados;

2. quando uma operação dinâmica é realizada, o AspectWerkz itera por todos as sombras de junção alvo da operação, verificando quais delas casam com os *pointcuts* dos adendos do aspecto que está sendo adicionado (`deploy`) ou removido (`undeploy`);

   a) se o aspecto estiver sendo adicionado: sombras preparadas serão instrumentadas para a execução do adendo; sombras totalmente instrumentadas terão apenas o adendo inserido na sua **cadeia de adendos**;

   b) se o aspecto estiver sendo removido, o reverso ocorre. As sombras instrumentadas que estiverem associadas ao adendo em questão terão o mesmo removido de sua cadeia de adendos. Após isso: sombras instrumentadas que deixaram de estar associadas a um adendo são instrumentadas novamente, dessa vez para retornarem ao seu estado inicial, permanecendo apenas preparadas para operações futuras; sombras de junção instrumentadas que continuam associadas a um ou mais adendos não serão instrumentadas;

3. todas as classes que foram instrumentadas em tempo de execução são recarregadas pelo seu `ClassLoader` (***hot swap***).

3.1.3. JasCO

O JasCO é outra ferramenta que utiliza a instrumentação de *bytecodes* para implementar a combinação de aspectos. Assim como no JBoss AOP, a instrumentação é realizada através do Javassist [37].

Similar ao AspectJ, o JasCO estende a linguagem Java com novas construções para a implementação de aspectos e adendos. Aspectos são declarados como classes Java que contêm uma série de ganchos (`hooks`) [2]. Essas classes são denominadas *Aspect Beans*. Cada `hook` é declarado da mesma forma que declaramos uma classe interna, só que com a palavra chave `hook` ao invés de `class`. O `hook` é a implementação de um adendo, e pode utilizar uma série de construções do JasCO na sua implementação. De forma sucinta, podemos dizer que a sintaxe de um `hook` se resume a: possuir um construtor que recebe um *pointcut*; e conter uma espécie de método ou rotina com a implementação do adendo. Esse

[2] O termo `hook` é uma palavra-chave no JasCO, e nada tem a ver com os ganchos adicionados nos *bytecodes* para a invocação de adendos que vimos até agora. Para evitar confusão desses termos, utilizaremos o termo `hook`, em inglês, quando estivermos falando da construção presente no JasCO, e usaremos o termo gancho quando estivermos nos referindo a métodos inseridos no código para a execução de adendos.

método não possui tipo de retorno em sua declaração e pode ter um dos três nomes `before`, `replace` ou `after`, dependendo do tipo do adendo que o gancho representa (*before*, *around* ou *after*).

No JasCO, a adição e remoção de aspectos em tempo de execução não depende de uma API, e sim de construções da própria linguagem para fazê-lo. Os conectores são componentes que realizam essas operações. Veja o exemplo abaixo:

```
1 static connector MyConnector
2 {
3     MyAspect.MyHook myHook =
4                         new MyAspect.MyHook (void POJO.doSomething(String));
5     myHook.before();
6 }
```

Listagem 3.6: Adição de um `hook` (adendo) no JasCO.

Esse exemplo implanta o gancho ou adendo `myHook` do aspecto `MyAspect`. Para isso, esse gancho é instanciado nas linhas 3-4, associado a um *pointcut* que casa com a execução do método `doSomething(String)` da classe `POJO`. Na linha 5, existe uma chamada à rotina `before()` desse gancho, indicando que ele representa um adendo do tipo *before*.

É possível também adicionar e remover aspectos a um objeto, de forma similar às operações da classe `InstanceAdvisor` no JBoss AOP.

Outra forma de realizar combinação dinâmica no JasCO é através do *hot deployment* de aspectos. Ele monitora o *classpath* detectando quando classes de aspectos são adicionadas ou removidas do *classpath*. Quando uma dessas ações é detectada, tem início a combinação dinâmica. Esse mecanismo pode ser desativado a fim de obter um melhor desempenho durante a execução do sistema.

A implementação da combinação dinâmica do JasCO é bem similar à do AspectWerkz. O fluxo de controle é mantido inalterado nas sombras preparadas. Quando um aspecto precisa ser adicionado a uma delas, ele utiliza *hot swap* para trocar os *bytecodes*. Diferente das outras ferramentas que vimos, o JasCO prepara todas as sombras, já que os autores consideram que um sistema orientado a aspectos não é realmente dinâmico senão tiver todas as suas sombras preparadas [75]. Outra diferença é que as sombras que foram instrumentadas durante a inicialização não são passíveis de *hot swap*, pois o JasCO assume que os adendos declarados na inicialização são adendos que se deseja manter implantados durante toda a execução do sistema.

3.1.4. PROSE

Da mesma forma que o JBoss AOP e o AspectWerkz, o PROSE disponibiliza uma API para realizar operações dinâmicas. O trecho a seguir utiliza essa API para adicionar um aspecto em tempo de execução:

```
1     MyAspect myAspect = new MyAspect();
```

```
2    Prose.extensionManager().insert(myAspect);
```

Listagem 3.7: Adição de um aspecto em tempo de execução no PROSE.

A adição de aspectos via execução de operações dinâmicas não é a única forma de adicionar um aspecto em tempo de execução. O meio mais interessante de utilizar combinação dinâmica no PROSE é através da JVMAI, ou *Java Virtual Machine Aspect Interface*, que ele disponibiliza. Essa interface permite a inserção e a remoção de aspectos de forma remota. Com ela, é possível se conectar no sistema orientado a aspectos e enviar um aspecto seriado para esse sistema. Assim como ocorre com o *hot deployment* de aspectos, os aspectos adicionados através da JVMAI não precisam ser conhecidos previamente. Para utilizar essa interface, basta executar o comando `clprose`, passando os argumentos apropriados.

O PROSE é um projeto voltado para pesquisas de programação orientada a aspectos dinâmica. Por esse motivo, ele atualmente já foi implementado utilizando-se três abordagens diferentes.

Na Seção 2.5 na página 20, citamos a primeira das três abordagens ao mencionar que o PROSE se utiliza de *breakpoints* em sua implementação. Nessa abordagem, a implementação da combinação dinâmica no PROSE é extremamente simples. À medida que aspectos são adicionados e removidos do sistema, o PROSE adiciona e remove *breakpoints* nas sombras afetadas. Quando esses *breakpoints* são atingidos durante a execução, o PROSE entra em ação, casando o ponto de junção com *pointcuts* e chamando os adendos apropriados. Essa implementação do PROSE [65, 66] apresenta alguns pontos positivos, como a contribuição que esse trabalho representa para o estudo da POA dinâmica com o uso de *breakpoints*. Outro ponto positivo é que, com essa abordagem, o PROSE é a única ferramenta que mantém o código do sistema base inalterado. Porém, como afirmam os próprios autores, é uma solução consideravelmente lenta.

A segunda abordagem [67] consiste em utilizar o compilador JIT (*Just in Time*) Jikes da IBM [41]. Nesse caso, o PROSE faz uma instrumentação similar à realizada pelo JBoss AOP, preparando todas as sombras de junção do sistema com a adição de ganchos vazios. Esses ganchos, denominados **ganchos minimais**, substituíram a funcionalidade obtida pelo uso de *breakpoints* da primeira abordagem. Como o Jikes transforma todo os *bytecodes* a serem executados em código nativo, essa preparação e inserção de ganchos é feita diretamente no código nativo, por motivos de eficiência (observe que o Jikes suporta também transformação de *bytecodes*). Em tempo de execução, quando aspectos são adicionados e removidos do sistema, o PROSE utiliza JIT para alterar o conteúdo desses ganchos, ativando os ganchos associados a sombras que devem ser interceptadas, e inativando aqueles associados a sombras que não deverão ser interceptadas. Os ganchos ativos funcionarão como os *breakpoints* da versão anterior, pois possuem uma chamada ao sistema interno do PROSE. No momento da execução de uma sombra com um gancho ativo, essa chamada será realizada, desencadeando uma busca pelos adendos que deverão interceptar o ponto de junção. Os ganchos inativos são ganchos no seu estado inicial, vazio, sem conter chamadas ao sistema interno do PROSE. Uma vantagem dessa abordagem é a instrumentação do código nativo gerado pelo Jikes, o que permite otimizações. Uma desvantagem é a avaliação de cada ponto de junção para casá-lo com *pointcuts* no momento de sua execução, que também ocorre nos ganchos ativos da mesma forma que é feita na implementação com *breakpoints* Além disso, apesar de não executar o

sistema em modo de depuração, essa abordagem executa o mesmo com ganchos inseridos em todas as sombras de junção, o que naturalmente causa um impacto no desempenho do sistema durante toda a sua execução.

Finalmente, a terceira abordagem [61] é realizar instrumentação de *bytecodes*, através do BCEL (*Byte Code Engineering Library*) [10]. Essa abordagem tem vários pontos em comum com a implementação das outras ferramentas que vimos, especialmente com o AspectWerkz. Os *bytecodes* são preparados de forma a introduzir ganchos para a interceptação de sombras, sem alterar o fluxo de controle original do sistema. Durante a adição dinâmica de um aspecto, dois passos ocorrem: chamadas a adendos são inseridas nos ganchos afetados pela operação; chamadas aos ganchos são inseridas nas sombras correspondentes. Esse novo código entra em ação através do *hot swap*. Para a remoção desses aspectos, o reverso é feito.

Dentro dessa abordagem, existe uma opção para a interceptação de métodos que não exige o uso de ganchos [62]. Nessa opção, o código é inserido diretamente no corpo do método a ser interceptado. Se uma nova operação dinâmica de adição ou remoção de aspectos ocorrer, uma versão original do código, armazenada na memória, é utilizada para gerar o corpo do método novamente.

Há uma peculiaridade nessa terceira versão do PROSE. Na etapa de preparação das leituras e escritas de campos, um levantamento de todas sombras do código que realizam tais operações é realizado. Em tempo de execução, isso é utilizado para encontrar, de forma rápida, onde inserir chamadas para ganchos quando um ponto de junção de leitura ou escrita de campos precisar ser interceptado.

No PROSE, o *hot swap* é realizado ou através de API específica do Jikes, ou através da JVMDI. Essa última opção impõe o custo de executar a aplicação em modo de depuração, visto na Seção 3.1.2.

A principal vantagem da terceira abordagem utilizada na implementação do PROSE é similar à do AspectWerkz: não há impacto no fluxo de controle na ausência de adendos, já que os ganchos ficam inativos. Além disso, essa é a única das três abordagens onde o casamento dos *pointcuts* é feito com as sombras de junção apenas. A única exceção é o casamento de `cflow`, que, como demonstraremos na Seção 3.2.2.1, requer verificações em tempo de execução.

3.2. Desempenho

Consideramos que os impactos da combinação dinâmica no desempenho do sistema são de grande importância. Ao contrário da combinação estática, a combinação dinâmica se dá durante a execução do sistema e o tempo que ela leva afeta diretamente o desempenho do sistema no qual ela ocorre. O sistema alterado, resultante da combinação dinâmica, também poderá ter seu desempenho afetado pela mesma. Ademais, vimos nos estudos de caso que são necessários artifícios para realizar a combinação dinâmica em Java, muitos dos quais afetam o desempenho do sistema durante toda a sua execução.

Nessa seção, analisaremos quais são os impactos no desempenho causados pela combinação dinâmica antes, durante e depois de sua execução. Trataremos a implementação da POA dinâmica e, em especial, a combinação dinâmica, de forma geral, independente de detalhes específicos relacionados a ferramentas. Como base para as generalizações que aqui faremos vamos utilizar os casos de estudo da seção anterior,

onde avaliamos as principais ferramentas de POA dinâmica que encontramos no mercado. Deixaremos claro quando esse não for o caso e estivermos falando de uma abordagem específica.

Iniciaremos essa seção com a descrição dos passos envolvidos na POA dinâmica, citando os principais impactos de desempenho que ela causa. Logo após, voltaremos nossa atenção para a etapa mais importante da execução de sistemas de POA dinâmica: a combinação dinâmica.

3.2.1. Programação Orientada a Aspectos Dinâmica

A programação orientada a aspectos dinâmica consiste na adição e remoção de aspectos em tempo de execução.

No geral, podemos dividir a implementação da POA dinâmica em quatro etapas:

1. preparação do sistema para a POA dinâmica;
2. reconfiguração dos aspectos contidos no sistema;
3. combinação dinâmica;
4. e interceptação correta nos pontos de junção afetados.

Note a similaridade desses passos com as etapas que compõem a implementação da POA estática no JBoss AOP (veja a Seção 2.5.1 na página 21).

A primeira etapa, **preparação**, deve ocorrer na inicialização do sistema, deixando o sistema num estado que possibilite a adição e a remoção de aspectos durante a sua execução. Nas páginas anteriores, vimos diversas formas de fazer isso, e a forma escolhida depende principalmente da abordagem utilizada para a implementação da combinação dinâmica. No JBoss AOP, ganchos são adicionados para a execução de cadeias de adendos vazias nas sombras preparadas. A vantagem é que o sistema preparado pode ser executado em qualquer máquina virtual sem a necessidade de configurações extras mas, durante a sua execução, existe o custo adicional da execução desses ganchos, que nada mais são do que chamadas a métodos que executam cadeias de adendos vazias. O mesmo ocorre na implementação do PROSE com ganchos minimais. No AspectWerkz, JasCO e outras versões do PROSE, o fluxo de controle das sombras preparadas permanece inalterada. Porém, com exceção do AspectWerkz, é preciso executar o sistema em modo de depuração, resultando numa sobrecarga de aproximadamente 40% no desempenho [75]. Em todos esses cenários, está claro que a preparação do sistema tem que ser feita conforme a abordagem escolhida para a implementação da POA dinâmica. Quando a escolha é a adição de *breakpoints* (PROSE), a preparação consiste em apenas executar o sistema em modo de depuração. Quando a escolha é a instrumentação, é preciso preparar o código adicionando campos e métodos auxiliares para a execução de aspectos (ganchos). Chamadas a esses métodos terão de ser inseridas nas sombras de junção durante a sua preparação caso a combinação dinâmica não inclua instrumentação em tempo de execução com *hot swap*, que é o que ocorre com o JBoss AOP.

A segunda etapa, **reconfiguração** dos aspectos no sistema, ocorre quando uma operação dinâmica é efetuada no sistema. Essa etapa é formada por dois passos. O primeiro é a leitura dos *pointcuts* e adendos adicionados, um passo que ocorre somente quando a operação dinâmica é uma operação de adição. O segundo é a adição ou remoção de dados configurados do sistema.

No primeiro passo da reconfiguração, a forma como a leitura desses dados é realizada, bem como o seu custo, dependem de cada ferramenta, inclusive da interface que provê as operações dinâmicas. No JBoss AOP, por exemplo, as operações dinâmicas (descritas na Seção 3.1.1) recebem as informações já no formato de representação interna. O objeto `AdviceBinding` usado na adição de uma associação, por exemplo, é o mesmo que será armazenado no `AspectManager`, dispensando qualquer processamento adicional durante a sua adição. Porém, se levarmos em consideração o tempo levado para criar esse objeto, veremos que existe o custo de transformar o *pointcut* em uma árvore AST, que ocorre no construtor dessa classe. Por outro lado, o uso do *hot deployment* requer a leitura de arquivos XML e de anotações. Como é de conhecimento geral, operações de entrada e saída de dados costumam ser custosas, o que indica que operações dinâmicas são preferíveis ao *hot deployment* em termos de desempenho. Note que essa afirmação é válida para todas as ferramentas de POA dinâmica que oferecem as duas possibilidades, já que a implementação do *hot deployment* consiste na leitura de arquivos seguida pela execução de operações dinâmicas.

O segundo passo da reconfiguração consiste no armazenamento das informações lidas em coleções, no caso de adição de aspectos, e na remoção de informações contidas em coleções, no caso de remoção de aspectos. Note que o armazenamento dessas informações não pode ser evitado, pois, caso contrário, esses dados seriam perdidos após o seu uso na combinação, impossibilitando os seguintes recursos:

- remoção de aspectos, pois o controle dos aspectos adicionados e das sombras de junção afetadas por eles seria perdido;
- instrumentação de aspectos em tempo de carga, pois é preciso conhecer os aspectos para poder instrumentar as classes à medida que elas são carregadas.

Observe que o último item só é válido para ferramentas que utilizam a instrumentação de `bytecodes` para fazer a combinação.

O impacto decorrente do armazenamento das informações configuradas, bem como da remoção das mesmas, geralmente não é significativo. Na maioria dos casos, o número de aspectos não ultrapassa as dezenas e esse armazenamento é feito em coleções providas por bibliotecas, que oferecem implementações otimizadas para conjuntos, mapas, listas, etc.

A etapa seguinte à reconfiguração, a **combinação dinâmica**, é a mais complexa das quatro etapas que compõem a implementação da POA dinâmica. Ela consiste de duas etapas menores: o casamento das sombras de junção com os *pointcuts* adicionados/removidos pela operação dinâmica; e a alteração dessas sombras de modo que os adendos associados aos *pointcuts* sejam adicionados ao seu fluxo de controle, ou removidos do mesmo. Podemos definir o custo da combinação dinâmica como sendo a soma dos custos de cada uma dessas etapas. Veremos detalhes sobre elas e avaliaremos o seu desempenho na próxima seção.

Finalmente, após a combinação dinâmica, a alteração realizada por ela entrará em vigor. Tem início a quarta etapa: **interceptação correta** dos pontos de junção afetados. A forma como isso é feito e, conseqüentemente, o custo dessa etapa, também são uma conseqüência natural da abordagem utilizada para a alteração do sistema durante a execução da combinação dinâmica.

Se o sistema é alterado através de instrumentação, o código gerado será responsável pelo desempenho dessa etapa. Nesse caso, a invocação de adendos adicionados através de chamadas comuns Java é uma solução mais eficiente do que chamá-los através de reflexão. Para pontos de junção que deixaram de ser interceptados como conseqüência de uma remoção de aspecto, o desempenho dependerá de como o ponto de junção é executado quando não está associado a nenhum adendo. No caso da abordagem do JBoss AOP, existirá um gancho que executa uma cadeia de adendos vazia, causando certamente um impacto maior do que a abordagem utilizada pelo AspectWerkz, que fará com que o ponto de junção execute como se ele nunca tivesse sido interceptado.

Ademais, nas implementações do PROSE com *breakpoints* e com ganchos minimais, os pontos de junção são casados com *pointcuts* somente no momento de sua execução, quando o PROSE definirá quais adendos devem ser invocados. Por si só, essa abordagem é menos eficiente. Porém, além disso, ambas as implementações apresentam um custo adicional. No caso do uso de *breakpoints*, existe a necessidade de executar a aplicação no modo de depuração, também presente na execução do AspectWerkz quando utilizado o *hot swap* com JVMDI. No caso dos ganchos minimais, existe o custo de executar os ganchos em todos os pontos preparados, assim como ocorre com o JBoss AOP.

Podemos concluir, dessa breve análise da programação dinâmica em Java, que: a reconfiguração não apresenta muito espaço para estudos de otimização; e a preparação e a interceptação são etapas que dependem diretamente de como é implementada a combinação dinâmica. Por conseguinte, a abordagem escolhida para implementar **a combinação dinâmica se torna a principal responsável pelo desempenho de um sistema orientado a aspectos dinâmico**. Essa afirmação justifica a escolha da combinação dinâmica como tema do nosso trabalho.

3.2.2. Combinação de Aspectos

Uma vez ilustrados os principais impactos das etapas da POA dinâmica, vamos examinar mais detalhadamente o foco do nosso trabalho: a combinação dinâmica de aspectos.

A combinação dinâmica de aspectos tem início com o casamento das sombras de junção com os *pointcuts* envolvidos numa operação dinâmica, e segue com a alteração do fluxo de controle do sistema, inserindo ou removendo nesse fluxo chamadas aos adendos de acordo com as expressões *pointcut*. Nas próximas seções, veremos cada uma dessas etapas em detalhes, avaliando as possibilidades para a sua implementação, e o impacto que cada uma delas causaria num sistema orientado a aspectos dinâmico.

3.2.2.1. Casamento de Sombras de Junção

Vimos na seção a Seção 3.2.1 que o casamento de sombras de junção é o primeiro passo executado na combinação dinâmica. Atualmente, todas as ferramentas realizam essa etapa iterando pelas sombras preparadas do sistema, verificando, para cada uma delas, quais casam com o(s) *pointcuts(s)* envolvidos na operação dinâmica a ser executada.

Existem pequenas variações no modo como o casamento de sombras de junção é realizado. O JBoss AOP, por exemplo, reconstrói todas as cadeias de adendos das classes que contêm sombras afetadas. Primeiramente, ele verifica quais são as classes que possuem uma ou mais sombras que podem ter sido

afetadas pela operação dinâmica[3]. Todas as classes afetadas terão suas cadeias de adendos reconstruídas. Note que estamos nos referindo a todas as cadeias de adendos associadas a uma classe, incluindo execuções de métodos e leituras para campos, entre outras. Isso significa que cadeias que não estão associadas a sombras afetadas serão também reconstruídas. Durante esse processo de reconstrução, o JBoss AOP itera por todos os *pointcuts* registrados no sistema, verificando, para cada um deles, se ele casa positivamente com a sombra em questão (isso inclui os *pointcuts* adicionados pela operação dinâmica). Caso positivo, os adendos associados ao *pointcuts* são inseridos na cadeia de adendos da sombra. O JAsCO também itera pelos *joinpoints* do sistema, reconstruindo a cadeia de adendos associada a cada um deles. Esse algoritmo é mais lento do que o do AspectWerkz, que apenas adiciona ou remove os adendos às cadeias das sombras de junção afetadas, sem precisar reconstruí-las do zero. Para encontrar as sombras de junção afetadas, ele itera pelas sombras preparadas (ou as sombras de um `DeploymentModel`, caso esteja se utilizando um dos métodos dinâmicos da classe `Deployment` que recebem um `DeploymentModel`), verificando uma a uma, quais delas casam com o(s) *pointcut(s)* envolvidos.

Já o PROSE apresenta três implementações de combinação dinâmica diferentes. A implementação com *breakpoints* demanda a adição de *breakpoints* somente nas sombras que casarem com os adendos adicionados. Já a implementação com os ganchos minimais, requer o *hot swap* do conteúdo dos ganchos minimais associados às sombras que casam positivamente com os aspectos adicionados. Finalmente, na instrumentação, é preciso casar as sombras de junção para definir onde serão inseridas chamadas aos adendos adicionados. De qualquer modo, as três implementações do PROSE executam o casamento de sombras de junção após uma operação dinâmica.

A etapa de casamento de sombras pode ser evitada na remoção de aspectos. É o que a maioria das ferramentas faz. Para isso, é preciso controlar em quais sombras cada adendo foi inserido. Assim, quando um adendo precisa ser removido, é fácil descobrir de onde ele deve ser removido sem realizar o casamento das sombras com o *pointcut* a ele associado.

Após essa breve análise das diversas abordagens de implementação para o casamento de sombras de junção, podemos supor que o **casamento de sombras de junção**, ou *pointcut matching*, é um processo necessário em um sistema de POA dinâmica completo, que deve ser realizado durante operações dinâmicas que adicionam um ou mais adendos associados a um *pointcut*.

Podemos ainda, notar duas invariantes em todas as ferramentas que vimos: *1)* esse casamento é feito de forma parcial, já que todas elas não avaliam *pointcuts* dinâmicos (isso é feito mais tarde, em tempo de execução); *2)* nenhuma delas dispensa o casamento de sombras de junção, que poderia ser substituído, em tempo de execução, pelo casamento de pontos de junção.

Nós próximos parágrafos questionaremos essas invariantes, verificando por quê elas ocorrem em todas as ferramentas avaliadas.

Casamento parcial de sombras de junção: queremos mostrar que o casamento parcial de sombras de junção é mais eficiente do que o casamento total de sombras de junção, onde a adição de checagens para casar *pointcuts* dinâmicos em tempo de execução seria dispensada.

[3]Para mais detalhes, veja a Seção 4.2.2 na página 59.

Para isso, considere duas operações dinâmicas a serem realizadas num sistema orientado a aspectos. Cada uma delas possui expressões *pointcut* bem diferentes uma da outra [4]:

- *"variável `x` tem valor inteiro maior que `10`"* [5]
- *"execução do método `A.y()`"*

O primeiro *pointcut* casa com todos os pontos do fluxo de controle do sistema nos quais, durante a sua execução, a variável `x` tem valor maior que `10`. Fica claro que não é uma tarefa fácil determinar as sombras de junção cuja execução resultará em pontos de junção que casarão positivamente com esse *pointcut*. De forma intuitiva, é possível afirmar que devemos verificar o valor de `x` em tempo de execução para determinar se a expressão *pointcut* casa com cada ponto de junção do sistema. Expressões *pointcut* que demandam verificações de condições em tempo de execução são ***pointcuts* dinâmicos** [6]. O tipo mais comum de *pointcuts* dinâmicos são expressões do tipo `cflow` (vide a Seção A.1 na página 138).

No caso do segundo *pointcut*, é preciso interceptar todas as execuções do método `A.y()`. É possível determinar de forma trivial quais as sombras de junção afetadas pela expressão *pointcut*. Afinal, somente a execução do corpo desse método irá gerar pontos de junção que casam com esse *pointcut*. *Pointcuts* cujo casamento não depende de condições da execução do sistema base são conhecidos como ***pointcuts* estáticos**.

Em geral, *pointcuts* dinâmicos diferem do *pointcut "variável `x` tem valor inteiro maior que `10`"*, visto há pouco, no sentido de que são compostos e possuem uma parte estática, que não requer verificações durante a execução. Um exemplo disso seria a intersecção dos dois *pointcuts* anteriores:

"(variável `x` tem valor inteiro maior que `10`) && (execução do método `A.y()`)"

Essa intersecção casaria somente com as execuções de `A.y()` que ocorressem quando `x` for maior que `10`. Nesse caso, não é necessária a verificação de todos os pontos de junção do sistema em tempo de execução. Sabemos que essa expressão só pode casar positivamente com execuções do método `A.y()`. Porém, ainda é preciso averiguar o valor de `x` quando esse método for executado, a fim de concluir se o *pointcut* em questão casa ou não com aquele ponto de junção.

Com esses exemplos, é possível entender a diferença entre **casamento de sombras de junção** e **casamento de pontos de junção**. Ambos se utilizam de um *pointcut* para identificar quais pontos de junção casam com essa expressão. Porém, um deles ocorre antes da instrumentação, utilizando-se do código para inferir quais sombras gerarão pontos de junção que casam com aquele *pointcut*, enquanto

[4] Preferimos utilizar uma versão verbosa dos *pointcuts* ao invés de escolhermos uma sintaxe específica. O motivo de nossa escolha se deve ao fato de que essa avaliação envolve conceitos e não ferramentas específicas. Queremos, portanto, ser o mais genéricos possível em nossa análise.

[5] *Pointcuts* desse tipo atualmente não são suportados pela sintaxe de *pointcut* das ferramentas que vimos. Porém, conceitualmente essa é uma expressão válida, e facilmente poderíamos implementá-la através de *pointcuts* extensíveis no JBoss AOP. Como veremos adiante, essa expressão representa de forma clara o conceito que estamos querendo mostrar.

[6] Aspectos cujos adendos são associados a *pointcuts* dinâmicos são denominados **aspectos dinâmicos**. Na literatura de programação orientada a aspectos, alguns autores denominam linguagens que possibilitam a criação de aspectos dinâmicos de linguagens de programação orientada a aspectos dinâmica. Porém, esse é o uso menos comum do termo, que geralmente se refere à combinação dinâmica de aspectos, e não a aspectos dinâmicos.

que o outro é realizado sempre que um ponto de junção for executado, para verificar se o *pointcut* casa com esse ponto de junção.

As ferramentas de POA dinâmica que estudamos na Seção 3.1 utilizam o casamento de sombras apenas para avaliar a parte estática de um *pointcuts*. Para *pointcuts* estáticos, isso equivale a afirmar que o casamento integral do *pointcut* é executado antes da execução dos pontos de junção. Caso se trate de *pointcuts* dinâmicos, ainda existe a verificação de uma condição a ser feita em tempo de execução, o que chamamos de **resíduo**. A única exceção a isso são as duas primeiras implementações do PROSE, com o uso *breakpoints* e de ganchos minimais. Nessas duas, os pontos de junção são casados em tempo de execução com *pointcuts* de forma integral, o que significa que a cadeia de adendos que irá interceptar um ponto de junção é calculada no momento da execução de cada ponto.

Uma vez que as diferenças entre os dois tipos de casamentos está clara, e que revimos como as ferramentas dinâmicas utilizam esses dois tipos para implementar o casamento de *pointcuts* dinâmicos, podemos responder à pergunta inicial: por quê o casamento de sombras não é feito de forma integral, e sim de forma parcial nas ferramentas de POA dinâmica que vimos? A resposta para isso está na complexidade e no desempenho de um suposto casamento integral de *pointcuts* dinâmicos com sombras de junção. Ficou provado que esse casamento é um problema NP-Completo [54], pois requer uma análise complexa de todo o código do sistema base. Com isso, fica claro porque o casamento com sombras de junção deve ser parcial.

Sombras de junção versus Pontos de junção A segunda invariante em todas as ferramentas é que todas realizam casamento de sombras de junção. Nenhuma delas utiliza exclusivamente o casamento de pontos de junção para identificar quais pontos de junção deverão ser interceptados. Nem mesmo o PROSE, nas suas primeiras versões, dispensa o casamento de sombras, apesar de fazer um casamento integral de pontos de junção em tempo de execução (nesse caso, o casamento de sombras é utilizado para definir onde adicionar *breakpoints* ou ativar ganchos minimais). Queremos mostrar que dispensar o casamento de sombras de junção é muito caro, afetando o desempenho do sistema drasticamente.

Sabemos que a averiguação de cada ponto de junção em tempo de execução é totalmente desnecessária para *pointcuts estáticos*. Da mesma forma, sabemos também que o casamento de cada ponto de junção com a parte estática de um *pointcut* dinâmico é desnecessária. Afinal, é possível casar de forma relativamente fácil a parte estática dos *pointcuts* com sombras de junção ao invés de pontos de junção, durante a combinação. Além disso, apesar de ser igualmente fácil e possível ignorar essa etapa de casamento de sombras, postergando o casamento para o momento da execução de cada ponto de junção, estaremos tendo um custo muito maior. O casamento de pontos de junção causa um impacto maior no sistema, por adicionar o custo de verificações de cada ponto de junção durante a sua execução. Sendo assim, esse tipo de casamento deve ser evitado.

Para provar isso, vamos avaliar os exemplos JBoss AOP e PROSE com *breakpoints*. No JBoss AOP, postergar o casamento para a execução significaria que reconstruiríamos uma `cadeia de adendos` a cada vez que uma sombra fosse executada. Além desse custo, que por si só já é proibitivo, teríamos que fazer isso com todas as sombras passíveis de interceptação, já que não teríamos idéia de quais sombras poderiam vir a ser interceptadas em sua execução. Em outras palavras, substituir o casamento de

sombras pelo casamento de pontos de junção no JBoss AOP, se traduziria na construção de uma cadeia de adendos a cada chamada de métodos e construtores, a cada execução de métodos e construtores, e a cada acesso a campos. Virtualmente, isso significa que a cada linha de código teríamos que realizar um casamento de ponto de junção e construir uma cadeia de adendos nova, excluindo-se apenas blocos de tratamento de exceções, e lançamento de exceções. Avaliando a hipótese do PROSE não realizar o casamento de sombras de junção durante a combinação, essa é uma idéia que também traz conseqüências graves para o desempenho. Como não sabemos quais sombras poderão ser interceptadas em tempo de execução, teríamos que adicionar um *breakpoint* em cada linha do código. A cada *breakpoint*, o PROSE reavaliaria o casamento de pontos de junção para determinar quais adendos devem interceptar aquela linha. Isso multiplicaria o impacto dos *breakpoints*, que subiriam de um número gerenciável (um *breakpoint* para cada sombra de junção que casa com um *pointcut*) para o número inconcebível de aproximadamente um *breakpoint* por linha de código.

Com isso, fica claro que dispensar o casamento de sombras durante a combinação não é uma idéia plausível. Assim, devido ao impacto de realizar exclusivamente casamentos de sombras de junção, e ao impacto de fazer o contrário, utilizando exclusivamente casamentos de pontos de junção, a abordagem utilizada pela maioria das linguagens e ferramentas orientadas a aspectos é utilizar um meio termo das duas soluções, como vimos há pouco. A parte estática de uma expressão *pointcut* é resolvida através do casamento de sombras; a parte dinâmica, se houver, é resolvida através do casamento de pontos de junção, o que requer checagens em tempo de execução.

Contudo, mesmo utilizando-se essa abordagem de casamento misto, há espaço para melhorias no impacto que a seleção de pontos de junção pode causar. Atualmente, as ferramentas de POA verificam cada sombra de junção passível de interceptação a cada inserção dinâmica de aspectos, fazendo o casamento da sombra com o *pointcut* adicionado. Isso resulta na repetida leitura e processamento das sombras de junção, problema que será abordado no Capítulo 6.

Por hora, podemos concluir que toda operação dinâmica de adição incorre num custo de casamento de sombras de junção, que consiste na avaliação de todas as sombras preparadas para a combinação dinâmica, a fim de encontrar os pontos de junção afetados por um ou mais *pointcuts*.

3.2.2.2. Alteração do Fluxo de Controle

Uma vez determinadas quais sombras de junção serão afetadas por uma operação de combinação dinâmica, é necessário combinar de fato tais sombras aos adendos associados à expressão *pointcut*, alterando o fluxo de controle dessas sombras a fim de executar os adendos.

Na Seção 3.1, estudamos as diversas abordagens utilizadas para alterar o fluxo de controle em tempo de execução. Dessas, a mais lenta é, sem dúvida, a implementação do PROSE com *breakpoints*. As demais abordagens que vimos utilizam de alguma forma a instrumentação do sistema base, solução mais popular dentre as ferramentas de programação orientada a aspectos dinâmica. Essa predileção pela instrumentação tem um motivo: eficiência. Ao instrumentarmos o sistema base, adicionando código para a execução de adendos nas sombras de junção, evitamos um custo adicional em tempo de execução. Afinal, quando as sombras forem executadas, o código inserido é automaticamente executado, sem a

necessidade de qualquer passo adicional para reavaliar em tempo de execução quais adendos devem ser chamados naquele ponto (salvo o casamento de pontos de junção com *pointcuts* dinâmicos). Veja que não é possível evitar esse passo adicional no caso do PROSE com *breakpoints*. Suponha que tentássemos evitar o casamento de pontos de junção integral realizado em tempo de execução, substituindo-o pelo casamento misto, à maneira que é feito pelas outras ferramentas. Mesmo assim, o PROSE pagaria um custo adicional em tempo de execução. Por não utilizar instrumentação, o PROSE tem que determinar quais adendos deve executar quando um *breakpoint* é atingido. A solução, nesse caso hipotético, seria montar uma estrutura de dados, contendo o resultado dos casamentos de sombras de junção, a fim de controlar quais sombras estão associadas a quais adendos. O custo em tempo de execução seria o custo da busca pela sombra num mapa ou estrutura similar, com o objetivo de encontrar uma cadeia de adendos a ser executada.

Até hoje, a instrumentação se mostrou como sendo a solução mais plausível para a implementação de combinação dinâmica em Java. Podemos classificar o uso da instrumentação em três grupos distintos: o primeiro, onde a instrumentação é realizada exclusivamente durante a preparação; o segundo, onde a instrumentação é feita durante a preparação e em tempo de execução, alterando-se o fluxo de controle mesmo na ausência de adendos; e o terceiro, onde a instrumentação é feita na preparação e em tempo de execução, mas o fluxo de controle só é alterado na presença de adendos.

O primeiro grupo é composto apenas pelo JBoss AOP. O fluxo de controle é alterado desde o início, afetando o desempenho do sistema mesmo na ausência de adendos, devido aos ganchos vazios adicionados nas sombras preparadas. Por outro lado, operações dinâmicas não possuem o custo adicional de instrumentação. Elas apenas resultam na atualização das cadeias de adendos, algo rápido se comparado com a geração e compilação de código através do Javassist.

O segundo grupo também é composto por apenas uma ferramenta, o PROSE, e somente na sua implementação com ganchos minimais. Aqui, existe o impacto adicional da execução desses ganchos, que deve ser menor do que o impacto dos ganchos no JBoss AOP, já que os ganchos minimais são inseridos no código nativo, com otimizações. Existe também o custo da instrumentação de código nativo em tempo de execução, que afetará o fluxo de controle dos ganchos minimais. Além disso, outro custo impactante decorre dessa abordagem com ganchos minimais. Esses ganchos, quando ativos, resultarão na execução do casamento de pontos de junção, da mesma forma que ocorre na implementação do PROSE com *breakpoints*. Podemos concluir que essa abordagem possui diversos impactos no desempenho, tanto no sistema executado sem adendos, como no sistema executado com adendos. Nos pontos do código que não estão sendo interceptados por nenhum adendo, o impacto é o da execução dos ganchos minimais vazios. Para pontos do código cujo gancho está ativo, o impacto é causado pela execução do próprio gancho, pelo casamento do ponto de junção com todos os *pointcuts* inseridos no sistema e pelas chamadas realizadas aos adendos. Classificamos essa como a abordagem mais cara em termos de desempenho dentre as abordagens que envolvem instrumentação.

Finalmente, o último grupo é formado por todas as outras abordagens vistas na Seção 3.1. Nesse grupo, os ganchos são inseridos durante a preparação, mas sem alteração do fluxo de controle. A alteração do fluxo de controle só ocorre como decorrência de uma operação dinâmica, para inserir esses ganchos no fluxo de controle e para inserir as chamadas aos adendos dentro desses ganchos (no

caso de adição de aspectos) bem como para removê-los (no caso de remoção de aspectos). Todas as ferramentas dentro desse grupo requerem o uso de uma API específica de uma máquina virtual, ou o uso da JVMDI, onde o sistema é executado em modo de depuração, resultando numa sobrecarga. A exceção é o AspectWerkz, que provê duas outras opções: o uso da JVMTI e redefinir o `ClassLoader` através do parâmetro `-Xbootclasspath`.

Cada uma das abordagens que descrevemos tem a sua vantagem. A primeira, onde toda a instrumentação é realizada somente durante a preparação, evita o custo de instrumentação em tempo de execução, que pode impactar o desempenho das operações de combinação dinâmica. A segunda, com ganchos minimais, tem a vantagem da instrumentação em código nativo. A terceira, finalmente, não acarreta em custos desnecessários na execução do sistema base quando não há adendos ativos, já que manterá seu fluxo de controle intocado até que aspectos sejam adicionados ao sistema.

3.3. Trabalho Proposto

O trabalho que propomos é a aplicação de novas soluções para a implementação da combinação dinâmica de aspectos, de forma a diminuir o seu impacto no desempenho do sistema base. Implementamos essas soluções com o objetivo estudá-las, a fim de avaliarmos seus impactos positivos e negativos, esperando, com isso, contribuir com as pesquisas na área.

A primeira solução que propomos consiste de uma nova implementação para o casamento de sombras de junção, o primeiro passo da combinação dinâmica. Vimos que essa é uma etapa que não deve ser evitada, já que é de primordial importância para o desempenho do sistema selecionar as sombras de junção que serão afetadas por uma operação dinâmica de adição. Nesse trabalho, propomos a otimização dessa etapa através do uso de um grafo de sombras de junção, onde as sombras serão buscadas utilizando-se como chave de busca o próprio *pointcut*. Mostraremos detalhes desse algoritmo no Capítulo 6.

Outra solução que propomos é a implementação da instrumentação em tempo de execução no JBoss AOP através da funcionalidade de *hot swap* fornecida pela JVMTI, como veremos no Capítulo 7. O objetivo é evitar o custo da execução de ganchos vazios no JBoss AOP, sem necessidade de executar o sistema em modo de depuração, ou de redefinir o *bootclasspath* do sistema, como fazem a maioria das ferramentas que vimos na Seção 3.1.

As soluções que propomos foram implementadas no JBoss AOP e avaliadas de acordo com os resultados obtidos nessa ferramenta. No próximo capítulo veremos uma visão geral da arquitetura interna do JBoss AOP, necessária para o entendimento dos algoritmos implementados, que serão detalhados nos capítulos subseqüentes.

4. JBoss AOP

As idéias propostas neste trabalho foram implementadas e testadas no JBoss AOP. Por esse motivo, dedicamos um capítulo exclusivamente a essa ferramenta, a fim de apresentar os principais componentes da sua arquitetura e de mostrar como eles interagem, combinando aspectos com um sistema base.

4.1. Arquitetura

As principais classes do JBoss AOP estão ilustradas no diagrama de classes da Figura 4.1. Muitas delas já foram apresentadas nos capítulos anteriores, como é o caso de `AspectManager`, `InstanceAdvisor` e `Interceptor`, dentre outras. Nessa seção será possível entender melhor os conceitos que elas representam e as funcionalidades que elas fornecem. As classes que compõem o diagrama da Figura 4.1 são descritas a seguir:

`org.jboss.aop.advice.Interceptor:` internamente, adendos e interceptadores são tratados de forma transparente, como objetos `Interceptor`. Por esse motivo, as cadeias de adendos são também chamadas de **cadeia de interceptadores**. Mais detalhes sobre essa interface serão vistos adiante, na Seção 4.1.2.

`org.jboss.aop.advice.InterceptorFactory` : cria instâncias de interceptadores.

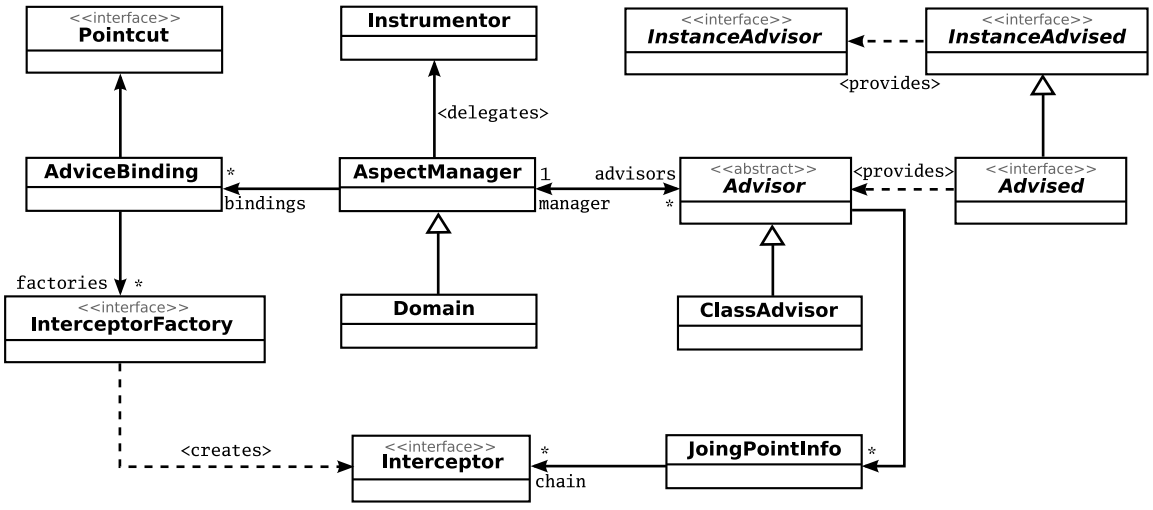


Figura 4.1.: Visão geral da arquitetura do JBoss AOP.

`org.jboss.aop.pointcut.Pointcut`*:* interface que representa uma expressão `Pointcut`. Possui métodos para fazer casamentos com os diversos tipos de ponto de junção.

`org.jboss.aop.advice.AdviceBinding`*:* essa classe foi apresentada na Seção 3.1.1.1 na página 27. Ela representa uma associação de um `Pointcut` a uma coleção de `InterceptorFactory`. Cada uma dessas fábricas representa um adendo ou interceptador, que deverá interceptar todos os pontos de junção que casarem com `Pointcut`.

`org.jboss.aop.AspectManager`*:* é a principal classe do JBoss AOP. Essa classe *Singleton* [27] contém todos os dados lidos durante a configuração. Ela também é a fachada do JBoss AOP e delega a execução das principais operações para as mais diversas classes. Dentre essas operações, a de maior importância para o nosso trabalho é o método `translate`, que delega a instrumentação de uma classe para `Instrumentor`. Esse método realiza a combinação de classes e será o tópico da Seção 4.2.

`org.jboss.aop.instrument.Instrumentor`*:* é a fachada do pacote onde estão as classes que excutam a instrumentação, `org.jboss.aop.instrument`. O seu método principal, `instrument`, instrumenta os `bytecodes` de uma classe, delegando cada etapa da instrumentação às classes internas do pacote.

`org.jboss.aop.Domain`*:* o gerenciamento das classes instrumentadas é dividido numa estrutura hierárquica de domínios. Vimos na Seção 3.1.1.1 que o domínio é também parte da API de operações dinâmicas. A divisão de um sistema em domínios será abordada na Seção 4.1.4.

`org.jboss.aop.Advisor`*:* é responsável pela construção e manutenção de cadeias de adendos dentro de um contexto determinado. Tal contexto pode ser uma classe ou uma instância específica. Um objeto `Advisor` está associado a uma instância de `AspectManager`, que representa o domínio a que o `Advisor` pertence. Essa classe possui uma vasta hierarquia de subclasses. Neste texto iremos apresentar apenas a subclasse `ClassAdvisor`, omitindo as demais classes de sua hierarquia, pois elas não são necessárias para a compreensão deste trabalho. Para simplificar, iremos nos referir a objetos dessa classe e de suas subclasses por *advisors*.

`org.jboss.aop.ClassAdvisor`*:* subclasse de `Advisor` que gerencia todas as cadeias de adendos a serem aplicadas para cada ponto de junção de determinada classe transformada. Essas cadeias são encapsuladas em objetos `JoinPointInfo`.

`org.jboss.aop.InstanceAdvisor`*:* interface responsável pelo gerenciamento das pilhas `around` a serem aplicadas aos pontos de junção da instância específica a ela associada (veja a Seção 3.1.1.1 na página 27).

`org.jboss.aop.JoinPointInfo`*:* internamente, sombras de junção são representadas por objetos `JoinPointInfo`. Essa é a classe vista na Seção 2.4.1 na página 10, que também mostra a hierarquia de subclasses, onde cada tipo representa um tipo de sombra de junção. Um `JoinPointInfo` é quem possui a cadeia de interceptadores que deverá ser invocada durante a execução da sombra

de junção que representa. Esses objetos são armazenados em `Advisors`, responsáveis por criar e manter a cadeia de interceptadores de cada `JoinPointInfo`.

`org.jboss.aop.Advised`*:* interface implementada por todas as classes instrumentadas, permite que se obtenha o `Advisor` da classe a partir de um objeto da classe instrumentada.

`org.jboss.aop.InstanceAdvised`*:* essa interface permite que se obtenha o `InstanceAdvisor` associado a uma instância instrumentada. Os métodos dessa interface, bem como os da interface `Advised`, podem ser vistos na Listagem 3.3 na página 30.

Uma vez apresentada a visão geral da arquitetura do JBoss AOP, vamos nos aprofundar em alguns grupos de classes. Na próxima seção, veremos como *pointcuts* são tratados internamente, avaliando sua estrutura de representação interna e o casamento dessas expressões com *joinpoints*. Após, mostraremos como aspectos, adendos e interceptadores são representados internamente em detalhes. Na Seção 4.1.3, será o momento de descrevermos as classes envolvidas na instrumentação. Finalmente, a estrutura de domínios do AOP será abordada na Seção 4.1.4.

4.1.1. Pointcuts

Internamente, o JBoss AOP utiliza o JJTree para processar as suas expressões *pointcut*. Trata-se de um pré-processador de texto que integra a ferramenta JavaCC [39], e transforma uma expressão analisada numa árvore AST (*abstract sintax tree*). No JBoss AOP, a árvores sintáticas dos *pointcuts* analisados ficam em objetos da classe `org.jboss.aop.pointcut.ast.ASTPointcut`. Expressões `cflow` também são transformadas em árvores sintáticas, contidas em objetos do tipo `org.jboss.aop.pointcut.ast.ASTCFlowExpres`

As árvores de análise contidas em objetos `ASTPointcut` e `ASTCFlowExpression` podem ser visitadas (padrão `Visitor` [27]) por uma implementação da interface `org.jboss.aop.pointcut.-ast.PointcutExpressionParserV`. O JBoss AOP possui diversas implementações desse *visitor*, uma para cada tipo de casamento que ele faz:

`org.jboss.aop.pointcut.ConstructorMatcher`*:* casamento de *pointcuts* com execuções de construtores;

`org.jboss.aop.pointcut.MethodMatcher`*:* casamento de *pointcuts* com execuções de métodos;

`org.jboss.aop.pointcut.ConstructorCallMatcher`*:* casamento de *pointcuts* com chamadas de construtores;

`org.jboss.aop.pointcut.MethodCallMatcher`*:* casamento de *pointcuts* com chamadas de métodos;

`org.jboss.aop.pointcut.FieldMatcher`*:* casamento de *pointcuts* com leituras e escritas de campos;

`org.jboss.aop.pointcut.SoftClassMatcher`*:* casamento de *pointcuts* com nomes de classes;

`org.jboss.aop.pointcut.CflowMatcher`*:* casamento de expressões `cflow` com pontos de junção.

Os cinco primeiros *visitors*, `ConstructorMatcher`, `MethodMatcher`, `ConstructorCallMatcher`, `MethodCallMatcher` e `FieldMatcher`, casam *pointcuts* com sombras de junção. Cada um deles contém uma sombra de junção, que é do tipo correspondente ao tipo do *visitor*. Para executar o casamento, eles percorrem a árvore de análise de um *pointcut*, procurando por uma estrutura que identifique pontos de junção do mesmo tipo a ser casado (por exemplo, um `MethodMatcher` irá procurar apenas por estruturas que especifiquem execução de métodos). Se essa expressão não for encontrada, o *visitor* devolverá `false`, indicando que o *pointcut* analisado não casa com a sombra de junção que aquele *visitor* contém. Caso contrário, uma vez encontrada essa expressão, terá início uma comparação das partes da expressão com a sombra a ser casada, a fim de definir se a expressão casa ou não com aquela sombra de junção. Para ilustrar, considere um objeto `FieldMatcher` que contém como sombra de junção a leitura do campo `int length` de uma classe chamada `Pojo`. Ao visitar a árvore de análise do *pointcut* `"get(int Pojo->count)"`, a fim de determinar se a sombra que contém casa com essa expressão, o `FieldMatcher` procura por construções que casem com leituras de campos. Uma vez detectada a presença de tal estrutura no *pointcut*, ele começa a comparar a expressão com a sombra de junção que está casando. Dessa forma, ele irá comparar o nome da classe, `Pojo`, com o nome da classe alvo da sombra de junção, que também é `Pojo`. Em seguida, ele irá comparar o nome do campo, `count`, com o nome do campo na sombra que está casando, `length`. Naturalmente, o `FieldMatcher` irá determinar que os nomes não batem, devolvendo `false`.

Além dos *visitors* que casam *pointcuts* com sombras de junção, existem os *visitors* `SoftClassMatcher` e `CFlowMatcher`. O primeiro faz casamentos de *pointcuts* com nomes de classes, algo que chamamos de **casamento *soft***. Esse casamento consiste em extrair o nome da classe alvo de qualquer *pointcut* e casar essa parte com o nome de uma classe. Dado o *pointcut* `"get(int Pojo->count)"`, por exemplo, esse *visitor* extrai o nome da classe `"Pojo"`. Ao visitar a árvore de análise da expressão `"get(int *oj*->count)"`, o `SoftClassMatcher` extrai a expressão `"*oj*"`. A parte extraída do *pointcut* é utilizada para realizar casamentos com nomes de classes. Em outras palavras, essa classe faz casamentos parciais, comparando apenas o nome de classes com uma classe em particular. O `SoftClassMatcher` é utilizado para otimizar a combinação, como veremos adiante, na Seção 4.2.

O *visitor* `CFlowMatcher` é responsável pelo casamento de expressões `cflow` com pontos de junção. Por se tratar de um *pointcut* dinâmico, o casamento de `cflow` é realizado em tempo de execução. O `CFlowMatcher` compara a pilha de execução de um ponto de junção com uma lista de restrições que determinam a presença e ausência de métodos e construtores na pilha. Se a pilha de execução atender a todas as restrições, o `CFlowMatcher` devolverá `true`.

Veremos no próximo capítulo a sintaxe dos *pointcuts* do JBoss AOP. Por ora, é suficiente entender como essas expressões são representadas internamente e quais as principais características que possuem. *Pointcuts* no JBoss AOP têm uma grande flexibilidade quanto à forma de identificar os pontos de junção. Nomes de campos, classes, métodos e parâmetros podem conter curingas, indicando padrões que devem ser casados com os nomes em questão. Um exemplo é a expressão `"get(int *oj*->count)"`, que utiliza o padrão `"oj"` no lugar do nome da classe e, portanto, casa com a leitura de campos `int count` de qualquer classe que contenha a seqüência de caracteres `"oj"` em seu nome. Além disso, é possível definir regras que determinam a presença de um campo, método, ou construtor na classe alvo, ou definir

apenas o pacote em que essa classe se encontra, ou, ainda, definir anotações presentes na classe. Esses são apenas exemplos das restrições que podem ser especificadas em *pointcuts* e `cflows`, mas servem para demonstrar que o casamento dessas expressões não é uma tarefa trivial. Felizmente, existem vários pontos em comum na execução dos *visitors*. Esses pontos se traduzem na forma de métodos que realizam tarefas comuns a todos os *visitors* e estão presentes na classe `org.jboss.aop.pointcut.Util`. Os *visitors* delegam diversas etapas do casamento para essa classe, que realiza as tarefas mais complexas de comparação durante o casamento de `pointcuts`.

Vimos na seção anterior que as expressões *pointcut* são representadas internamente através da interface `Pointcut`. Essa interface possui métodos para casar *pointcuts* com todos os tipos de sombras de junção. Ela é implementada pela classe `PointcutExpression`, que contém um objeto `ASTPointcut` e delega o casamento com sombras de junção para os *visitors* apropriados.

4.1.2. Aspectos, Adendos e Interceptadores

Veja o diagrama da Figura 4.2. Nele, podemos ver as classes utilizadas internamente para a representação de aspectos, adendos e interceptadores.

Aspectos são representados pela classe `AspectDefinition`. Essa classe contém nome, escopo e uma referência para uma fábrica de aspectos, instância de `AspectFactory`. Essa fábrica é responsável por criar todas as instâncias do aspecto necessárias durante a execução do sistema. Quando o aspecto é declarado através de uma *factory*, e não através do nome da classe, essa referência apontará para a fábrica fornecida pelo usuário. Caso contrário, o JBoss AOP cria uma classe do tipo `GenericAspectFactory`, responsável por criar instâncias da classe configurada pelo usuário (no caso de arquivos `jboss-aop.xml`, podemos nos recordar que o nome da classe do aspecto era determinado no atributo `class` da *tag* `aspect`; no caso de anotações, o classe do aspecto é aquela que foi anotada com `@Aspect`). As definições de aspectos (i.e., instâncias de `AspectDefinition`) ficam armazenadas numa coleção dentro do `AspectManager`.

Um interceptador também é representado por um objeto `AspectDefinition`. Quando, durante a configuração, uma declaração de interceptador é encontrada, ela se traduz na criação de uma definição de aspectos e de uma fábrica de interceptadores. Em outras palavras, a representação interna de um interceptador é dividida em duas partes: aspecto, através da `AspectDefinition` (essa identifica a classe do interceptador e como ela deverá ser instanciada); e interceptador, no sentido de ser o elemento que compõe uma cadeia que será utilizada para a interceptação. Nesse último caso, interceptadores são representados por instâncias de `InterceptorFactory`. Como mostra o diagrama da Figura 4.1 na página 48, as fábricas de interceptadores são mantidas em associações `AdviceBinding` e são utilizadas para criar cadeias de adendos. Podemos ver no diagrama que `InterceptorFactory` é apenas uma interface. A classe utilizada para instanciar interceptadores no JBoss AOP é a `ScopedInterceptorFactory`. Essa fábrica de interceptadores, quando invocada, delega sua execução para a fábrica de aspectos contida na definição de aspectos. Durante esse passo, o escopo do interceptador é repassado para a fábrica de aspectos.

Quanto aos adendos, esses são representados internamente através de instâncias de `AdviceFactory`.

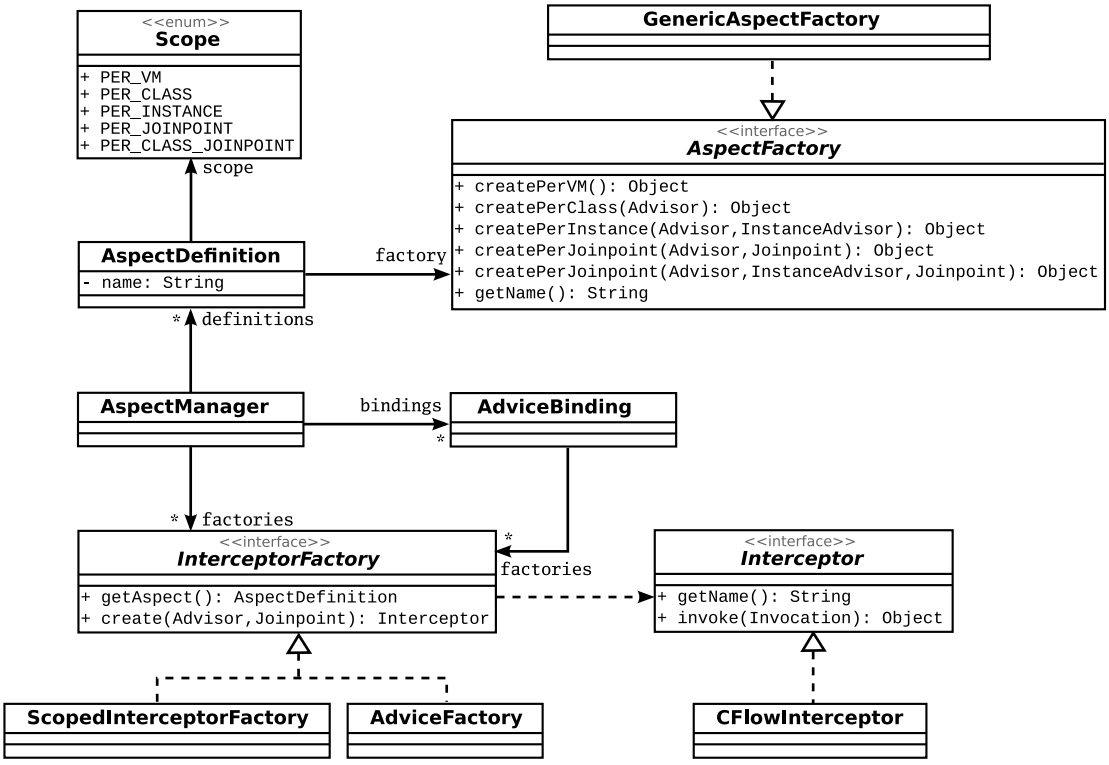


Figura 4.2.: Classes utilizadas para representar aspectos, interceptadores e adendos. Todas as classes desse diagrama, exceto `AspectManager`, pertencem ao pacote `org.jboss.aop.advice`.

Essa classe implementa a interface `InterceptorFactory` e deve, portanto, criar instâncias de interceptadores. A classe `AdviceFactory` cria adaptadores, que implementam a interface `Interceptor`[1] e invocam o adendo adaptado sempre que o método `invoke` é executado. Isso permite que adendos sejam tratados como interceptadores de forma transparente. A listagem a seguir mostra como seria o adaptador gerado para um adendo chamado `MyAspect.logAdvice(Invocation invocation)`, do tipo *around*:

```
1 package org.jboss.aop.advice;
2
3 public class MyAspect_z_logAdvice implements Interceptor
4 {
5     // aspecto que contém o adendo logAdvice
6     MyAspect aspect;
7
8     // retorna um nome gerado pelo JBoss AOP, que identifica
9     // esse interceptador unicamente
```

[1] O método `invoke` da interface `Interceptor` foi especificado na página 14.

```
10      public String getName()
11      {
12          ...
13      }
14
15      // quando o interceptador é invocado, ele delega a execução
16      // para o adendo logAdvice da classe MyAspect
17      public Object invoke(Invocation invocation) throws Throwable
18      {
19          return aspect.logAdvice(invocation);
20      }
21  }
```

Listagem 4.1: Exemplo de adaptador gerado por `AdviceFactory`.

A classe `ClassAdvisor` é responsável por invocar as fábricas de interceptadores, operação realizada quando essa classe constrói as cadeias de adendos associadas a cada sombra de junção. Quando há presença de `cflow` na associação que contém as fábricas de interceptadores, o `ClassAdvisor` cria um interceptador especial, que encapsula os interceptadores da associação. Esse interceptador, `CFlowInterceptor`, realiza o casamento do `cflow` com o ponto de junção, invocando a cadeia que encapsula somente se o casamento do `cflow` for positivo:

```
1   public class CFlowInterceptor implements Interceptor
2   {
3       // cadeia de interceptadores que esse interceptador encapsula
4       Interceptor[] chain;
5       // árvore de sintaxe de uma expressão cflow
6       ASTCFlowExpression expr;
7
8       ...
9
10      public Object invoke(Invocation invocation) throws Throwable
11      {
12          // se o casamento cflow com o ponto de junção representado
13          // por invocation é positivo
14          if (new CFlowMatcher().matches(this.expr, invocation))
15          {
16              // insere a cadeia chain na pilha around e procede
17              // a execução para o próximo interceptador da pilha
18              return invocation.getWrapper(this.chain).invokeNext();
19          }
20          // caso contrário, apenas procede a execução
```

```
21          // para o próximo interceptador da pilha around
22          return invocation.invokeNext();
23      }
24  }
```

Listagem 4.2: Trechos da classe `CFlowInterceptor`. Esse interceptador encapsula uma cadeia e faz uma verificação em tempo de execução, casando uma expressão `cflow` com o ponto de junção representado por `Invocation`. Somente se o casamento tem resultado positivo esse interceptador executará a cadeia que encapsula, `chain`.

Da mesma forma que as definições de aspecto, as fábricas de interceptadores ficam armazenadas em uma coleção do `AspectManager`. Existem também referências para fábricas de interceptadores na classe `AdviceBinding`. Além de conter uma instância de `Pointcut`, e uma de `ASTCFlowExpression`[2], a classe `AdviceBinding` contém um vetor de objetos do tipo `InterceptorFactory`, responsável por criar a cadeia de interceptadores que será invocada durante a interceptação dos pontos de junção afetados pela associação em questão.

4.1.3. Transformadores

A classe `org.jboss.aop.instrument.Instrumentor` é a fachada que realiza a instrumentação de classes. Essa funcionalidade é provida através do seu método `instrument`, que recebe como argumento uma classe no formato do Javassist, `CtClass`, e processa os seus *bytecodes*, instrumentando-os quando necessário. Esse método também recebe o `ClassAdvisor` como argumento, pois ele contém informações sobre introdução de metadados. Esses metadados são anotações adicionadas a métodos, construtores, campos e à própria classe, e podem ser utilizados para casar expressões *pointcuts* (para mais detalhes, veja a Seção A.3 no Apêndice A). Por esse motivo, é necessário ter acesso ao `ClassAdvisor` da classe que será transformada. O método `instrument` devolve um valor booleano para indicar se a classe foi instrumentada ou não:

```
public boolean transform(CtClass clazz, ClassAdvisor advisor)        (4.1.1)
```

Para executar o método `transform`, o `Instrumentor` utiliza uma série de classes transformadoras:

`org.jboss.aop.instrument.ConstructorTransfomer:` instrumenta construtores;

`org.jboss.aop.instrument.MethodTransformer:` instrumenta métodos;

`org.jboss.aop.instrument.CallTransformer:` instrumenta chamadas;

`org.jboss.aop.instrument.FieldTransformer:` instrumenta acessos a campos (leitura e escrita).

Cada um desses transformadores casa todas as sombras de junção da classe a ser transformada com os *pointcuts* registrados no domínio principal. Toda sombra de junção identificada por um ou

[2]Quando não há restrições `cflow` numa associação, essa referência é nula.

mais *pointcuts* tem os seus *bytecodes* instrumentados, com a adição de ganchos para a invocação dos adendos. A forma como a chamada a esses ganchos é inserida no código difere entre os transformadores. O `MethodTransformer` substitui o corpo do método original pelo corpo do gancho. Esse gancho fará uma invocação a um método inserido na classe, que contém o corpo do método original. Essa é a transformação mais simples, por consistir na inserção do gancho em um único ponto: o próprio método a ser interceptado. Os outros transformadores, por outro lado, inserem chamadas a um gancho gerado em diversos pontos da classe. O `ConstructorTransformer` precisa substituir todas as chamadas ao construtor por uma chamada a um gancho. O mesmo ocorre com o `CallTransformer` para chamadas de métodos e construtores. No caso do `FieldTransformer`, pontos do código que lêem ou alteram o valor de um campo precisam ser encontrados e substituídos por chamadas ao gancho correspondente sempre que tiverem que ser interceptados.

É preciso detectar também chamadas a construtores e acessos a campos de outras classes. Todos eles têm que ser substituídos pela chamada a um gancho se forem pontos interceptados. Essa substituição é realizada pelo próprio *Instrumentor*. Essa classe é também responsável por adicionar interfaces e *mixins* (vide o Apêndice A para mais informações sobre essas funcionalidades).

4.1.4. Domínios

No JBoss AOP, os domínios são uma divisão do sistema de POA em unidades menores. Existe um domínio para cada instância, responsável apenas por gerenciar aquela instância. Acima desse nível de domínios de instância, encontramos os domínios de classe. Esses realizam operações apenas em uma classe instrumentada, e possuem os domínios de instância como subdomínios. Os domínios de classe compõem o **domínio principal**, que é a instância única de `AspectManager`. Nesse sentido, esse objeto é considerado um domínio apenas semanticamente, já que ele não é uma instância da classe `Domain`, e sim uma instância da classe `AspectManager`. A estrutura que acabamos de descrever pode ser vista na Figura 4.3. Nessa figura, vemos os domínios de instância, contidos em domínios de classe, que, por sua vez, fazem parte do domínio principal.

A disposição hierárquica de domínios ilustrada pela Figura 4.3 não é a única organização possível para domínios. No servidor de aplicações JBoss AS, existe um nível adicional nessa hierarquia, onde domínios de classe se agrupam para representar os escopos das aplicações[3]. Cada aplicação com escopo terá um domínio correspondente, que gerencia apenas as cadeias de adendos associadas às classes daquele escopo. Nessa configuração, os domínios de escopo ficam dentro do domínio principal.

Atualmente, não existe uma API para obter domínios de escopo, que são utilizados apenas internamente pelo servidor. Os domínios de instância e de classe, por outro lado, podem ser obtidos através da interface `Advised` (veja a Listagem 3.3 na página 30). Essa interface disponibiliza o `ClassAdvisor` e o `InstanceAdvisor` da instância através dos métodos `_getAdvisor()` e `_getInstanceAdvisor()`,

[3]No JBoss AS, escopos são utilizados na estrutura de `ClassLoader`, com o intuito de restringir a visibilidade de classes entre aplicações. A escolha do escopo ao qual pertence cada aplicação é feita pelo usuário, que também pode optar pela ausência de escopo. Nesse caso, as classes de uma aplicação que não pertence a um escopo podem estar visíveis às classes das outras aplicações.

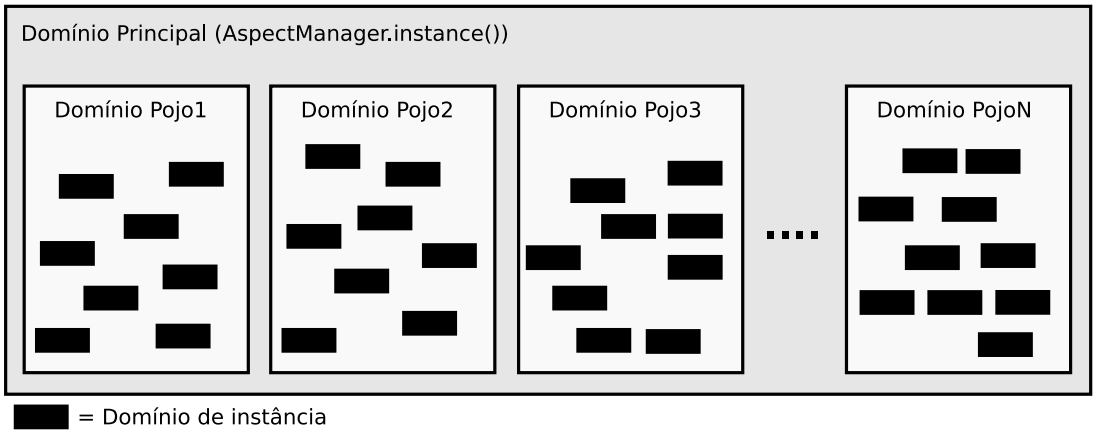


Figura 4.3.: Estrutura de domínios numa aplicação *standalone*. Existe um domínio principal, que é a instância única de `AspectManager`. Dentro dele, encontramos os domínios de classe, responsáveis por gerenciar todas as cadeias de adendos associadas àquela classe. Como mostra o diagrama, existe um domínio para cada uma das N classes instrumentadas, representadas aqui pelos nomes de `Pojo1` a `PojoN`. Existe também um domínio para cada instância, responsável por gerenciar apenas as cadeias de adendos de uma instância específica. Esses domínios estão inseridos nos domínios das classes correspondentes.

respectivamente. Através deles, é possível obter o domínio e, assim, realizar operações dinâmicas com ele:

```
1  Pojo pojo = new Pojo();
2  // obter o ClassAdvisor de Pojo
3  Advisor classAdvisor = ((Advised) pojo)._getAdvisor();
4  // obter o domínio da classe
5  AspectManager domain = classAdvisor.getManager();
6  // criar um binding
7  AdviceBinding binding = new AdviceBinding(...);
8  ...
9  // executar operações dinâmicas no domínio
10 domain.addBinding(binding);
```

Listagem 4.3: Obtenção de um domínio de classe.

O código correspondente à listagem acima para domínios de instância é bem similar. Porém, como `InstanceAdvisor` é apenas uma interface, não há como saber se o `InstanceAdvisor` obtido é uma subclasse de `Advisor` (classe que possui o método `getManager()`). Dessa forma, é preciso fazer uma verificação para o tipo do *advisor* obtido, antes de chamar o método `getManager()`, que devolve o domínio daquele *advisor*[4].

[4]Atualmente, em apenas um dos modos de instrumentação do JBoss AOP é possível acessar o domínio da

4.2. Combinação

Nesta seção iremos mostrar como as principais classes do JBoss AOP colaboram para realizar a combinação de aspectos e interceptadores.

4.2.1. Combinação Estática

A combinação estática, no JBoss AOP, é feita classe a classe. Quando realizada em tempo de compilação, todas as classes são processadas de uma só vez. Quando realizada em tempo de carga, as classes são processadas à medida que são carregadas por um *class loader*.

O ponto de entrada está na fachada `AspectManager`, através do método `transform`, que processa uma classe, transformando-a se necessário. Tanto o compilador `aopc` quanto o agente JVMTI invocam esse método para cada classe que querem processar.

O método `transform` faz algumas verificações sobre a classe. Caso ela seja uma interface ou pertença a um pacote de biblioteca, a transformação é abortada. Caso contrário, o `AspectManager` cria um `Advisor` para aquela classe e delega a execução para o método `transform` de `Instrumentor`. O `Instrumentor`, por sua vez, delegará a instrumentação dos pontos de junção daquela classe para os diversos tipos de transformadores, como foi explicado na Seção 4.1.3.

É curioso notar que o código dos ganchos inseridos durante a transformação não possui nenhuma referência aos adendos e associações. Isso se deve à própria natureza dinâmica do JBoss AOP. Esses ganchos apenas executam uma cadeia de adendos associada a uma sombra de junção. Durante a execução do sistema, um `Advisor` é criado para cada classe instrumentada, e é esse objeto o responsável por inicializar as cadeias de adendos de cada sombra de junção daquela classe.

Por esse motivo, quando fazemos a instrumentação em tempo de compilação (com `aopc`), o arquivo de configuração `jboss-aop.xml` deve estar disponível também durante a execução do sistema. Quando o sistema compilado pelo `aopc` é executado, o `AspectManager` é inicializado, carregando o arquivo. E assim, à medida que as classes instrumentadas são carregadas, os `Advisors` são criados e inicializados. Esses utilizam as informações configuradas, armazenadas no `AspectManager`, para montar as cadeias de interceptadores. Como sabemos, essas cadeias serão invocadas pelos ganchos que foram previamente inseridos no código durante a instrumentação.

---

instância dessa forma. O modo de instrumentação pode ser configurado pelo usuário e determina detalhes de implementação dos ganchos gerados para a invocação de adendos. Atualmente existem três modos: *non-optimized*, onde a chamada ao ponto de junção, dentro da pilha *around*, é feita por reflexão; *classic*, onde o ponto de junção englobado é invocado diretamente, sem reflexão; e *generated advisor*, similar ao modo *classic*, com a diferença de que o código que realiza chamadas aos adendos é gerado em tempo de execução, de forma *lazy*, à medida que as sombras de junção são executadas. É no modo *generated advisor* que o `InstanceAdvisor` obtido é uma subclasse de `Advisor`, o que nos permite acessar o domínio da instância. Nos outros modos, isso já não se aplica. O motivo dessa limitação é que os domínios são uma funcionalidade nova do JBoss AOP e ainda não foram totalmente integrados com o código restante.

4.2.2. Combinação Dinâmica

Vimos no capítulo anterior que a combinação dinâmica é feita classe a classe, da mesma forma que a combinação estática. Sempre que uma operação dinâmica de adição é realizada (`addBinding`), o `AspectManager` invoca o método `updateAdvisorsForAddedBinding`. Esse método é responsável por atualizar as cadeias de adendos dos *advisors*. Para isso, ele itera por todos os *advisors* que gerencia, executando o casamento *soft* para as classes associadas a esses *advisors*. Caso o casamento *soft* tenha um resultado positivo, o `AspectManager` notifica o *advisor* de que ele deve reconstruir as cadeias de adendos da classe que gerencia. Para tanto, o *advisor* faz o casamento de cada sombra de junção daquela classe com cada um dos *pointcuts* registrados em seu domínio, recriando as cadeias de adendos através das fábricas de interceptadores presentes nas associações. O diagrama da Figura 4.4 ilustra esse processo.

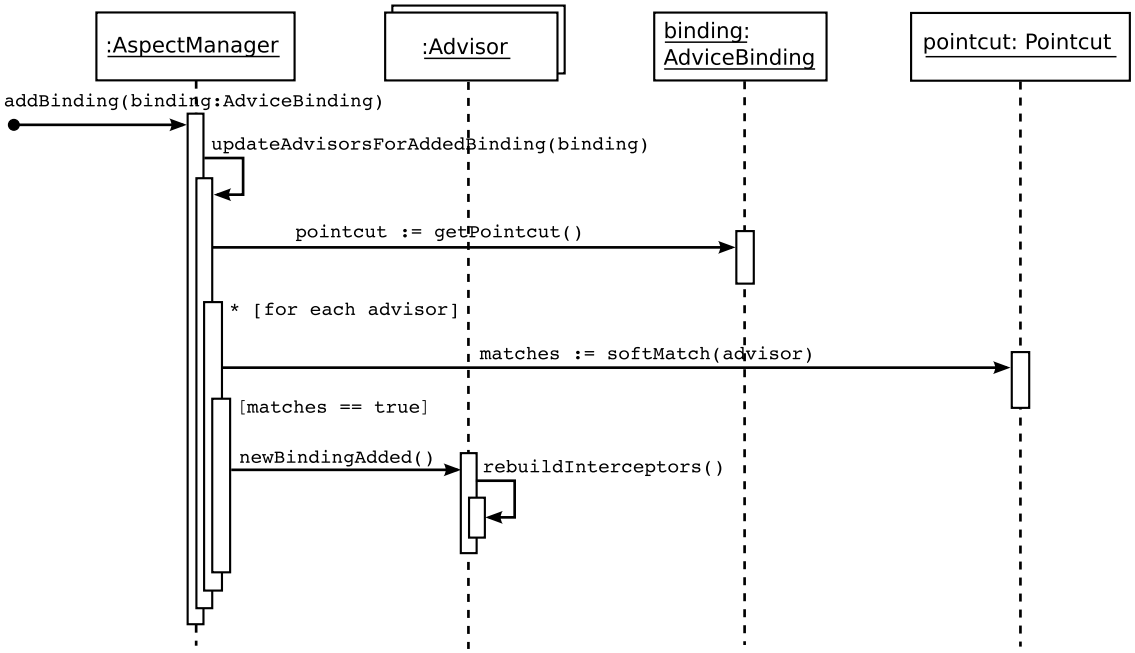


Figura 4.4.: Diagrama de seqüência que ilustra a execução do método `AspectManager.addBinding(AdviceBinding)`.

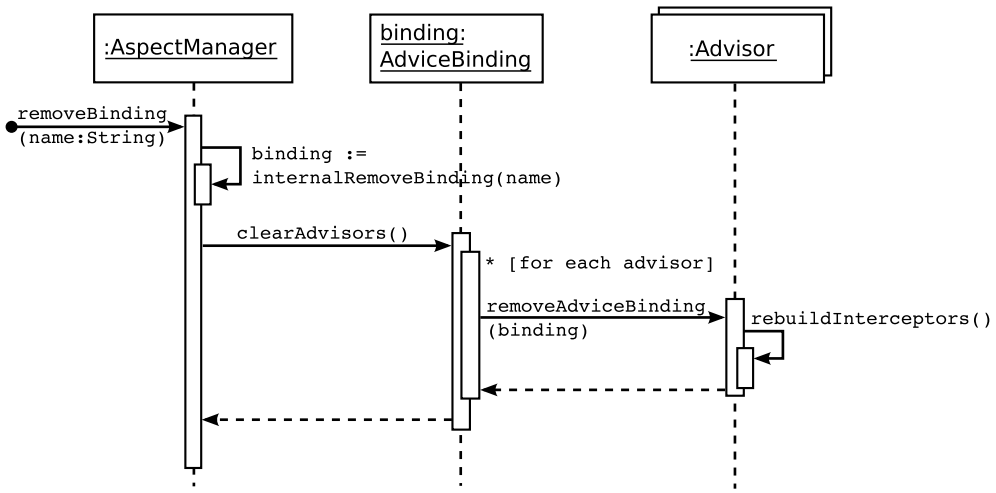


Figura 4.5.: Diagrama de seqüência que ilustra a execução do método `AspectManager.removeBinding(String)`.

Durante o processo de adição de associações, a coleção de `AdviceBinding` que `AspectManager` possui é atualizada. Essa coleção permitirá a remoção da associação no futuro. Além disso, durante o processo de construção de cadeias (`Advisor.rebuildInterceptors()`), cada `AdviceBinding` envolvido no processo é notificado sobre os *advisors* que possuem sombras afetadas por ele. Como resultado, o `AdviceBinding` forma uma coleção de *advisors*, que também será utilizada para a remoção do `AdviceBinding` no futuro. Essa remoção é ilustrada na Figura 4.5.

Como mostra a figura, a remoção não envolve casamentos `soft`. O `AspectManager` simplesmente encontra a associação a ser removida em sua coleção interna e invoca o método `clearAdvisors`. Esse método notifica cada *advisor* que utilizou aquela associação para criar cadeias sobre a remoção dessa associação. Como resultado, os *advisors* reconstroem suas cadeias de interceptadores.

5. Atualização de Cadeias de Adendos

A primeira parte do nosso trabalho visa otimizar a atualização das cadeias de adendos, realizada sempre que uma operação dinâmica ocorre. A implementação atual do JBoss AOP é uma abordagem ingênua. Quando uma associação é adicionada ou removida em tempo de execução, o casamento *soft* é executado para cada classe transformada, a fim de determinar se a classe pode conter uma ou mais sombras afetadas pela operação. Se esse passo não descartar a classe, então o JBoss AOP reconstrói todas as cadeias de adendos que interceptarão pontos de junção daquela classe, garantindo que elas estarão atualizadas conforme a operação dinâmica que foi realizada.

Infelizmente, a aplicação dessa solução gera um impacto indesejado no desempenho de uma operação dinâmica, pois todas as classes do sistema terão que ser verificadas no processo, inclusive aquelas que não serão interceptadas pelos aspectos adicionados ou removidos. Além disso, para cada cadeia a ser reconstruída, o JBoss AOP faz o casamento da sombra com cada um dos *pointcuts* contidos no `AspectManager`, o que significa que o número de *pointcuts* presentes no sistema durante uma operação dinâmica impacta o desempenho da atualização de cadeias. O pior caso é quando as classes que não são descartadas no casamento *soft* não contêm sombras de junção afetadas pela operação dinâmica executada. No JBoss AOP, essas classes terão todas as suas cadeias de adendos reconstruídas, resultando em cadeias exatamente iguais às existentes antes da operação dinâmica ter sido realizada.

Para eliminar o custo de verificar todas as classes que possuem sombras preparadas durante a seleção de sombras, bem como o custo de casar sombras com todos os *pointcuts* existentes, nós propomos substituir esse algoritmo por um outro, mais eficiente. O nosso algoritmo determina quais sombras são afetadas por um *pointcut* sem que seja necessário verificar cada classe e sombra preparada do sistema. Ele também não reconstrói as cadeias associadas a essas sombras. Ao invés disso, os novos interceptadores são apenas adicionados ou removidos dessas cadeias, conforme a operação de adição ou remoção que está sendo executada.

O algoritmo proposto deve criar uma estrutura de dados que armazene todas as informações relevantes sobre as classes carregadas e as suas sombras. Chamamos essa estrutura de **grafo de sombras**. Quando uma associação é adicionada dinamicamente, utilizamos esse grafo para buscar pelas sombras de junção que casam com o *pointcut* da associação. As cadeias das sombras encontradas são apenas alteradas, com a adição dos interceptadores declarados na nova associação. Quando uma associação é removida, também queremos evitar a reconstrução das cadeias afetadas. Para isso, é preciso manter um controle dos interceptadores criados pelas fábricas contidas em cada associação e mapear esses interceptadores para as cadeias que os contêm. Desse modo, basta remover os interceptadores das cadeias afetadas quando a associação for removida.

O uso desse algoritmo de atualização de cadeias é opcional e pode ser habilitado através de uma

propriedade de sistema denominada `jboss.aop.matching`. Para utilizar o algoritmo original, o valor dessa propriedade deve ser `"standard"`. Caso contrário, o seu valor deve ser `"indexed"`, o que habilitará a solução otimizada que descrevemos nesse capítulo para a atualização de cadeias de adendos.

Internamente, o algoritmo de atualização de cadeias é determinado através de uma aplicação do padrão *Strategy* [27]. A estratégia `DynamicMatchingStrategy` e as classes que a implementam podem ser vistas na Figura 5.1. Essa estratégia é responsável por prover:

- um registro de objetos `JoinPointInfo`: esse registro será utilizado para adicionar ao grafo informações sobre as sombras, representadas por objetos `JoinPointInfo`;
- um grafo de sombras para o domínio principal (`AspectManager.instance()`): esse grafo é capaz de buscar por sombras que casam com um *pointcut* dentro do domínio principal;
- um grafo de sombras para um domínio qualquer: esse grafo busca pelas sombras que casam com um *pointcut* dentro de um domínio.

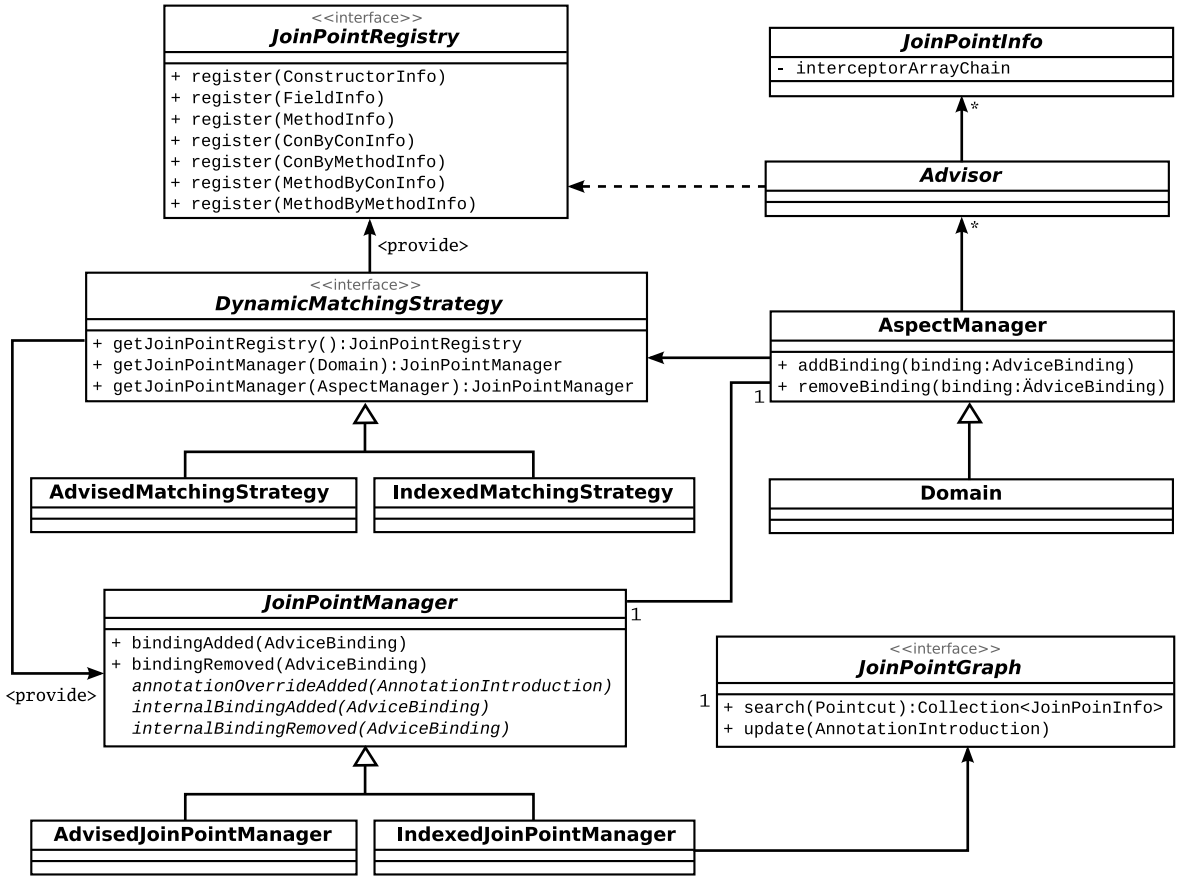


Figura 5.1.: Arquitetura das classes envolvidas na atualização de cadeias.

Existem duas versões dessa estratégia. A primeira, `AdvisedMatchingStrategy`, representa o algoritmo original, que utiliza os *advisors* para reconstruir cadeias como foi descrito no capítulo anterior. A segunda é a classe `IndexedMatchingStrategy` e habilita o algoritmo de atualização de cadeias que implementamos, utilizando um grafo para buscar informações sobre as sombras quando uma associação dinâmica é adicionada. Veremos detalhes dessa solução nas próximas seções.

5.1. O Grafo de Sombras de Junção

O grafo de sombras de junção é uma estrutura de dados composta por informações sobre as sombras de junção preparadas do sistema base. O objetivo desse grafo é substituir o casamento de um *pointcut* por uma busca. Assim, quando uma associação é adicionada em tempo de execução, esse grafo deve utilizar o *pointcut* dessa associação para encontrar as sombras de junção que precisam ter as suas cadeias de adendos atualizadas.

O grafo é representado pela interface `JoinPointGraph`, ilustrada na Figura 5.1. Ele possui dois métodos:

- `Collection<JoinPointInfo> search(Pointcut pointcut)`
- `void update(AnnotationIntroduction annotationOverride)`

O método de busca `search` devolve uma coleção de objetos `JoinPointInfo`, que representam as sombras de junção que casam com o *pointcut* especificado como parâmetro. Esses objetos devolvidos possuem as cadeias de interceptadores que serão atualizadas.

O segundo método, `update`, notifica o grafo sobre a adição de uma introdução de anotações. Essa funcionalidade permite que se adicione anotações a elementos do sistema base em tempo de execução, e é descrita brevemente no Apêndice A. O grafo deve ser atualizado sempre que tal operação for executada, pois ele armazena informações sobre anotações. Essas informações são relevantes para *pointcuts* no JBoss AOP, que podem utilizar uma anotação para casar com uma classe ou um método, como no exemplo a seguir:

```
"execution(* *->@java.lang.Deprecated(..))"
```

No exemplo dado, o *pointcut* casa com todos os métodos que estiverem anotados com `java.-lang.Deprecated`. Veremos a sintaxe de *pointcuts* na Seção 6.1.1.1.

Quando o *pointcut* especifica anotações, o grafo utilizará informações sobre anotações para executar a busca. Assim, se uma classe, método, campo ou construtor passa a ter uma nova anotação em tempo de execução, é preciso atualizar as informações no grafo. Esse é o objetivo do método `update` de `JoinPointGraph`.

Veremos a implementação do grafo em detalhes no Capítulo 6. Por ora, é suficiente saber a quais requisitos ele atende através da sua interface.

5.2. Registro de Sombras

Na Seção 4.1 na página 48, vimos que os objetos `JoinPointInfo` são criados e gerenciados por *advisors*. A criação desses objetos ocorre no momento da construção dos *advisors*, que, por sua vez, ocorre à medida que classes transformadas são carregadas. Quando um *advisor* é criado, ele utiliza informações reflexivas sobre a classe à qual está associado para construir os objetos `JoinPointInfo`. Naturalmente, criar um *advisor* antes que a classe à qual se relaciona seja carregada seria um erro, pois forçaria a carga da classe durante a obtenção dos dados reflexivos.

Pela mesma razão, um requisito importante do nosso algoritmo é que não podemos criar o grafo de sombras de junção de uma única vez. Os dados sobre as sombras só poderão ser adicionados a ele à medida que classes são carregadas no sistema.

Na nossa solução, os objetos `JoinPointInfo` são registrados num objeto `JoinPointRegistry` pelos *advisors*, durante o momento da sua criação. A interface `JoinPointRegistry` representa um registro de sombras que é fornecido pela estratégia `DynamicMatchingStrategy`. Esse registro será responsável por adicionar os objetos `JoinPointInfo` no grafo quando `IndexedMatchingStrategy` for a estratégia utilizada.

5.3. Atualização de Cadeias através de Associações

Um dos princípios de arquitetura orientada a objetos é que uma classe deve manipular os dados que possui sempre que possível, ao invés de servir apenas como um repositório de dados para as outras classes[1]. Na arquitetura original do JBoss AOP, nos deparamos com um cenário onde esse princípio não é seguido. Esse cenário, como veremos, se provou um empecilho para a implementação do nosso algoritmo de maneira clara e simples.

O diagrama da Figura 4.2 na página 53 mostra que `AdviceBinding` contém objetos `Interceptor-Factory`, responsáveis por criar interceptadores. Apesar de estarem contidas em `AdviceBinding`, essas fábricas são utilizadas pela classe `Advisor` e suas subclasses para adicionar interceptadores às cadeias de adendos das sombras. Isso é feito através de um método da classe `Advisor`:

```
void pointcutResolved(JoinPointInfo info, AdviceBinding binding,
 Joinpoint joinpoint)
```

Esse método não contém operações que envolvem dados contidos no `Advisor`. Ele é sobrescrito por suas subclasses que, além de delegarem a criação para as fábricas, aplicam regras adicionais à criação dos interceptadores. Um exemplo de regra adicional é criar um `CFlowInterceptor` quando houver uma expressão `cflow`. Todas as regras adicionais contidas nessas classes são dependentes do modo de instrumentação e da presença ou ausência de `cflow`. Manter essa lógica na hierarquia de `Advisor` é uma decisão que sobrecarrega classes que já possuem dezenas de métodos e realizam uma série de operações. As fábricas pertencem a `AdviceBinding`, que poderia ser responsável por delegar a essas fábricas a criação de interceptadores.

[1] Esse princípio é afirmado no padrão *Information Expert*, que faz parte da série de padrões GRASP (*General Responsibility Assignment Software Pattern*) apresentados em [53].

O nosso algoritmo de atualização de cadeias, representado pela estratégia `IndexedMatchingStrategy`, também deve invocar as fábricas de interceptadores, adicionando interceptadores às cadeias de sombras afetadas por uma adição dinâmica de associação. No início desse capítulo, fizemos uma descrição desse algoritmo. Não mencionamos os *advisors* nessa descrição, pois o nosso algoritmo consiste justamente em mover o controle das sombras para uma estrutura onde sombras são encontradas independentemente do *advisor* a elas associado. Porém, a criação de interceptadores e a sua adição a cadeias de sombras está no método `pointcutResolved` de `Advisor` e, com isso, temos três opções para implementar o nosso algoritmo:

- copiar para `IndexedMatchingStrategy` o código de criação de interceptadores, contido na classe `Advisor` e nas suas subclasses;
- obter o `Advisor` associado a cada sombra, invocando o método `pointcutResolved` em cada um deles para adicionar os interceptadores à cadeia da sombra;
- ou refatorar o código, movendo a invocação das fábricas de interceptadores para a classe `AdviceBinding`.

A primeira solução se opõe aos bons princípios de desenho, facilita a inserção de erros no código bem como dificulta a sua manutenção. Por esses motivos, ela foi descartada. Invocar o método `pointcutResolved` em `Advisor` é uma solução possível, porém ela representa uma complicação desnecessária que estivemos querendo evitar até agora: fazer com que `IndexedMatchingStrategy` dependesse de `advisors` para atualizar cadeias. Nesse caso, dado um objeto `AdviceBinding` que será adicionado dinamicamente, precisamos obter o *advisor* de cada sombra, para invocar o método `pointcutResolved`, cuja implementação não envolve a manipulação de dados do *advisor*, e sim de dados de `AdviceBinding` e da cadeia contida em `JoinPointInfo`. Fica claro, então, que a terceira opção é mais plausível, porque simplifica a implementação do nosso algoritmo.

Assim, decidimos mover a lógica da criação de interceptadores, contida no método `pointcutResolved` de `Advisor`, para a classe `AdviceBinding`. Contudo, simplesmente mover o método `pointcutResolved` de uma classe para a outra se mostrou uma operação complicada, pois seria preciso adicionar à classe `AdviceBinding` as lógicas adicionais aplicadas à criação de interceptadores pelas diversas classes da hierarquia `Advisor`. Apesar dessa dificuldade, insistimos que a invocação de fábricas de interceptadores estava localizada nas classes erradas e criamos a classe `InterceptorChainFactory`. Essa é uma classe abstrata que contém fábricas de interceptadores (`InterceptorFactory`) e delega para essas fábricas a criação dos interceptadores. As operações adicionais que devem ser executadas nesse processo são definidas pelas subclasses de `InterceptorChainFactory`. O resultado da refatoração é ilustrado no diagrama da Figura 5.2.

Ao invés de conter uma coleção de fábricas de interceptadores, `AdviceBinding` passou a conter uma instância de `InterceptorChainFactory`. O método `pointcutResolved`, que agora faz parte de `AdviceBinding`, delega a criação dos adendos para essa fábrica. Os interceptadores que ela cria são adicionados à cadeia de interceptadores de uma sombra. Além disso, `AdviceBinding` armazena os interceptadores criados a um mapa, utilizando a sombra como chave.

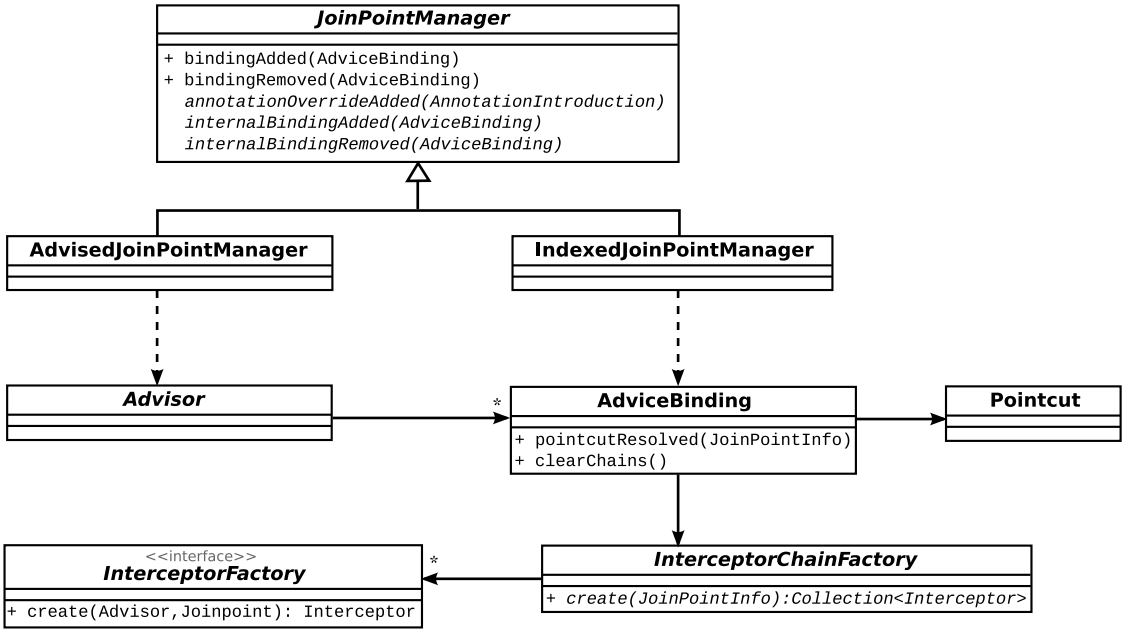


Figura 5.2.: Resultado da refatoração realizada na criação de interceptadores. O método `pointcutResolved` de `AdviceBinding` delega a criação desses interceptadores para `InterceptorChainFactory` e os adiciona à cadeia contida em um objeto `JoinPointInfo`. Dados sobre essa operação são armazenados no `AdviceBinding`, para serem utilizados durante a execução de `clearChains`.

Durante a remoção de uma associação, o mapa de interceptadores criados possibilita a remoção de associações sem a necessidade de reconstruir cadeias. Sem esse mapa, o mesmo não seria possível, já que a criação realizada pelas fábricas é totalmente transparente (não sabemos quantos interceptadores foram criados nem os seus tipos; portanto, não temos condições de utilizar as fábricas para identificar esses interceptadores). Adicionamos à classe `AdviceBinding` o método `clearChains`, invocado durante a remoção de uma associação. Esse método utiliza o mapa de interceptadores para remover todos os interceptadores criados das cadeias que os contêm. Essa solução para a atualização de cadeias após uma operação de remoção não seria tão simples e direta sem a refatoração que realizamos.

5.4. Gerenciamento de Sombras

Outra mudança que fizemos no JBoss AOP para implementar a nossa solução é a adição da classe `JoinPointManager`. Para cada domínio existe um objeto `JoinPointManager`, responsável por gerenciar as sombras do sistema após a criação dos objetos que as representam, `JoinPointInfo`. Quando operações dinâmicas são realizadas, esse objeto deve atualizar as cadeias associadas a essas sombras, o que é realizado através dos métodos `bindingAdded(AdviceBinding)` e `bindingRemoved(AdviceBinding)`. Além desses métodos, `JoinPointManager` possui um método para a notificação da adição de introduções

de anotações, `annotationOverrideAdded`.

Existem duas subclasses de `JoinPointManager`. A primeira é `AdvisedJoinPointManager` e delega a execução de `bindingAdded` para o método `updateAdvisorsForAddedBinding` de `AspectManager`, que atualizará as cadeias conforme explicado na Seção 4.2.2[2]. Quando uma associação é removida, o método `bindingRemoved` é invocado e `AdvisedJoinPointManager` itera por todos os *advisors* do domínio ao qual está associado, chamando o método `rebuildChains` em cada um deles. Essa classe ignora notificações feitas através do método `annotationOverrideAdded`, pois não há necessidade de atualizar informações quando isso ocorre.

A outra subclasse de `JoinPointManager`, `IndexedJoinPointManager`, atualiza as cadeias de adendos conforme o algoritmo que apresentamos nas últimas páginas. Sempre que uma associação é adicionada, uma busca no grafo é feita de modo a encontrar as sombras que casam com o *pointcut* contido na associação adicionada. Para cada uma dessas sombras, criamos interceptadores e adicionamos esses interceptadores à sua cadeia, através de chamadas ao método `pointcutResolved` da classe `AdviceBinding`. Quando uma associação é removida, basta invocar o método `clearChains` na associação removida. Essas operações são ilustradas na Figura 5.3 na página seguinte. Finalmente, quando `IndexedJoinPointManager` é notificado de uma adição de introdução de anotações, ele invoca o método `update` de `JoinPointGraph`, para atualizar as informações contidas no grafo.

5.5. Solução Final

Um diagrama da solução final foi apresentado na Figura 5.1 na página 62. Esse diagrama ilustra os dois tipos de `DynamicMatchingStrategy`: `AdvisedMatchingStrategy` e `IndexedMatchingStrategy`. Cada um desses é responsável por habilitar uma das soluções disponíveis para a atualização de cadeias. A classe `AdvisedMatchingStrategy` habilita o algoritmo original do JBoss AOP, descrito no capítulo anterior. Para isso, ela fornece:

- um `JoinPointRegistry` fantoche, que ignora as chamadas de registro invocadas pelos *advisors* para registrar objetos `JoinPointInfo`;
- uma instância de `AdvisedJoinPointManager` para `AspectManager`;
- uma instância de `AdvisedJoinPointManager` para cada domínio.

A estratégia `IndexedMatchingStrategy`, por sua vez, tem a função de habilitar o algoritmo que descrevemos nesse capítulo. Para isso, essa fornece:

- uma instância de `JoinPointRegistry` que irá registrar todas as sombras `JoinPointInfo` no grafo de sombras;

[2]Uma vez que a classe `JoinPointManager` representa um novo nível de indireção no gerenciamento dessas sombras, poderíamos mover o método `updateAdvisorsForAddedBinding` de `AspectManager` para `JoinPointManager`. Nesse caso, `AdvisedJoinPointManager` não precisaria invocar um método de `AspectManager` para atualizar as cadeias de sombras. Porém, o método `pointcutResolved` é utilizado em outras classes do JBoss AOP e do servidor de aplicações JBoss AS. O mesmo é válido para a coleção de *advisors*, que não é de acesso privado. Para manter a compatibilidade, não refatoramos essa parte do código.

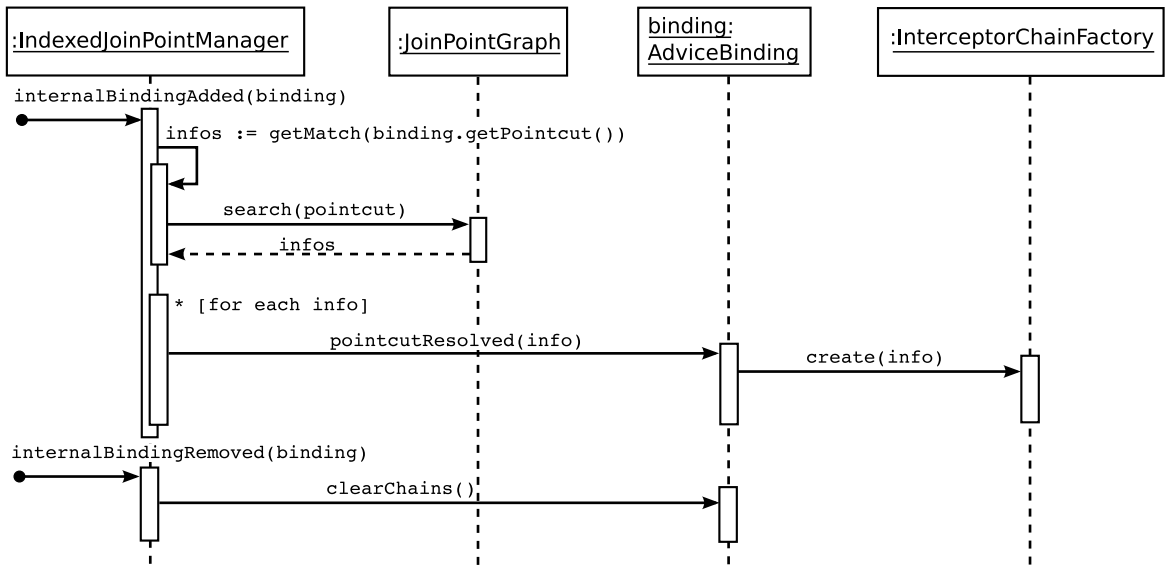


Figura 5.3.: Diagrama de seqüência ilustrando a execução dos métodos `bindingAdded` e `bindingRemoved` da classe `IndexedJoinPointManager`.

- uma instância de `IndexedJoinPointManager` para `AspectManager`, domínio principal;
- uma instância de `IndexedJoinPointManager` para cada domínio.

Essa estratégia se baseia nas funcionalidades do grafo. O diagrama da Figura 5.1 mostra a interface `JoinPointGraph`, presente no pacote `org.jboss.aop.joinpoint.graph`. Para obter um grafo de sombras, `IndexedJoinPointManager` precisa invocar métodos da fábrica `JoinPointGraphFactory`. Os seguintes métodos compõe a interface dessa fábrica:

`JoinPointRegistry getRegistry()`*:* fornece o registro que irá adicionar as sombras ao grafo;

`JoinPointGraph create()`*:* devolve o grafo principal . Esse grafo contém informações sobre todas as sombras registradas até o momento;

`JoinPointGraph create(Domain domain)`*:* devolve um grafo cuja busca se limita às sombras de um domínio.

Uma vez que apresentamos a classe `JoinPointGraphFactory` e os métodos que fornece, é possível concluir o que ocorre quando `IndexedMatchingStrategy` cria instâncias de `IndexedJoinPointManager`. O diagrama da Figura 5.4 na página seguinte ilustra esse processo.

Quando a instância única de `AspectManager` é criada e inicializada, ela pede uma instância de `JoinPointManager` à estratégia `IndexedJoinPointStrategy`. Essa estratégia cria `IndexedJoinPointManager`, passando o objeto `AspectManager` como argumento para o seu construtor. Durante a sua criação, `IndexedJoinPointManager` obtém o grafo correspondente ao domínio principal, através do método `create()` de `JoinPointGraphFactory`.

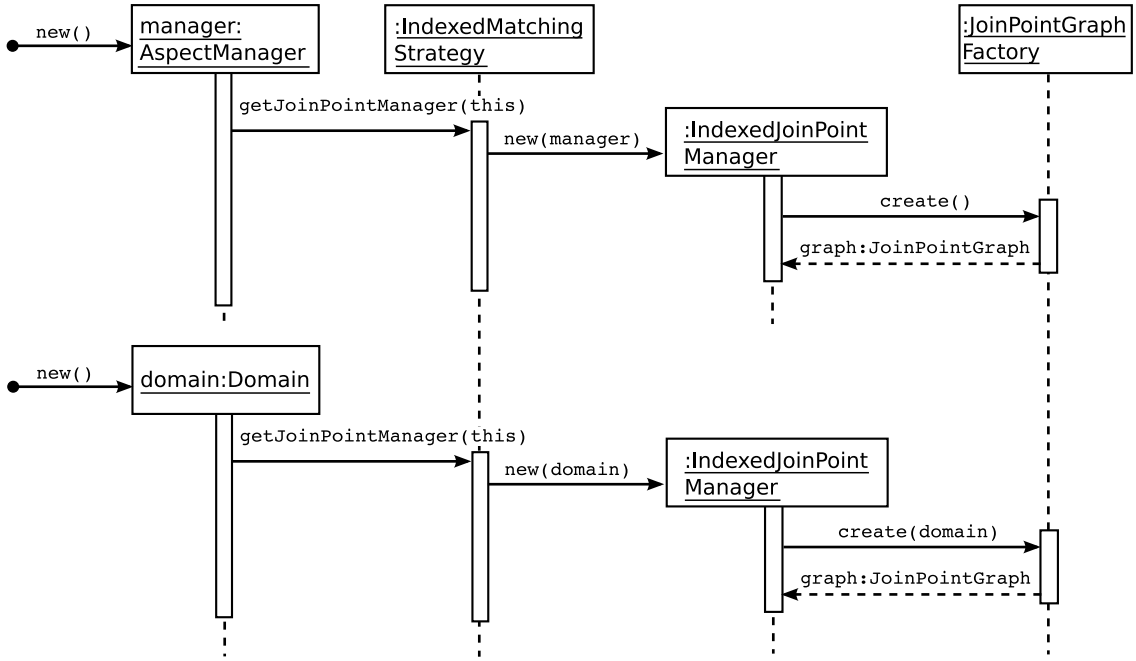


Figura 5.4.: Processo de criação de `IndexedJoinPointManager`, durante o qual instâncias do grafo são obtidas através de `JoinPointGraphFactory`.

Quando um domínio qualquer é inicializado, ele também requisita uma instância de `JoinPointManager` à estratégia. Dessa vez, `IndexedJoinPointStrategy` cria um objeto `IndexedJoinPointManager` passando o domínio como argumento para o construtor. Esse construtor, então, obtém o grafo equivalente àquele domínio invocando o método `create(Domain)` de `JoinPointGraphFactory`.

A interface `JoinPointGraph` e a classe `JoinPointGraphFactory` são os únicos tipos do pacote `org.jboss.aop.joinpoint.graph` com acesso público. As classes que implementam as funcionalidades do grafo, pertencentes ao mesmo pacote, possuem acesso *package-protected*. Essas classes e o algoritmo que implementam é o assunto do próximo capítulo.

6. Grafo de Sombras de Junção

O grafo de sombras é uma estrutura de dados que armazena todas as informações relevantes sobre as classes carregadas e as suas sombras. Dado um *pointcut*, ele executa uma busca e devolve as sombras de junção que casam com o *pointcut*. Vimos no Capítulo 5 que o grafo é utilizado durante a adição dinâmica de associações. Quando isso ocorre, o `JoinPointManager` faz uma busca no grafo para obter as sombras cujas cadeias serão afetadas pela operação.

Esse capítulo apresentará o grafo de sombras em detalhes. Na Seção 6.1 iremos mostrar uma visão geral do grafo, descrevendo o problema que ele se propõe a resolver e a abordagem que ele utiliza para resolvê-lo. Na Seção 6.2, o principal componente da estrutura do grafo, uma árvore, terá a sua estrutura descrita e formalizada. A seguir, na Seção 6.3, veremos como a árvore é utilizada para compor o grafo e, na Seção 6.4, como é realizada a busca no mesmo. Por fim, o suporte a domínios é o assunto da Seção 6.5.

6.1. Visão Geral

O objetivo dessa seção é prover uma visão geral da implementação do grafo de sombras de junção. A seguir, o problema que propomos resolver será formalizado. Depois, apresentaremos a solução proposta na Seção 6.1.2.

6.1.1. O Problema

Queremos substituir o casamento de *pointcuts* com sombras, realizado durante a combinação dinâmica, por uma busca de sombras. Assim, dada uma expressão *pointcut*, esse algoritmo deve encontrar as sombras de junção que precisam ter as suas cadeias de adendos atualizadas após uma operação dinâmica. No Capítulo 4, vimos que as sombras são representadas por objetos `JoinPointInfo`, gerenciados por *advisors*. Os *advisors*, por sua vez, estão contidos em domínios. Essa estrutura é compatível com o algoritmo de combinação dinâmica descrito na Seção 4.2.2. Uma vez que uma operação dinâmica é invocada no domínio, ele delega a combinação dinâmica para *advisors* que, por sua vez, reconstruirão as cadeia de adendos contidas em objetos `JoinPointInfo`. Porém, queremos ser capazes de encontrar esses objetos `JoinPointInfo` através de um acesso indexado numa estrutura de dados, sem ter que, para isso, encontrar os *advisors* que gerenciam esses objetos. De posse desses objetos, podemos atualizar a cadeia de adendos que eles contêm, adicionando ou removendo interceptadores.

Dada uma sombra s, e um *pointcut* p, denotamos por $s \sqsubset p$ a relação de casamento positivo entre elas. Definimos também o conjunto dos objetos `JoinPointInfo` pertencentes a um domínio d como

S_d. Utilizando essa notação, o problema que o algoritmo de busca de sombras de junção se propõe a resolver é definido a seguir:

Definição Dado o domínio d e o *pointcut* estático p, o algoritmo de busca de sombras deve devolver o conjunto $S_{d \cdot p}$, definido como:

$$S_{d \cdot p} = \{s \in S_d \mid s \sqsubset p\} \tag{6.1.1}$$

Está claro que a busca realizada por esse algoritmo se baseia num *pointcut*. Isso não só é determinante para a implementação dessa busca, como também é determinante para o modo como a estrutura de dados é formada. É preciso armazenar nessa estrutura informações relevantes às expressões *pointcut* estáticas. Por esse motivo, consideramos que a sintaxe dessas expressões é de fundamental importância para entender a quais requisitos esse algoritmo deve atender.

6.1.1.1. Linguagem Pointcut

Sabemos que expressões *pointcut* são utilizadas para determinar em quais pontos de junção do sistema base adendos devem ser invocados. No JBoss AOP, todos *pointcuts* são estáticos, exceto expressões do tipo `cflow`. Por motivos apresentados no Capítulo 3, o nosso algoritmo trata de sombras apenas e não envolve *pointcuts* dinâmicos, cujo casamento incorre em comparações feitas em tempo de execução.

Nesta seção mostraremos as construções da linguagem *pointcut* do JBoss AOP. Nos restringiremos às construções disponíveis para *pointcuts* estáticos, que são o alvo do algoritmo de busca de sombras.

A sintaxe das estruturas dessa linguagem será descrita por meio de expressões regulares. A fim de facilitar sua leitura, agrupamos as expressões comuns a mais de uma construção em variáveis, que podem ser vistas na tabela 6.1 na próxima página.

Na primeira linha da tabela podemos notar a principal característica dessa linguagem: os curingas, denotados pelo caractere `'*'`. A estrutura *`class`*, utilizada para casar nomes de classes, pode conter um ou mais curingas em sua definição, formando padrões que podem ser casados em seqüência[1]. Veja os exemplos abaixo:

```
"my.package*al.MyClass"
"*package*"
"my*package1.*package2*package3*Class"
"my.package.InternalClass"
```

Na primeira linha, temos uma expressão que começa com `"my.package"`, seguida de curinga, seguida de `"al.MyClass"`. Os nomes das classes que casam com essa expressão devem começar com `"my.package"`, e terminar com `"al.MyClass"`. Exemplos de nomes de classes que casam com essa expressão são `"my.package.internal.MyClass"`, `"my.packageal.MyClass"` e `"my.-packageInternal.MyClass"`.

[1] Por questões de legibilidade, a expressão regular *`class`* é definida utilizando-se o conjunto de caracteres alfanuméricos. Essa expressão, portanto, ignora regras importantes da linguagem Java como restrições de caracteres no começo de nomes de classes e a presença válida de outros caracteres no nome de classes (`'$'` por exemplo). A mesma simplificação foi feita nas outras expressões regulares da tabela 6.1.

Nome	Expressão Regular
class	(\w?([.*]\w)*)|*
package	\w?([.*]\w)*\.\.
field	((\w?(*\w)*)|[*])|@*class*
constructor	(new)|@*class*
method	((\w?([*]\w)*)|[*])|@*class*
type	*class*|@*class*|*package*|$instanceof\(*type*\)\|$typedef\(\w\)
primitive	byte|short|int|long|float|double|boolean|char
attributes	(!?(public|protected|private|static|synchronized|volatile|transient))*
return-type	*class*|*primitive*|*
field-type	*class*|*primitive*|*
arg	*class*|*primitive*|*|\.\.
arg-list	(*arg*(, *arg*)*)?
throws-list	(throws *class*(, *class*)*)?
field-exp	*attributes field-type type->field*
constructor-exp	*attributes type->constructor*\(*param-list*\) *throws-list*
method-exp	*attributes return-type type->method*\(*arg-list*\) *throws-list*

Tabela 6.1.: Componentes comuns às construções da linguagem *pointcut*. Na primeira coluna vemos o nome dos componentes; na segunda, sua estrutura é definida em expressões regulares. Palavras em itálico nessa última indicam referência a uma outra expressão da mesma tabela.

E, como exemplos de nomes que não casam, temos `"your.-package.internal.MyClass"`, `"com.my.packageal.MyClass"` e `"com.mypackage.MyInter-face"`.

A segunda expressão, `"*package*"`, começa e termina com curinga, indicando que as classes procuradas devem conter o padrão `"package"` em seu nome. Portanto, todos os exemplos dados no parágrafo anterior casam com essa expressão, mas `"com.myPackage.AnyClass"` não casa.

Já a terceira expressão, `my*package1.*package2*package3*Class"`, é mais complexa. Ela determina que o nome da classe deve começar com `"my"` e terminar com `"Class"`. Além disso, seu nome deve conter, nessa ordem, os seguintes padrões `"package1."`, `"package2"` e `"package3"`. Exemplos de nomes de classes que casam com essa expressão são `"mypackage1.package2.package3.Class"` e `"my.otherpackage1.internalpackage245.modelpack-age3000.SomeClass"`.

Finalmente, a última expressão de classe que mostramos, `"my.package.InternalClass"`, não contém curingas e pode, portanto, casar apenas com uma classe de mesmo nome.

A expressão *class* é o bloco principal de *type*, expressão utilizada para definir tipos. Vemos na tabela que uma expressão *type* pode ser de 5 formas diferentes:

- uma simples expressão *class*, como as que vimos acima;
- uma expressão do tipo anotação, formada por `'@'` seguido por uma expressão do tipo *class*. Essa expressão casa com anotações, utilizando a parte *class* para selecionar os tipos das anotações. Quando uma expressão *type* é desse tipo, ela casa com todas as classes que têm as anotações

selecionadas. Por exemplo, `"@javax.persistence.Entity"` casa com todas as classes anotadas com `@javax.persistence.Entity`.

- uma expressão *`package`*, composta por uma ou mais palavras separadas por curingas e pontos (‘`.`’), seguidas pelo sufixo `".."`. As expressões `"java.util.."` e `"java.io.."` seguem essa forma e casam com todas as classes pertencentes aos pacotes `java.util` e `java.io`, respectivamente[2].

- um expressão do tipo *instanceof*. Esse tipo de expressão seleciona todas as classes e subclasses (diretas e indiretas) das classes identificadas pela expressão *`type`* que ela contém. Exemplos válidos são: `"$instanceof(java.util.Collection)"` (casa com a classe `Collection` e todas as suas subclasses) e `"$instanceof(@javax.persistence.Entity)"` (casa com todas as classes anotadas com `javax.persistence.Entity` e suas subclasses).

- uma expressão do tipo *typedef*. Essa expressão recebe o nome de uma definição de tipos, `typedef`, definida no arquivo jboss-aop.xml, ou numa anotação `org.jboss.aop.TypeDef`. Veremos a forma das expressões `typedef` mais adiante.

Também na tabela 6.1, vemos que as expressões *`method`* e *`field`* podem ser de duas formas distintas. A primeira delas (`(\w?(\*\w)*)|[*]`) é similar à expressão *`class`*, com a diferença de que não possui o caractere ‘`.`’, necessário em `class` para separar os nomes dos pacotes. Exemplos de expressões que identificam nomes de métodos e de campos válidas são: `"method"`, `"method*"`, `"count*"`, `"*list"`, etc. O curinga funciona nessas expressões da mesma forma que em *`class`*. Outra alternativa é casar métodos e campos por anotações ao invés de nomes, seguindo o formato *`@class`*. Por exemplo, a expressão `"@java.lang.Deprecated"` é válida como identificador de métodos e campos, e casa com métodos e campos que possuem a anotação `@java.lang.Deprecated`. Anotações podem ser usadas também para casar construtores da mesma forma. A tabela na página anterior mostra que, além de anotações, é possível utilizar a palavra `"new"` para identificar construtores.

Essas expressões, somadas a tantas outras, são utilizadas para formar as estruturas de *`method-exp`*, *`constructor-exp`* e *`field-exp`*, que casam com métodos, construtores e campos. Vejamos alguns exemplos de cada uma delas abaixo:

```
"public int package.MyClass->doSomething(int) \\
                                  throws java.io.IOException"
"$instanceof(package.Interface)->new(long,java.lang.String)"
"private !static double *->amount"
```

O primeiro exemplo não contém curingas e, portanto, casa apenas com um método específico, `public int doSomething(int) throws java.io.IOException`, da classe `package.MyClass`.

[2]Observe que o sufixo `".."` utilizado nessas expressões não é equivalente ao curinga ‘`*`’. A expressão `"java.util.*`, por exemplo, casa não só com as classes do pacote `java.util` como também com as classes dos subpacotes `java.util.concurrent` e `java.util.jar`, dentre outros. O mesmo não se aplica à expressão `"java.util.."`, restrita somente às classes do pacote `java.util`.

A segunda linha declara uma expressão que casa com construtores, já que possui a palavra chave `new`. Os construtores selecionados devem pertencer às classes que implementam a interface `package.Interface` e receber dois valores como argumentos: um do tipo `long` e outro do tipo `String`.

A última expressão identifica um campo com acesso privado não-estático, do tipo `double`, cujo nome é `amount`. Como já vimos, o curinga no lugar do nome da classe casa com todas as classes disponíveis no sistema base.

Veja agora as expressões a seguir:

```
"org.project.Pojo->new(..)"
"org.project.Pojo->new(*)"
"org.project.Pojo->new(.., *)"
```

Todas casam com construtores de uma classe denominada `org.project.Pojo`, cujas assinaturas podem ser vistas na listagem abaixo. Ao contrário das expressões que vimos até aqui, essas expressões utilizam curingas na lista de argumentos. Na primeira linha, vemos o curinga `..`, válido somente para argumentos. Esse curinga casa com qualquer número e tipo de argumentos. Assim, a primeira linha casa com todos os construtores da classe `org.project.Pojo`. A próxima linha utiliza o curinga `'*'`, que casa com um único argumento, de qualquer tipo. Portanto, essa expressão casa apenas com o construtor `Pojo(int)`. A última expressão, por fim, casa com os construtores que possuem um ou mais argumentos, `Pojo(int)` e `Pojo(String,String)`.

```
1  package org.project;
2
3  public class Pojo
4  {
5      public Pojo()
6      {
7          ...
8      }
9
10     public Pojo(int arg)
11     {
12         ...
13     }
14
15     public Pojo(String arg, String arg2)
16     {
17         ...
18     }
19 }
```

Listagem 6.1: Exemplo de classe Java.

As estruturas *field-exp*, *constructor-exp* e *method-exp* são os blocos principais da linguagem *pointcut* do JBoss AOP. As construções que os utilizam, permitindo o casamento com todo o tipo de sombra de junção, estão na tabela 6.2 abaixo. Chamamos as construções da tabela abaixo de **construções primitivas**.

Construção	Ponto de Junção	Exemplo
execution\(*method-exp*\)	execução de métodos	"execution(* *->*(..))"
execution\(*constructor-exp*\)	execução de construtores	"execution(*->new(..))"
call\(*method-exp*\)	chamada de métodos	"call(* *->*(..))"
call\(*constructor-exp*\)	chamada de construtores	"call(*->new(..))"
get\(*field-exp*\)	leitura de campos	"get(* *->*)"
set\(*field-exp*\)	escrita de campos	"set(* *->*)"
field\(*field-exp*\)	leitura e escrita de campos	"field(* *->*)"
all\(*type*\)	execução de todos os métodos e construtores, leitura e escrita de todos os campos	"all(*)"

Tabela 6.2.: Construções primitivas da linguagem *pointcut* do JBoss AOP. A primeira coluna mostra a estrutura sintática de cada construção e a segunda, o significado semântico da mesma. Note que a primeira coluna referencia expressões definidas na tabela 6.1. A terceira coluna ilustra cada construção com um exemplo.

Com a conjunção AND, podemos adicionar restrições a essas construções. As **construções restritivas** encontram-se na tabela 6.3 na página seguinte. As duas primeiras restrições aplicam-se a *pointcuts* do tipo chamada, enquanto que as duas últimas aplicam-se a qualquer tipo de expressão *pointcut*.

Como exemplos de utilização das restrições within e withincode temos as expressões:

```
"call(public int java.util.Collection->size()) \\
                              AND within(com.mypackage.Pojo)"
"call(public int java.util.Collection->size()) \\
         AND withincode( * com.mypackage.Pojo->someMethod())"
```

A primeira expressão casa somente com as chamadas ao método int Collection.size() realizadas dentro da classe com.mypackage.Pojo. A segunda expressão restringe ainda mais as chamadas que podem ser casadas, ao definir que as chamadas só podem ocorrer dentro do método "someMethod" da mesma classe.

Além das restrições within e withincode, temos as construções has e hasfield, que adicionam restrições às classes selecionadas por construções primitivas:

```
"call(public int *->size()) AND has(public * *->iterator())"
"execution(public $instanceof(java.util.Collection)->new(int)) \\
                             AND hasfield( * *->values)"
```

Construção	Semântica / Exemplo	
within\(*type*\)	*Semântica:*	chamadas devem ser feitas dentro da(s) classe(s) que casam com ***type***.
	Exemplo:	`"within(my.package.Pojo)"`
withincode\(*method-exp*\)	*Semântica:*	chamadas devem ser feitas dentro do(s) método(s) que casam com ***method-exp***.
	Exemplo:	`"withincode(int Pojo->doCalls())"`
withincode\(*constructor-exp*\)	*Semântica:*	chamadas devem ser feitas dentro do(s) construtores(s) que casam com ***constructor-exp***.
	Exemplo:	`"withincode(private Pojo->new(..))"`
has\(*method-exp*\)	*Semântica:*	classe deve ter um ou mais métodos que casam com ***method-exp***.
	Exemplo:	`"has(public void *->notify(..))"`
has\(*constructor-exp*\)	*Semântica:*	classe deve ter um ou mais construtores que casam com ***constructor-exp***.
	Exemplo:	`"has(*->new(long, boolean, \\ java.util.Collection))"`
hasfield\(*field-exp*\)	*Semântica:*	classe deve ter ter um ou mais campos que casam com ***field-exp***.
	Exemplo:	`"hasfield(!private \\ my.package.Pojo *->pojoField)"`

Tabela 6.3.: Construções restritivas da linguagem *pointcut* do JBoss AOP. Na primeira coluna, é possível ver a sintaxe dessas construções e, na segunda, encontramos seu significado semântico e exemplos. A sintaxe das expressões ***type***, ***method-exp***, ***constructor-exp*** e ***field-exp*** foi definida na tabela 6.1 na página 72.

O primeiro exemplo acima casa com as chamadas aos métodos `public int size()` das classes que possuem um método chamado `iterator()`. O segundo exemplo, com a execução dos construtores que recebem um valor `int` como argumento, de todas as subclasses da classe `Collection` que têm um campo chamado `values`.

Além das restrições, é possível formar outros tipos de ***pointcuts* compostos**, utilizando-se as conjunções `AND` e `OR`. A primeira tem função de intersecção e a segunda, de união. É possível também utilizar parênteses para definir prioridade. A expressão a seguir casa com todos os pontos de junção selecionados por uma das expressões vistas acima:

```
"(call(public int *->size()) AND has(public * *->iterator())) OR \\"
"(execution(public $instanceof(java.util.Collection)->new(int)) \\
                          AND hasfield( * *->values))"
```

Também é possível compor *pointcuts* com o operador de negação `!`:

```
"!call(public int *->size())"
```

Esse exemplo mostra um *pointcut* que casa com todas as sombras que não forem uma chamada a um método `public int size()`.

Pointcuts que não são compostos, cuja forma segue uma das construções da tabela 6.2, são denominados ***pointcuts* simples**.

Definições de Tipo No JBoss AOP, é possível criar uma definição de tipo através da expressão `typedef`, para referenciá-la mais tarde dentro de *pointcuts*.

Uma expressão `typedef` pode seguir a estrutura da expressão regular `class\(`*`type`*`\)`, onde *`type`* se refere à expressão regular definida na tabela 6.1 na página 72. Exemplos de expressões `typedef` válidas seguindo essa estrutura são `"class(com.company.package.AnyClass)"` e `"class($instanceof(package.Pojo))"`.

A vantagem das expressões `typedef`, entretanto, não está nessa estrutura simples, e sim na possibilidade de composição da mesma. É possível compor expressões *`class\(type\)`* com o uso de `AND`, `OR` e parênteses:

```
"class(com.company.package.AnyClass) OR \\
                                class($instanceof(package.Pojo))"
"class(com.company.package.AnyClass) OR \\
   (class($instanceof(package.Pojo)) AND class(internalPackage.*))"
```

A primeira expressão é a união dos exemplos dados no parágrafo anterior. A segunda possui uma construção `class` adicional no segundo bloco da união. Ela casa com a classe `"com.company.package.AnyClass"`, ou com as subclasses da classe `package.Pojo` que estiverem dentro do pacote `internalPackage` (incluindo subpacotes).

É possível, ainda, adicionar as restrições do tipo `has` e `hasfield`, como abaixo:

```
"class(com.company.package.*) AND has(public int *->anyField)"
```

Para que possam ser referenciadas por *pointcuts*, as definições de tipo são declaradas com um nome. Veja o exemplo abaixo, extraído de um exemplo da documentação do JBoss AOP:

```
<typedef name="TD" expr="(class(POJO) AND has( * *->method(..))) OR \\
    class($instanceof{ExecutionTypedefInterface}) OR \\
    class(@ExecutionTypedef)" />
```

Essa expressão foi declarada num arquivo `jboss-aop.xml` e é composta de três blocos separados pela conjunção `OR`. O primeiro casa com a classe chamada `POJO` se ela tiver um método denominado `method`. O segundo, com as classes que implementam a interface `ExecutionTypedefInterface`. O terceiro, por sua vez, casa com todas as classes que possuírem a anotação `@ExecutionTypedef`. Essa expressão do tipo `typedef` foi declarada com o nome `TD`, e é utilizada nas expressões *pointcut* que se seguem:

```
"execution($typedef{TD}->new())"
"execution( * $typedef{TD}->method())"
"field( * $typedef{TD}->field1)"
"all($typedef{TD})"
```

A primeira expressão seleciona os construtores padrão das classes que casam com a definição de tipo `TD`. A segunda, casa com as execuções dos métodos `method()` das mesmas classes. A terceira, casa com todos os pontos onde o campo `field1`, das classes identificadas por `TD`, tem seu valor lido ou escrito. Finalmente, a última casa com as execuções de todos os métodos e construtores das classes que casam com `TD`, bem como todos as sombras onde seus campos são lidos ou escritos.

6.1.2. Solução Proposta

A solução proposta consiste em criar um grafo que armazena todos os objetos `JoinPointInfo` do domínio principal (`AspectManager.instance()`), utilizando informações relevantes sobre as classes instrumentadas e suas sombras para indexá-los. Dado um *pointcut* p, o algoritmo de busca deve utilizar esse grafo para encontrar todos os elementos do conjunto $S_{d \cdot p}$, onde d é o domínio principal.

No JBoss AOP, as sombras instrumentadas podem ser: execução de métodos e construtores, leituras e escritas de campos, além de chamadas a métodos e construtores. Portanto, o nosso grafo deve ser capaz de armazenar todos esses tipos de sombras. O grafo de sombras é construído utilizando-se como componente principal uma outra estrutura de dados, um tipo específico de árvore prefixa ou *trie* [51]. Como veremos na Seção 6.2, essa árvore armazena objetos identificados por cadeias de caracteres como chaves e suporta chaves de busca que contêm curingas.

A grosso modo, podemos afirmar que o grafo de sombras é uma seqüência de árvores aninhadas. A árvore mais externa contém informações relativas a uma classe. Nessa árvore, inserimos nós que representam classes, utilizando o nome da classe que o nó representa como chave. O nó de classe, por sua vez, contém instâncias do mesmo tipo de árvore, que chamamos de subárvores. As subárvores do nó de classe são utilizadas para armazenar informações sobre campos, métodos e construtores. Da mesma forma que a árvore de classes, as árvores que contêm informações de campos possuem um nó para cada campo, associado ao nome do campo como chave. De forma análoga, métodos e construtores são representados por nós associados ao nome do método ou construtor como chave. Além disso, os nós de classe possuem conexões com outros nós de classe, para indicar a relação de hierarquia entre as classes. Existem também interconexões entre os nós que representam métodos e construtores, para indicar as chamadas realizadas por cada método e construtor.

A estrutura do grafo de sombras é ilustrada na Figura 6.1-A. Nessa imagem, fica clara a estrutura formada por camadas de árvores aninhadas: a mais externa se refere a classes; a seguir temos as árvores de campos, métodos e construtores. Além disso, temos as interconexões entre os nós dessas árvores. Podemos visualizar essa estrutura como sendo um grafo, cujos vértices são os nós das árvores aninhadas, e as arestas são as arestas das mesmas árvores aninhadas e as interconexões entre os nós do grafo. Por esse motivo denominamos essa estrutura de grafo de sombras de junção.

Uma vez que esse grafo é composto principalmente por árvores aninhadas, a busca por sombras é delegada a essas árvores. Para ilustrarmos como isso funciona, considere o *pointcut*:

```
"call(* B->y()) AND  withincode(* A->x())"
```

Essa expressão casa com todas as chamadas ao método `B.y()` feitas pelo método `A.x()`. Para encontrar as sombras que casam com essa expressão, o nosso algoritmo extrai o nome da classe que

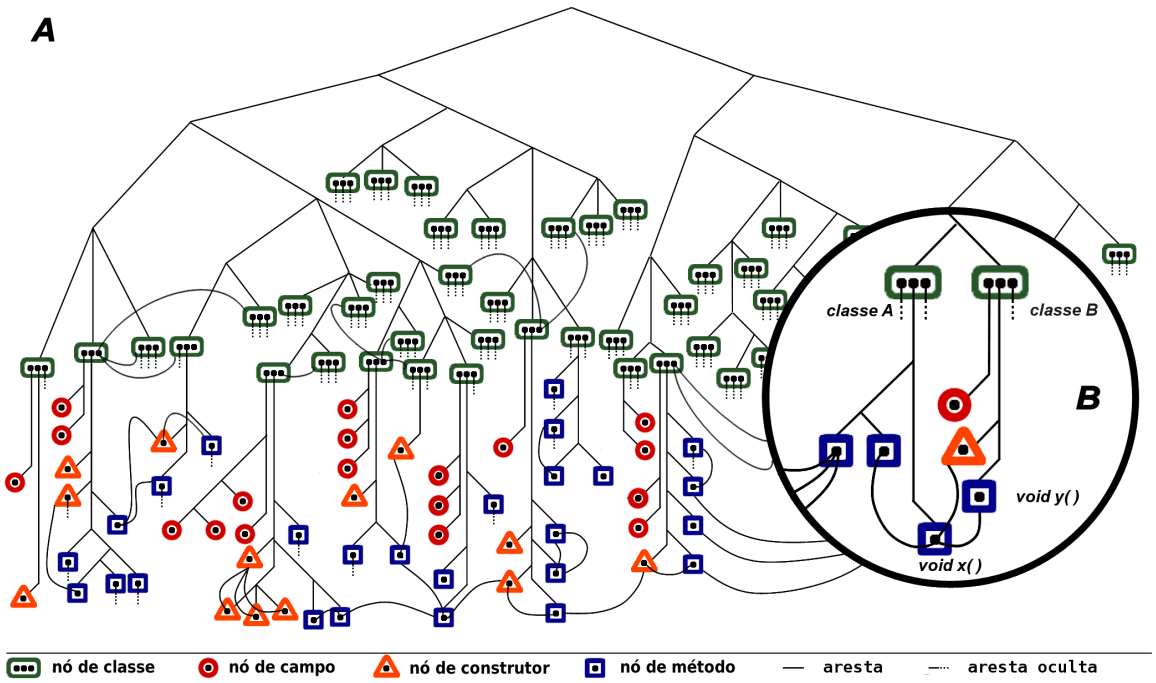


Figura 6.1.: O grafo de sombras de junção. **A-** Esse grafo é composto por uma série de árvores aninhadas. No nível mais externo, existe uma árvore de classes. Cada nó representa uma classe e é composto por duas subárvores, uma para nós de campos, e outra subárvore para inserir nós de construtores e métodos. Além disso, cada nó de classe possui conexão com os nós que representam as subclasses. Note também que os nós de métodos e construtores contém outras subárvores, que representam relações de chamada entre eles. **B-** A ligação do nó do método `A.x()` ao nó do método `B.y()` indica que `x()` faz uma chamada a `y()`. Além disso, podemos ver que `A.x()` faz uma chamada ao único construtor da classe `B`. Além de `A.x()`, a classe `A` possui outros dois métodos: um faz uma chamada ao método `x()`; e outro faz chamadas a três outros métodos da aplicação.

faz a chamada, `"A"`. Esse nome é utilizado como chave de busca na árvore de classes. Uma vez que o nó correspondente é obtido, a busca continua em uma das subárvores contidas no nó da classe `A`. No exemplo, buscaremos na subárvore de métodos e construtores pelo nó que representa o método `x()`. Por fim, é realizada uma busca por `B.y()` na árvore de chamadas contida dentro do nó que representa o método `x()`. As arestas e nós percorridos durante essa busca são ilustrados na Figura 6.1-B.

A busca por outros tipos de sombras, como execução de métodos e construtores, e leituras e escritas de campos, é realizada de forma similar. Em geral, dado um *pointcut* estático simples p, cuja estrutura é uma das construções da tabela 6.2 na página 75, extraímos de p a expressão que define o tipo da classe, ***`type`*** (definida na tabela 6.1 na página 72). Essa expressão é utilizada para fazer uma busca na árvore de classes. Uma vez obtidos um ou mais nós que casam com a expressão ***`type`*** extraída, é feita uma busca pelo componente procurado (seja um campo, método ou construtor) na subárvore apropriada. Para encontrar esse componente, obtemos uma chave de busca a partir de p, de forma similar ao que foi feito no exemplo anterior. Quando chegamos aos nós que representam campos, métodos e construtores, acessamos os objetos `JoinPointInfo` que casam com p conforme especificado por esse *pointcut*.

Para *pointcuts* compostos, é preciso avaliar os componentes envolvidos. Composições de uma construção primitiva com uma ou mais construções de restrição da linguagem, especificadas na tabela 6.3 na página 76, significam uma única busca no grafo. As restrições devem ser aplicadas durante a busca por nós que representam classes. Para ilustrar esse tipo de busca veja o exemplo abaixo:

```
"all(org.*.A) AND  has(* A->x())"
```

No exemplo, a segunda parte do *pointcut* composto é uma restrição que deve ser aplicada à busca por todas as classes que casam com `"org.*.A"`. Veremos mais informações sobre esse tipo de busca na Seção 6.4.

Quando o *pointcut* composto é formado por uma ou mais construções primitivas, é preciso quebrá-lo em duas ou mais partes, e processar essas partes como buscas independentes. Cada parte deve conter uma construção primitiva e zero ou mais construções restritivas. Após realizar as buscas das partes, podemos fazer a intersecção, união ou subtração deles de acordo com o *pointcut* original. Veja os três exemplos abaixo:

```
"execution(* *->*(int)) OR execution(*->new(int))"
"!execution(* *->*(int))"
"execution(* Pojo->(..)) AND !execution(* *->*(int))"
```

O primeiro *pointcut* é quebrado nas partes `execution(* *->*(int))"` e `"execution(*->new(int))"`. Fazemos a busca pelas sombras que casam com cada uma delas de maneira independente e devolvemos como resultado a união das duas buscas.

O segundo *pointcut* é composto pela negação de uma construção primitiva. Ele deve casar com toda sombra que não casar com a construção `execution(* *->*(int))`. Internamente, lemos essa expressão como:

```
"(all(*) OR call(* *->*(..)) OR call(*->new(..))) \\
 AND !execution(* *->*(int))"
```

Ou seja, todas as leituras e escritas de campo, todas as execuções de métodos e construtores, e todas as chamadas a métodos e construtores, menos o *pointcut* que foi negado na expressão original. Novamente, quebramos o *pointcut* em partes e realizamos as buscas de forma independente. Fazemos a união do resultado das buscas por `"all(*)"`, `"call(* *->*(..))"` e `"call(*->new(..))"` e, finalmente, subtraímos desse resultado o resultado da busca por `execution(* *->*(int))`.

O terceiro *pointcut*, `"execution(* Pojo->(..)) AND !execution(* *->*(int))"` contém uma negação mas não precisa ser transformado em outro *pointcut* internamente antes da busca. Nesse exemplo, a negação se aplica explicitamente a um conjunto de sombras de junção, aquele que casa com `"execution(* Pojo->(..))"`. O algoritmo de busca deve, portanto, apenas quebrar esse *pointcut* em duas partes (`"execution(* Pojo->(..))"` e `"execution(* *->*(int))"`) e subtrair o resultado da busca da segunda parte da busca da primeira parte.

Naturalmente, o mecanismo de busca que detalhamos possui generalizações e simplificações. Como vimos na Seção 6.1.1.1, é permitido o uso de curingas nessas expressões, o que indica um importante requisito que deve ser atendido pelas árvores internas do grafo: suporte a busca de chaves com curingas. Além disso, um tipo não necessariamente é uma expressão *`class`*. Ele pode também ser uma expressão *`package`*, `typedef`, `instanceof` ou `@`*`class`*.

Outro requisito que deve ser atendido pelo grafo é o suporte a domínios. A estrutura que apresentamos até o momento representa o domínio principal, mas será preciso suporte aos diversos domínios que podem existir numa aplicação. Esse tópico será abordado mais adiante, na Seção 6.5.

Nas próximas seções descreveremos o algoritmo proposto em detalhes. Daremos início na próxima seção descrevendo a estrutura da árvore de busca, avaliando algumas de suas propriedades e como essas propriedades permitem a busca de chaves com um ou mais curingas. Na seções seguintes, veremos como varias instâncias dessa árvore são estruturadas para compor o grafo de sombras e mais alguns detalhes sobre a busca nesse grafo, além da solução utilizada para prover suporte a domínios.

6.2. A árvore de sombras

Por ser o componente central do grafo de ponto de junção, a árvore é uma peça fundamental para o nosso algoritmo. Essa árvore deve atender, na medida do possível, a dois importantes requisitos:

- alocar um espaço mínimo de memória;
- e prover uma busca eficiente.

Quanto ao primeiro requisito, é importante observar que o volume de informações contidas na árvore pode ser alto no caso de sistemas grandes. No segundo item, falamos em busca eficiente. Com isso, queremos dizer que essa busca deve ser, no mínimo, mais eficiente que o algoritmo de casamento atualmente utilizado, descrito na Seção 3.2.2.1. Idealmente, o algoritmo deve suportar cargas pesadas de informações de maneira escalável, permitindo que sistemas grandes sejam completamente preparados para operações dinâmicas sem, no entanto, sofrerem um impacto considerável em seu desempenho ou na quantidade de memória que ocupam durante a sua execução.

A quantidade de memória alocada por uma árvore e a velocidade da busca nessa árvore se refletirão diretamente no espaço ocupado pelo grafo e seu desempenho, já que essa estrutura é o seu principal componente. Quanto melhor a árvore atender a esses requisitos, mais escalável será a nossa solução, permitindo que o número de sombras preparadas não interfira de forma drástica no desempenho do casamento de sombras. Porém, atender a esses requisitos não é trivial e, muitas vezes, eles podem ser conflitantes[3].

Na Seção 6.3, veremos como essa árvore é composta numa estrutura aninhada para formar o grafo da Figura 6.1. Apesar de ser utilizada para armazenar os diversos tipos de nós (classe, campo, método e construtor), vamos considerar nesta seção apenas os requisitos necessários para montar a árvore de nós de classes. Mais adiante, veremos que a nossa árvore também atende aos requisitos necessários para montar as subárvores do grafo.

Assim, assumimos que a árvore deve conter nós de classes, identificados pelo nome da classe que representam. O nome que identifica um nó de classe é o que chamamos de chave; o nó que representa uma classe é o valor identificado por aquela chave. Sabemos que pode haver mais de uma classe com o mesmo nome, quando há mais de um *class loader* envolvido. Além disso, uma vez que *pointcuts* identificam classes também por anotações, e não apenas por seu nome, queremos inserir nessa árvore os nós de classes utilizando também as anotações dessas classes como chave. De tudo isso, podemos concluir que a árvore pode ter mais de uma chave identificando o mesmo valor (por exemplo, `org.package.OldClasse` e `@java.lang.Deprecated`), além de mais de um valor identificado pela mesma chave (duas classes com o mesmo nome carregadas por *class loaders diferentes* ou, ainda, duas classes com a mesma anotação). Definimos a árvore de busca interna do grafo de sombras, denominada **árvore de sombras**, a seguir.

Definição Definimos uma instância da árvore de sombras T como sendo a sêxtupla (N, r, E, K, V, δ), onde N é um conjunto de nós, $r \in N$ é a raiz de T, $E \subset N \times N$ é o conjunto de arestas que conecta nós internos aos seus nós filhos, K é o conjunto de chaves inseridas em T, V é o conjunto de valores contidos em T e, $\delta \in K \times V$ é o mapeamento de chave para valor. T possui as seguintes propriedades:

(i) toda chave $k \in K$ identifica um ou mais valores $v \in V$, onde $(k, v) \in \delta$;

(ii) uma chave $k \in K$ é uma palavra denotada por $S[0..|k|-1]$, onde $|k|$ é o comprimento de k, cujos caracteres pertencem ao alfabeto finito Σ, composto pelos caracteres considerados válidos em um nome Java e pelo caractere `'@'`;

(iii) cada nó $n \in N$ contém uma parte $S[i..j]$ de uma chave $k = S[0..|k|[\in K$, onde $0 \leq i \leq j < |k|$. Essa parte é chamada de **subchave**;

[3]Um exemplo simples dessa propriedade pode ser verificado ao compararmos o desempenho da busca numa tabela de dispersão (*hashing*) com o desempenho da mesma operação numa árvore binária balanceada. A busca numa tabela de dispersão apresenta um desempenho melhor em relação à árvore, mas requer mais espaço na memória. A mesma propriedade pode ser encontrada ao analisarmos apenas a tabela de dispersão: quanto mais espaços vazios, melhor será o desempenho da busca (veja [72] para mais detalhes). Outro exemplo pode ser também observado na implementação do casamento de sombras em tempo de execução: a abordagem utilizada originalmente pelo JBoss AOP não requer espaço extra na memória; já o grafo de sombras requer espaço, mas no geral apresenta melhor desempenho, como será mostrado no Capítulo 8.

(iv) o caminho entre n_1 e n_2, onde $n_1, n_2 \in N$, identificado por $path_T(n_1, n_2)$, também denota uma subchave, que é o resultado da concatenação das subchaves contidas em todos os nós ao longo do caminho; se $n_1 = r$ e n_2 é uma folha, então $path_T(n_1, n_2)$ é uma chave $k \in K$;

(v) a raiz r é definida de acordo com o tamanho de K:

- $|K| = 0 \implies r$ é `null`;
- $|K| = 1 \implies r$ é uma folha, cuja subchave é a única chave k contida em K;
- $|K| > 1 \implies r$ é um nó interno cuja subchave é o prefixo comum a todas as chaves inseridas em T.

(vi) cada folha $l \in N$ contém um conjunto de valores V_l, definido como:
$V_l = \{v \in V \mid (path_T(r, l), v) \in \delta\}$;

(vii) cada nó $n \neq r \in N$ é a raiz de uma subárvore $T_n = (N_n,\ n,\ E_n,\ K_n,\ V_n,\ \delta_n)$, onde $N_n \subset N$, $E_n \subset E$, $V_n \subset V$, $K_n = \{k' \mid path_T(r, n') \cdot k' \in K\ ,\ n' \in N$ e $(n', n) \in E\}$ e $\delta_n = \{(k', v)\ | k' \in K_n, v \in V_n,\ (path_T(r, n') \cdot k', v) \in \delta\ ,\ n' \in N, (n', n) \in E\}$.

Uma vez que um caminho na árvore denota uma subchave, nós utilizaremos esses termos de maneira equivalente.

Um exemplo da estrutura da nossa árvore é dada na a Figura 6.2. Apesar de não se tratar de uma árvore balanceada, o desempenho de sua busca não é afetado por esse fator, uma característica comum a árvores prefixas [51].

Definimos o **identificador** ou ***id*** de uma subchave como sendo o seu primeiro caractere. Exemplos de *ids* podem ser vistos na Figura 6.2: o *id* de `"secCompany.project"` é `'s'`; o nó irmão, por sua vez, `"firstCompany.project.com"`, possui o caractere `'f'` como *id*. Os identificadores de subchaves são utilizados por nós internos para identificar os seus filhos.

Nas seções seguintes, procedemos para os algoritmos responsáveis pelas operações de inserção e de busca na árvore de sombras.

6.2.1. Inserção

Uma vez que o grafo de sombras é composto por diversas instâncias de árvores, a semântica das chaves contidas na árvore dependerão do tipo de objetos que a árvore armazena. No caso da árvore de classes, cada chave pode ser o nome de uma classe ou o nome de uma anotação precedido de `'@'`. A expressão regular define a forma geral que a chave pode ter[4]:

$$\texttt{@ ? (}\underline{X}\texttt{+ [.])* }\underline{X}\texttt{+} \tag{6.2.1}$$

Onde $\underline{X}$ é um caractere do alfabeto Σ. Exemplos de chaves válidas são `"org.company.package.Class"` e `"@javax.persistence.Entity"`.

[4]Para simplificar, ignoramos a regra que diz que todo nome Java válido não pode começar com dígitos. Essa informação não afeta o algoritmo proposto.

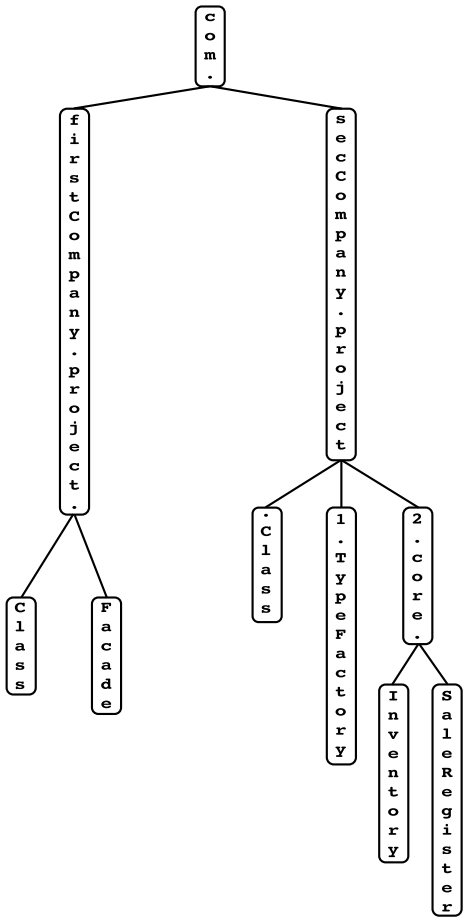


Figura 6.2.: Um exemplo de árvore de sombras. Cada nó possui uma subchave, que é uma parte de uma chave inserida na árvore. Obtemos as chaves da árvore através da concatenação das subchaves encontradas ao longo dos caminhos da raiz a cada uma das folhas.

Dividimos o algoritmo de inserção em duas partes. A primeira é o Algoritmo 1 na próxima página.

Sempre que um valor *value*, identificado por uma chave *key*, precisa ser inserido numa árvore *tree*, esse algoritmo de inserção é executado. O bloco das linhas 1-5 é executado se a raiz for nula, onde inserimos o primeiro par chave-valor na árvore. Caso contrário, o Algoritmo 2 é executado na raiz de *tree*. Esse algoritmo compara cada caractere da chave de inserção, *key*, com cada caractere da subchave contida na raiz da árvore, *node* (linhas 5-8). Quando essa verificação chega ao fim, o próximo passo a ser executado depende dos valores dos índices i e j. Quando $i < n$ e $j = m$ (linha 9), chegamos ao final da subchave contida na raiz da árvore, mas não ao final da chave a ser inserida. Nesse caso, se há um nó filho cujo *id* é $key[i]$, executamos o mesmo algoritmo de forma recursiva, dessa vez especificando esse nó filho como valor do parâmetro *node*. Se não há um nó filho identificado pelo caractere $key[i]$, criamos uma folha com o valor a ser inserido e com subchave $key[i..n-1]$, inserindo essa folha como

Algoritmo 1 INSERT(key, $value$, $tree$)

```
1: if tree.root = NIL then
2:     create leaf root
3:     root.keyPart ← key
4:     root.values ← {value}
5:     tree.root ← root
6: else
7:     tree.root ← INSERT(key, value, tree.root, 0)
8: end if
```

nó filho da raiz, identificada pelo *id* $key[i]$.

O bloco que se inicia na linha 22 é executado quando *node* é uma folha e chegamos tanto ao final de key como da subchave $keyPart$. Esse cenário indica que já havia uma chave igual a key na árvore e estamos apenas inserindo um novo valor associado a essa chave.

O último bloco desse algoritmo, nas linhas 25-34, é executado sempre que não chegamos ao final da subchave $keyPart$, ou quando chegamos ao final de ambas as chaves mas *node* não é uma folha. Quando isso ocorre, dividimos a subchave $keyPart$ em duas subchaves: uma composta pelos caracteres que já foram comparados (prefixo comum a $keyPart$ e a key); outra composta pelo restante dos caracteres de $keyPart$. A primeira subchave será a subchave de um novo nó, $father$, que substituirá *node* na árvore. A segunda subchave, $keyPart[j..-m-1]$, é atribuída para *node* na linha 28. Esse nó passa a ser filho do nó $father$, que o identifica através do *id* $keyPart[j]$. Finalmente, criamos um nó folha $newNode$, que terá $key[i..n-1]$ como subchave e conterá o valor $value$. O algoritmo adiciona $newNode$ à lista de filhos do nó $father$ e devolve $father$ como resultado da inserção. O nó devolvido como resultado da inserção substituirá a posição originalmente ocupada por *node* na árvore.

Observe o que ocorre quando key é prefixo de uma chave key' já presente na árvore. A inserção percorrerá os nós da árvore seguindo o mesmo caminho que representa a chave key'. Esse processo acaba quando o algoritmo chega ao final da chave key que, por ser prefixa de key', chegou ao final antes de compararmos todos os caracteres de key'.

Nesse caso, o algoritmo de inserção irá entrar no bloco das linhas 25-34, quebrando *node* em dois pedaços. A linha 31 do Algoritmo 2 definirá da nova folha como sendo $key[i..n-1]$, que é igual a uma cadeia de caracteres vazia, já que $i = n$. O resultado é uma folha na árvore que contém uma subchave vazia, identificada por $k[n-1]$, o caractere nulo `'\0'`. O cenário resultante é o mesmo se a chave key' for inserida após key. Nesse caso, no momento da inserção de key', o algoritmo percorrerá o caminho de key na árvore até terminar de comparar todos os seus caracteres. Ao executar o bloco `else` da linha 25, o algoritmo irá quebrar a folha que contém os valores associados a key em duas partes: $keyPart[0..j]$ e $keyPart[j..m-1]$. Como chegamos ao final dos caracteres de key, que estava inserida na árvore, $j = m$ e, portanto, a folha passa a ter uma subchave vazia e será identificada pelo caractere nulo.

Algoritmo 2 INSERT(key, $value$, $node$, i)

```
1: j ← 0
2: keyPart ← node.keyPart
3: n ← length[key]
4: m ← length[keyPart]
5: while i < n and j < m and key[i] = keyPart[j] do
6:    i++;
7:    j++;
8: end while
9: if i < n and j = m then
10:   id ← key[i]
11:   child ← node.children[id]
12:   if child = NIL then
13:      create leaf newNode
14:      newNode.keyPart ← key[i..n − 1]
15:      newNode.values ← {value}
16:      node.addChild(id, newNode)
17:   else
18:      newChild ← INSERT(key, value, child, i)
19:      node.addChild(id, newChild)
20:   end if
21:   return node
22: else if node is leaf and j = m then
23:   node.values add value
24:   return node
25: else
26:   create node father
27:   father.keyPart ← keyPart[0..j]
28:   node.keyPart ← keyPart[j..m − 1]
29:   father.addChild(keyPart[j], node)
30:   create leaf newNode
31:   newNode.keyPart ← key[i..n − 1]
32:   newNode.values ← {value}
33:   father.addChild(key[i], newNode)
34:   return father
35: end if
```

Esse mecanismo de suporte a chaves prefixas de outras chaves é importante na árvore de sombras, pois permite o suporte a classes internas como, por exemplo, seria o caso de uma classe chamada `my.package.AnyClass.InternalClass`.

6.2.2. Busca

A particularidade da árvore de sombras em relação a uma árvore *trie* comum está na chave de busca, devido à presença de curingas:

$$\texttt{([*]? @ ? (}\underline{X}\texttt{+ [.*])* }\underline{X}\texttt{+ [*]?) | [*]} \tag{6.2.2}$$

Assim como na expressão regular (6.2.1), o caractere $\underline{X}$ acima representa qualquer caractere do alfabeto Σ.

Apesar de reduzirem o acoplamento entre classes e aspectos [48], os curingas são os grandes responsáveis pela complexidade do algoritmo de busca da árvore de sombras.

Uma chave de busca pode representar diferentes tipos de busca, conforme o número e a posição dos curingas que ela contém. Se ela não contém curingas, ela representa uma busca simples, onde é preciso apenas verificar se há uma chave igual na árvore. Nesse caso, o algoritmo de busca é bem similar ao de inserção, pois percorre a árvore utilizando os caracteres de uma chave. Por se tratar de uma busca, esse algoritmo termina quando uma comparação falha ou quando chegamos ao final de uma folha.

Por outro lado, se a chave de busca contém curingas, existem três algoritmos diferentes a serem utilizados. Veja as chaves de busca abaixo, que foram utilizadas como exemplo de expressões *`class`* na Seção 6.1.1.1.

```
"my.package*al.MyClass"
"*package*"
"my*package1.*package2*package3*Class"
"my.package.InternalClass"
```

Podemos dizer que a primeira expressão especifica que o prefixo de uma chave deve ser igual a `"my.package"` e o seu sufixo, igual a `"al.MyClass"`. Da mesma forma, afirmamos que a segunda expressão casa com chaves que contêm o padrão `package`, idependente de qual o seu prefixo ou sufixo. Quanto à terceira expressão, esta contém o prefixo `my`, os padrões `"package1."`, `package2` e `package3` e o sufixo `"Class"`. Assim, ao buscar por essas expressões na árvore, podemos usar o caractere `'*'` para dividi-las em partes e buscar por cada uma dessas partes, conforme se trata de um prefixo, um padrão ou um sufixo. Chamamos, cada uma dessas partes de **subchaves de busca**. Veja a última expressão da lista. Ela não contém curingas e, portanto, é composta por uma única subchave de busca, que chamamos de subchave simples.

Assim, há quatro tipos de subchaves de busca: prefixo, padrão, sufixo e simples. Para realizar uma busca por uma chave na árvore, atribuímos uma parte da busca a cada subchave de busca, que deve comparar os caracteres contidos nas subchaves da árvore com os próprios caracteres. A forma como essa comparação é realizada depende do tipo de busca que aquela subchave representa.

No caso de uma chave de busca com curingas, a primeira subchave a ser comparada é uma subchave prefixa (para expressões como `"*pattern*"`, consideramos que ela começa e termina com subchaves de busca vazias), que executa uma busca por prefixo na árvore. Uma vez que essa busca termina, a próxima subchave de busca inicia a sua busca no mesmo nó onde a busca prefixa terminou com resultado positivo. Desse modo, o bastão é passado adiante para cada uma das subchaves de busca. Esse processo continua até que a última subchave de busca, uma subchave sufixa, devolva um resultado positivo em uma ou mais folhas, ou até que uma comparação resulte em uma falha.

Uma conseqüência dessa divisão é que precisamos de quatro tipos de algoritmos, uma para cada tipo de subchave: busca simples, busca de prefixo, busca de padrão e busca de sufixo. Assim como a inserção, esses algoritmos comparam caracteres na árvore de forma organizada, comparando a subchave $keyPart$ de um nó n com uma subchave de busca $searchKeyPart$. Sempre que a comparação de $keyPart$ terminar com sucesso, esses algoritmos seguem para os nós filhos de forma apropriada. As buscas simples e de prefixo iniciam a comparação na raiz da árvore. Já a busca de padrão e a busca de sufixo têm início a partir de uma ou mais posições na árvore, posições essas que foram o resultado da busca realizada pela subchave anterior.

O algoritmo de busca simples é mostrado na página seguinte (Algoritmo 3). Esse algoritmo é similar à inserção: cada caractere da subchave contida em *node* é comparado com um caractere da subchave de busca $searchKeyPart$ (linhas 5-8). Se uma dessas comparações falhar antes de chegarmos ao final de $keyPart$, a busca falha (linhas 9-10). Mas, se chegamos ao final de $keyPart$ e de $searchKeyPart$ ao mesmo tempo numa folha, o algoritmo devolve os valores contidos naquela folha (linha 13). Naturalmente, a busca falha se chegamos ao final de uma folha e não há mais caracteres na árvore para comparar, mas ainda há caracteres da chave que não foram comparados(linha 15). Caso contrário, chega-se ao final de $keyPart$ e o algoritmo segue para o nó filho cujo *id* é igual ao próximo caractere da chave de busca a ser comparado (linha 15). Por fim, se esse nó filho não existe, significa que não há uma chave na árvore igual a $searchKeyPart$ e a busca falha (linha 20).

O algoritmo de busca de prefixo (Algoritmo 4 na página 90) também percorre um caminho a partir do nó raiz, comparando cada caractere da subchave de busca $searchKeyPart$ com um caractere do nó atual *node* (linhas 5-8). A diferença desse algoritmo com o anterior é que a busca termina com sucesso quando o final de $searchKeyPart$ é atingido (linhas 9-10). Observe que o algoritmo de busca de prefixo é executado na primeira etapa de uma busca quando a chave de busca possui curingas. Por esse motivo, ao invés de devolver valores contidos em folhas da árvore, ele devolve a posição onde terminou a busca com sucesso, dada pelo nó *node* e pelo índice j. Esse estado será utilizado pela próxima subchave de busca (do tipo padrão ou sufixo) como estado inicial de busca.

O algoritmo de busca de padrão é baseado no algoritmo Knuth-Morris-Pratt [31,51], também conhecido como KMP. Esse algoritmo faz um pré-processamento em cima da subchave padrão $searchKeyPart$ que queremos casar, calculando os valores de uma função π para cada caractere de $searchKeyPart$. O cálculo de π leva tempo $O(|searchKeyPart|)$, onde $|searchKeyPart|$ é o comprimento de $searchKeyPart$. Após esse pré-processamento, o algoritmo KMP percorre uma cadeia de caracteres key em busca do padrão, comparando cada caractere de key uma única vez, utilizando a função π como auxiliar nessa busca. O algoritmo KMP leva tempo $\Theta(|key|)$ para realizar o casamento de padrões, onde $|key|$ é o

Algoritmo 3 SIMPLE_SEARCH(*searchKeyPart*, *node*, *i*)

```
1: j ← 0
2: keyPart ← node.keyPart
3: n ← length[searchKeyPart]
4: m ← length[keyPart]
5: while i < n and j < m and searchKeyPart[i] = keyPart[j] do
6:    i++;
7:    j++;
8: end while
9: if j < m then
10:   return NIL
11: else if node is leaf then
12:   if i = n then
13:      return node.values
14:   else
15:      return NIL
16:   end if
17: else
18:   id ← searchKeyPart[i]
19:   child ← node.children[id]
20:   if child = NIL then
21:      return NIL
22:   else
23:      return SIMPLE_SEARCH(searchKeyPart, child, i)
24:   end if
25: end if
```

comprimento de *key*. A nossa versão do algoritmo KMP é um pouco mais complexa do que a forma tradicional do KMP, pois deve procurar pelo padrão *searchKeyPart* não em uma chave, mas sim em todas as chaves da árvore que casaram com as subchaves que precedem *searchKeyPart* na chave de busca. Para isso, se a subchave que precede *searchKeyPart* for uma subchave prefixa, começamos executando o algoritmo em um nó da árvore, a partir da posição (*node*, *j*) devolvida pelo algoritmo de busca de prefixo. Ao chegarmos ao final desse nó, se o padrão ainda não tiver sido encontrado, continuamos a execução do algoritmo em todos os nós filhos, e assim por diante, até encontramos o padrão em cada uma das subárvores, devolvendo as posições (*node*, *j*) onde a busca terminou com sucesso. O efeito de seguir para todos os nós filhos é equivalente ao resultado de buscar pelo padrão em todas as chaves da árvore que casaram com a subchave prefixa antecedente ao padrão *searchKeyPart* na chave de busca. De forma similar, se *searchKeyPart* não é precedida por uma subchave prefixa, mas sim por outra subchave padrão, a busca por *searchKeyPart* terá início em todas as posições (*node*, *j*) devolvidas pela busca da subchave anterior.

O algoritmo de busca de padrão é o Algoritmo 5 na página 91. Ele inicia da mesma forma que os

Algoritmo 4 PREFIX_SEARCH($searchKeyPart$, $node$, i)

```
 1: j ← 0
 2: keyPart ← node.keyPart
 3: n ← length[searchKeyPart]
 4: m ← length[keyPart]
 5: while i < n and j < m and searchKeyPart[i] = keyPart[j] do
 6:    i++;
 7:    j++;
 8: end while
 9: if i = n then
10:    return (node, j)
11: else if j = m then
12:    id ← searchKeyPart[i]
13:    child ← node.children[id]
14:    if child ≠ NIL then
15:       return PREFIX_SEARCH(searchKeyPart, child, i)
16:    end if
17: end if
18: return NIL
```

dois anteriores: cada caractere de $searchKeyPart$ é comparado com um caractere de $node$ (linhas 5-8). Quando essa iteração termina, ele verifica qual das seguintes condições é válida: (i) a totalidade de $searchKeyPart$ foi comparada com sucesso; (ii) a totalidade de $keyPart$ foi comparada com sucesso; (iii) uma comparação falhou. Na condição (i), a busca termina e a posição atual em $node$ é adicionada ao resultado (linha 10). Caso a condição (ii) seja verdadeira, a busca é executada em cada nó filho (linhas 12-13). Finalmente, na condição (iii), a função π é aplicada a i de forma que o algoritmo possa continuar a comparação no caractere j de $node$ (linhas 16-17). O resultado desse algoritmo é uma coleção de posições j em nós $node$. Essas posições serão utilizadas na busca da subchave posterior a $searchKeyPart$, que pode ser uma subchave padrão ou sufixa.

O algoritmo de busca de sufixo é o mais complexo dos quatro algoritmos de busca, pois a busca de sufixo se opõe à estrutura da árvore prefixa. Naturalmente, seria possível criar duas árvores, uma prefixa e a outra sufixa, de modo a prover algoritmos simples e eficientes para a busca tanto de prefixos como de sufixos. Mas nós ainda não teríamos uma solução para a busca de padrão, nem para chaves de busca mais complexas (como, por exemplo, `"prefix*pattern*suffix"`). Por esse motivo e dadas as restrições de espaço de memória, nós decidimos utilizar apenas a árvore prefixa definida na Seção 6.2.1.

A grande dificuldade da busca de sufixo está no fato de que os últimos caracteres de uma chave estão espalhados numa árvore prefixa. Dado um sufixo $searchKeyPart$ de comprimento $n = |searchKeyPart|$, queremos comparar os últimos n caracteres das chaves da árvore. Porém, é difícil determinar qual a posição dos caracteres da subchave de um dado nó em relação ao final das chaves contidas na subárvore desse nó. Os caracteres podem estar a uma distância x do final de uma chave e, ao mesmo tempo, a

Algoritmo 5 PATTERN_SEARCH($searchKeyPart$, $node$, i, j)

```
 1: keyPart ← node.keyPart
 2: n ← length[searchKeyPart]
 3: m ← length[keyPart]
 4: create collection result
 5: while i < n and j < m and searchKeyPart[i] = keyPart[j] do
 6:    i++;
 7:    j++;
 8: end while
 9: if i = n then
10:    result.add(node, j)
11: else if j = m then
12:    for all child of node do
13:       result.add(PATTERN_SEARCH(searchKeyPart, child, i))
14:    end for
15: else
16:    i ← π[i]
17:    result.add(PATTERN_SEARCH(searchKeyPart, node, i, j))
18: end if
19: return result
```

uma distância y do final de outra chave. Devido a essa imprecisão, uma implementação ingênua da busca de sufixo seria realizar uma busca pelo sufixo da mesma forma que buscamos por padrões, só que, sempre que o sufixo for encontrado na árvore, aplicaríamos a função π para continuar a busca. Esse algoritmo ingênuo devolveria os valores de todas as folhas que tivessem uma busca bem sucedida terminada no último caractere da sua subchave.

Para que possamos evitar o custo de comparar o sufixo com os caracteres da árvore de maneira tão imprecisa, é preciso usar uma estimativa do quão distante podemos estar do final de uma chave na árvore. Isso permite que a busca de sufixo pule os caracteres que estão longe demais de um final de chave, mais longe do que o comprimento da subchave que quero comparar. Para ilustrar melhor esse conceito, considere o sufixo `"Observer"`. Durante a busca por esse sufixo, suponha que estamos num nó cuja distância mínima do final de uma chave é de 11 caracteres. Considerando que o sufixo `"Observer"` tem comprimento 8, é fácil ver que não é necessário comparar caracteres que estão tão distantes assim do final de uma chave. Evitar essa comparação desnecessária é a função do mecanismo de **controle de comprimento**. Uma formalização desse mecanismo está disponível na Seção B.2 na página 145. Por ora, o importante é entender que o intuito desse controle é evitar a comparação desnecessária de caracteres com um sufixo quando estamos distantes demais do final de uma chave.

Continuando o exemplo do sufixo `"Observer"`, suponha agora que, dada uma posição ($node$, j) na árvore, estamos a uma distância de 8 caracteres do final de uma chave, e estamos também a uma distância de 20 caracteres do final de outra chave mais longa. Como mostra a Figura 6.2 na página 84, a

árvore prefixa não possui uma estrutura balanceada e, durante um passeio pelos nós da árvore, podemos estar muito próximos ao final de uma chave e ainda assim distantes do final de outra chave. Para lidar com isso, utilizamos a função π do algoritmo KMP. A uma distância de 8 caracteres do final de uma chave, não temos alternativa a não ser começar a comparação do nosso sufixo `"Observer"` na posição atual. À medida que seguimos a busca pelos nós filhos do nó atual, o valor da distância mínima ao final de uma chave pode mudar. Como exemplo, ao seguir para um nó filho, posso passar a ter uma distância de 5 caracteres do final da chave mais curta, mas estar no índice $i = 7$ do meu sufixo `"Observer"`. Nesse cenário, aplicamos a função π do algoritmo KMP, pois ela permite que continuemos a comparação com a subchave do nó atual, apenas atualizando o índice i da subchave sufixa `"Observer"`.

O Algoritmo 6 na próxima página implementa a busca de sufixo. Para isso, ele lança mão do controle de comprimento de chaves (linhas 5-9), o que permite saber de antemão o quão distante a posição atual está do final de uma chave, evitando comparações desnecessárias. Após realizar esse primeiro passo, os caracteres de *searchKeyPart* são comparados com os de *node* (linhas 10-13). Uma vez que essa comparação chega ao fim, é preciso averiguar os valores dos índices i e j, para que possamos determinar se chegamos ao final de *searchKeyPart* e de *node*. Caso positivo, a busca termina com sucesso se *node* for uma folha, ou continua num nó filho identificado pelo caractere nulo. Esse último passo requer atenção especial. Devido ao controle de comprimentos, o algoritmo de busca de sufixo terminará sua comparação com *searchKeyPart* apenas se estiver fazendo a comparação com o final de uma chave $k \in K$. Sendo assim, se *node* não é uma folha, deve haver um nó filho com uma subchave vazia, identificada pelo caractere nulo[5]. Nas linhas 26-27, se o algoritmo não chegou ao final de *searchKeyPart* mas terminou a comparação de *node*, a busca deve continuar em cada nó filho. Por fim, se uma comparação falhou, a função π é aplicada da mesma forma que é feito com o algoritmo de busca de padrão, para que possamos dar continuidade à comparação de caracteres a partir de um índice diferente de *searchKeyPart*.

Além desses quatro algoritmos, a árvore de sombras suporta buscas por expressões ***`package`*** (veja a tabela 6.1 na página 72). O algoritmo para realizar essa busca é apresentado no Apêndice B. Nesse apêndice, também formalizamos o mecanismo de otimização baseado em controles de comprimentos de chaves que a árvore utiliza, além de mostrar mais detalhes sobre sua estrutura.

Nas próximas seções iremos detalhar a estrutura do grafo e os mecanismos de busca que ele suporta.

[5] Vimos na Seção 6.2.1 que esse cenário ocorre quando há uma chave *key* que é prefixo de outra chave *key*′ na árvore. Na árvore de nós de classe, essa situação ocorre na presença de classes internas, já que o nome da classe externa é sempre um prefixo do nome da classe interna.

Algoritmo 6 SUFFIX_SEARCH($searchKeyPart$, $node$, i, j)

```
1: keyPart ← node.keyPart
2: n ← length[searchKeyPart]
3: m ← length[keyPart]
4: create collection result
5: relevantIndex ← Φ_min(node) − |searchKeyPart|
6: if j < relevantIndex then
7:    i ← 0
8:    j ← relevantIndex
9: end if
10: while i < n and j < m and searchKeyPart[i] = keyPart[j] do
11:    i + +;
12:    j + +;
13: end while
14: if i = n then
15:    if j = m then
16:       if node is not leaf then
17:          child ← node.children['\0']
18:          if child ≠ NIL then
19:             result.add(SUFFIX_SEARCH(searchKeyPart, child, i, 0))
20:          end if
21:       else
22:          result.add(node.values)
23:       end if
24:    end if
25: else if j = m then
26:    for all child of node do
27:       result.add(SUFFIX_SEARCH(searchKeyPart, child, i, 0))
28:    end for
29: else
30:    i ← π[i]
31:    result.add(SUFFIX_SEARCH(searchKeyPart, node, i, j))
32: end if
33: return result
```

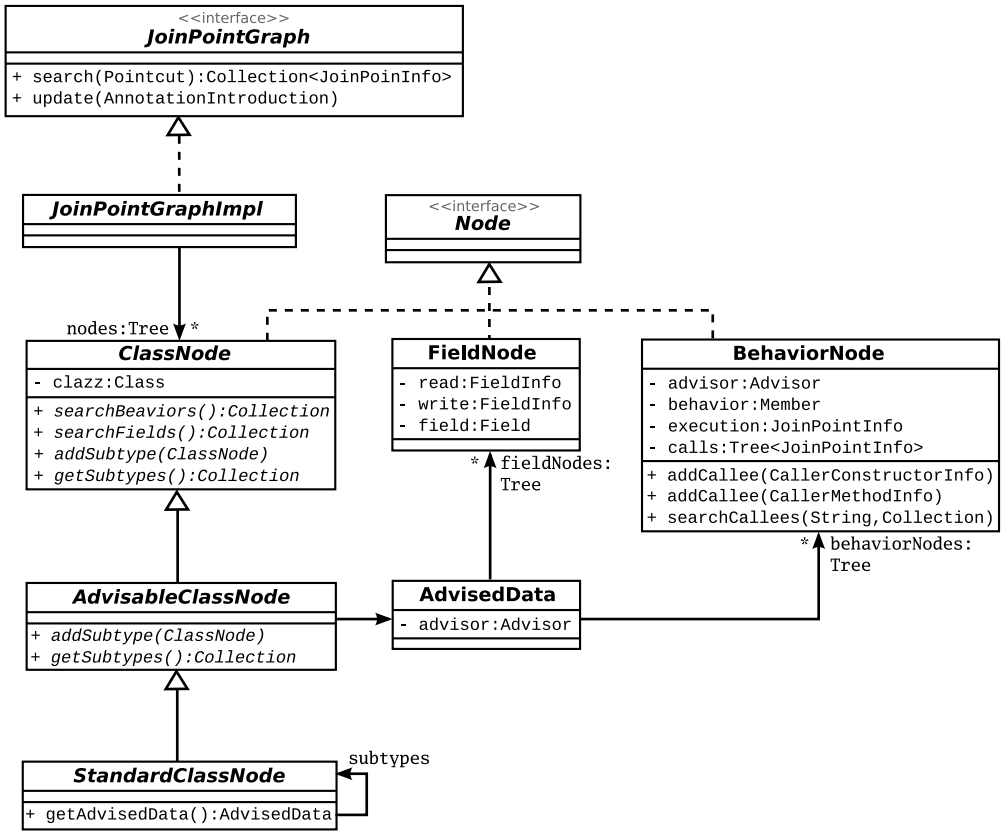


Figura 6.3.: Classes que compõem a estrutura do grafo.

6.3. Estrutura

Na Seção 6.1.2 na página 78 apresentamos a estrutura do grafo e ilustramos essa estrutura na Figura 6.1. Nesta seção iremos mostrar mais detalhes sobre essa estrutura, que é formada pelas classes do diagrama da Figura 6.3.

6.3.1. Árvore de Classes

A árvore de classes foi utilizada como modelo na Seção 6.2 para definirmos os requisitos aos quais a árvore de busca deve atender. Naquela seção, definimos o formato dessa árvore por completo. A árvore de classes é a raiz do grafo, a partir da qual todas as inserções e buscas serão efetuadas. Inserimos nessa árvore nós que representam classes utilizando como chaves os nomes das classes bem como os nomes das anotações utilizadas nessas classes. Os nomes de anotações são precedidos por `'@'` para indicar que se trata de anotações.

Os nós que representam classes são os objetos do tipo `ClassNode`. Vemos no diagrama da Figura 6.3 que esse nó é uma classe abstrata e possui uma referência para a classe que representa, além dos métodos

`searchBehaviors` e `searchFields`, que possibilitam a busca por métodos, construtores e campos, como veremos adiante.

Na Figura 6.1 na página 79, vimos que os nós de classe possuem árvores internas para representar campos, métodos e construtores. No diagrama, porém, vemos que na realidade essas árvores estão em uma classe chamada `AdvisedData`, que é referenciada por `AdvisableClassNode`, outra classe abstrata. O motivo desse nível de indireção é o suporte a domínios e essa indireção será justificada na Seção 6.5.

Além disso, um nó de uma classe deve possuir uma referência às suas subclasses. O método `getSubytpes` de `ClassNode` devolve as subclasses da classe representada pelo objeto `ClassNode`. A forma como esse método é implementado é algo definido nas subclasses de `ClassNode`, decisão de desenho também necessária ao suporte a domínios, como veremos na Seção 6.5. Por ora, definimos o nó de um classe como sendo uma instância de `StandardClassNode`, que possui uma relação de 1 para n com ela mesma, para referenciar as subclasses da classe que representa.

A árvore de objetos `ClassNode` é a base do grafo de sombras e se encontra na classe `JoinPointGraphImpl`. Essa classe abstrata implementa os métodos da interface `JoinPointGraph`, que é visível às classes do outro pacote. Todas as árvore ilustradas na Figura 6.3 possuem acesso protegido, com exceção dessa interface.

6.3.2. Árvore de Campos

Os campos são representados por nós do tipo `FieldNode`, que estão contidos em uma árvore de campos da classe `AdvisedData`, conforme mostra o diagrama da Figura 6.3 na página anterior. Vemos também nessa figura que cada um desses nós pode conter uma sombra do tipo leitura, escrita, ou ambas.

As chaves que identificam os nós `FieldNode` na árvore de campos devem ser compatíveis com as chaves que iremos utilizar na busca. Como as chaves da busca serão extraídas a partir de *pointcuts*, as chaves que identificam objetos `FieldNode` devem conter informações relevantes para os *pointcuts*.

O componente de um *pointcut* que casa com campos foi definido na tabela 6.1 na página 72 como:

field-exp = *attributes field-type type->field*

A parte *type* dessa expressão será utilizada para a busca na árvore de classes. Todo o restante da expressão *field-exp* é utilizado para identificar um campo. Assim, se extraímos a expressão `type` da expressão acima chegamos à seguinte estrutura:

attributes field-type field

Idealmente, utilizaríamos esse formato para gerar chaves. Porém, uma análise mais profunda da expressão acima nos leva à questão dos atributos. A expressão *attributes* é opcional no *pointcut* e é composta por uma lista de atributos que podem estar negados ou não. Assim, para um campo `public`, temos uma infinidade de expressões *attribute* que casam com esse campo. Alguns exemplos de expressões *attribute* possíveis seguem:

```
"public", "!private", "!private !protected !synchronized public",
"public !synchronized", "!static !synchronized public"
```

Para que pudéssemos gerar uma chave identificadora de um campo que case com uma expressão de busca no formato de *field-exp*, precisaríamos gerar mais de uma chave identificadora. Precisamente,

teríamos que gerar uma chave para cada expressão *attributes* possível que casa com os atributos do campo em questão. A quantidade de chaves necessárias para identificar um campo nesse caso seria um número muito alto, aumentando o espaço ocupado pelo grafo na memória.Por esse motivo, geramos chaves identificadores sem os atributos do campo:

field-type field

Como exemplo, dado um campo `public java.lang.Integer identifier`, a chave identificadora dele será `"java.lang.Integer identifier"`. Se o campo possui uma anotação, geramos também uma chave utilizando essa anotação no lugar do nome do campo, como, por exemplo, `"java.lang.Integer @javax.persistence.Id"`.

Ao buscar pelos nós `FieldNode` que casam com uma expressão *field-exp*, fazemos uma busca sem a parte *attributes* da expressão. Se a expressão de busca *field-exp* contiver especificação de atributos, utilizamos esses atributos na etapa posterior à busca, que filtra o resultado de acordo com os atributos contidos em cada campo.

6.3.3. Árvore de Métodos e Construtores

Os métodos e construtores são representados por nós do tipo `BehaviorNode`. Esses objetos estão contidos em uma árvore de `AdvisedData`, referenciada por `AdvisableClassNode`.

Esses nós possuem uma referência para o `JoinPointInfo` que representa a sombra da execução do método ou construtor que o nó representa. Os nós `BehaviorNode` também contêm uma árvore de sombras de chamadas.

A chave que identifica um nó `BehaviorNode` é gerada a partir da assinatura do método ou construtor que ele representa. Assim como fizemos com os outros nós, o formato da chave identificadora do `BehaviorNode` é definido em função da sintaxe de expressões *pointcut*, uma vez que extrairemos dessas expressões a chave de busca. Assim, queremos suportar buscas compatíveis com os formatos as estruturas *method-exp* e *constructor-exp*, especificadas na tabela 6.1 na página 72:

method-exp = attributes return-type type->method\(arg-list\) throws-list

constructor-exp = attributes type->constructor\(param-list\) throws-list

Pelo mesmo motivo que ignoramos os atributos de um campo ao gerar chaves identificadores de campos, iremos ignorar os atributos na chave de um método ou construtor. Decidimos também omitir a lista de exceções `throws-list` pelo mesmo motivo.

A parte `type` das expressões acima também deve ser ignorada, já que essa parte é utilizada para fazer a busca na árvore de classes e não na árvore de métodos e construtores. Excluindo *type*, *attributes* e *throws-list*, temos as expressões que compõem as chaves identificadoras de métodos e construtores:

return-type method\(arg-list\)

constructor\(param-list\)

Do mesmo modo que os nós `FieldNode`, um nó `BehaviorNode` pode ser identificado por mais de uma chave, pois iremos gerar uma chave adicional para cada anotação que puder ser encontrada no método ou construtor que ele representa.

6.3.3.1. Árvores de chamadas

Todo nó `BehaviorNode` possui uma subárvore de chamadas, como pode ser visto na Figura 6.3 na página 94. Essa árvore contém objetos `JoinPointInfo` que representam sombras do tipo chamada, ligando o nó `BehaviorNode` aos métodos e construtores que ele chama durante a sua execução, como foi ilustrado na Figura 6.1-B na página 79.

A chave de busca que utilizamos nessa árvore é extraída a partir de expressões *call*, especificadas pela tabela 6.2 na página 75:

`call\(`*method-exp*`\)`

`call\(`*constructor-exp*`\)`

Extraímos das expressões *call* as estruturas *method-exp* e *constructor-exp*, que serão utilizadas para a busca por chamadas. Sabemos que o formato delas é dado por:

method-exp = *attributes return-type type->method*`\(`*arg-list*`\)` *throws-list*

constructor-exp = *attributes type->constructor*`\(`*param-list*`\)` *throws-list*

Assim como fizemos anteriormente, não utilizaremos a parte *attributes* e *throws-list* para formar as chaves. A expressão `type` dessa vez permanece na chave, pois identifica a classe que está sendo chamada:

method-exp = *return-type type->method*`\(`*arg-list*`\)`

constructor-exp = *type->constructor*`\(`*param-list*`\)`

Esses são os elementos que utilizamos para gerar as chaves identificadoras de chamadas.

6.4. Busca

A busca no grafo é suportada pelos elementos do diagrama da Figura 6.4 na próxima página.

A classe central desse diagrama é `SearchKey`, que é a chave de busca do grafo. O método de `SearchKey` responsável pela busca no grafo é definido em `Searcher`, interface genérica implementada por todos os componentes que realizam buscas. Vemos no diagrama que `SearchKey` faz uma busca em uma árvore de nós `ClassNode`, que, como vimos há pouco, é a raiz do grafo.

A chave de busca é criada por `SearchKeyParser`, que visita um objeto `Pointcut` extraindo partes da expressão *pointcut* para criar toda a estrutura de `SearchKey`. Vemos em `SearchKey` alguns métodos utilizados durante esse processo. Os métodos `addConjunctiveSearchKey` e `addDisjunctiveSearchKey` possibilitam a composição de chaves quando há conectores `AND` e `OR` no *pointcut*. Além disso, o método `addNegativeSearchKey` adiciona chaves negadas. A implementação dessas chaves de busca compostas é trivial e foi descrita na Seção 6.1.2. A classe `DisjunctiveSearchKey` faz a união dos resultados de duas chaves na busca ao passo que `ConjunctiveSearchKey` faz a intersecção dos mesmos. Além disso, `NegativeCompositeSearchKey` extrai do resultado de uma busca o resultado das buscas realizadas por chaves negadas. Essa classe é estendida por `CompositeSearchKey`, que representa a chave composta por duas chaves, e pode ser uma chave disjuntiva ou conjuntiva. Além disso, vemos no diagrama a classe `EmptySearchKey`, utilizada para fazer buscas que devolvem um resultado vazio. Quando há *pointcuts* compostos do tipo `"execution(* *->*(..)) AND field(Pojo)"`, o grafo consegue enxergar que esses

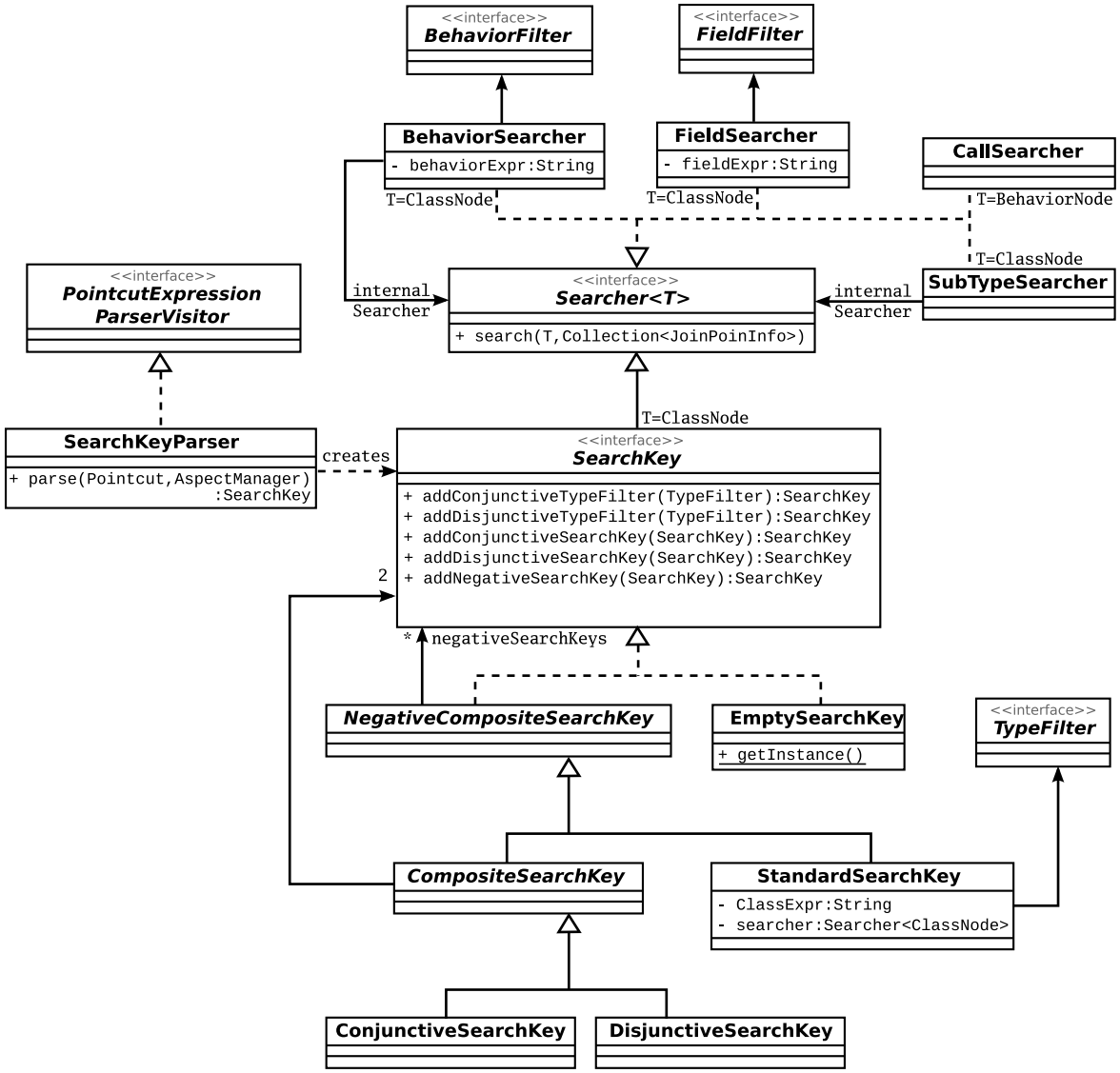


Figura 6.4.: Classes responsáveis pela busca no grafo. Todas as classes desse pacote estão localizadas no pacote `org.jboss.aop.joinpoint.graph` e são de acesso privado do mesmo (*package-protected*).

pointcuts não casam com nenhuma sombra, e cria essa chave de busca fantoche, evitando uma busca desnecessária.

A classe `NegativeCompositeSearchKey` é também estendida por `StandardSearchKey`. Essa é a chave de busca que realiza uma busca na árvore por *pointcuts* primitivos. Para isso, ela contém a chave de busca `classExpression`, extraída a partir da expressão do tipo ***`type`*** do *pointcut*. `StandardSearchKey` também possui uma referência para `TypeFilter`. Esse filtro é utilizado quando a expressão ***`type`***, contida no *pointcut*, é uma definição de tipos, algo que descrevemos na Seção 6.1.1.1 na página 77. As definições de tipos são o único caso que requer filtragem dos objetos `ClassNode` após uma busca. Apesar da filtragem ser necessária, `SearchKeyParser` extrai uma chave de busca de classes a partir de uma declaração `typedef`. Uma definição de tipos pode conter nome de classes ou super classes, ou anotações utilizadas na classe que estamos buscando. A chave de busca mais restritiva é extraída a partir da definição de tipo, para ser utilizada como `classExpression` por `StandardSearchKey` na busca na árvore de nós de classe. Isso evita que tenhamos que procurar por todas as classes e depois filtrar cada uma delas. Para ilustrar, considere a definição de tipo a seguir:

```
"class(org.jboss.aop..) and hasfield(java.lang.Class *->clazz)"
```

A partir dessa definição de tipo, extraímos a chave de busca de classe `"org.jboss.aop.."`, restringindo a quantidade de nós `ClassNode` que teremos que filtrar após a busca.

Após encontrar os nós `ClassNode`, `StandardSearchKey` delega a busca para um objeto buscador interno, que irá utilizar o nó `ClassNode` como alvo de sua busca, devolvendo as sombras de junção resultantes dessa busca. No diagrama vemos que há três classes que implementam `Searcher<ClassNode>`: `SubtypeSearcher`, `FieldSearcher` e `BehaviorSearcher`. A primeira é utilizada para buscas de subclasses, quando há uma expressão `$instanceof` (veja a tabela 6.1 na página 72). A classe `SubtypeSearcher`, inclui no resultado todos os subtipos, diretos e indiretos de um nó de classe. As outras classes que implementam `Searcher<ClassNode>` são `FieldSearcher` e `BehaviorSearcher`. Ambas as classes utilizam uma chave de busca para encontrar os nós procurados nas árvores de campo ou na árvore de construtores e métodos. Além disso, elas também estão associadas a filtros. Esses filtros são utilizados para uma diversidade de filtragens que precisam ser realizadas nos nós encontrados, como filtragem de atributos, por exemplo.

A busca realizada por `BehaviorSearcher` vai além de encontrar nós `BehaviorNode`. De posse dos nós `BehaviorNode`, essa classe delega a busca para um buscador interno, capaz de extrair de um `BehaviorNode` as sombras que devem ser encontradas. Há um buscador interno para retornar sombras do tipo execução, classe que foi omitida na figura, e há o `CallSearcher`. Esse objeto busca por chamadas realizadas por um método ou construtor no `BehaviorNode`.

6.5. Grafos e Domínios

Na Seção 5.5 na página 67, vimos que o grafo deve suportar domínios, de modo que há uma instância de `JoinPointGraph` para ser utilizada para cada domínio.

Isso sugere um cenário de multiplicação dos dados contidos no grafo, onde teríamos uma instância

dessa estrutura de dados para cada domínio, contendo apenas as sombras das classes que compõem o domínio em questão. Nesse cenário, porém, não poderíamos reutilizar os nós `ClassNode` do grafo que representa o domínio principal. Caso pudéssemos, a implementação do suporte a domínios seria trivial. Teríamos um grafo que representa todas as sombras, correspondente ao domínio principal. E teríamos grafos de domínio contendo um subconjunto dos objetos `ClassNode` do grafo principal. Esse subconjunto seria naturalmente composto apenas pelos nós que representam as classes contidas no domínio daquele grafo. Contudo, não é possível arbitrariamente selecionar um conjunto de nós `ClassNode` e criar uma nova árvore de classes com esses nós. Os nós `ClassNode` possuem referências entre si para representar relações de hierarquia. Essas referências inevitavelmente conectarão as classes da nova árvore a outras classes, que podem ou não fazer parte do conjunto de classes que queremos representar. Se esses nós referenciados representam classes que não estão no domínio do novo grafo, teremos um erro no grafo, que poderá devolver sombras dessas classes durante uma busca envolvendo expressões `instanceof`. Assim, estamos impedidos de reutilizar os nós `ClassNode` do grafo principal e, nesse cenário, teríamos que criar os grafos de domínio com cópias desses nós. Isso geraria um volume maior de dados na memória, acarretando numa sobrecarga que queremos evitar.

Ao invés disso, lançamos mão de uma solução seletiva, que será aplicada de forma diferente dependendo do tipo do domínio que um grafo deve representar. Um domínio pode ser de três tipos: uma representação de um escopo no servidor de aplicações JBoss AS; um domínio de uma classe ou um domínio de uma instância. A interface que permite acesso ao domínio de um escopo no servidor ainda está em construção e, portanto, dificilmente será utilizada no momento. Assim, provemos suporte a esse tipo de domínio com um algoritmo ingênuo de busca, que ocupa um espaço mínimo na memória. Essa solução é implementada pela classe `DomainJoinPointGraph`, ilustrada na Figura 6.5 na próxima página. Internamente, esse grafo é formado pela mesma árvore de nós `ClassNode` que compõe o grafo do domínio principal, `MainJoinPointGraph`. A diferença entre esses dois grafos é que `DomainJoinPointGraph` faz uma filtragem adicional durante a busca, selecionando apenas os dados referentes ao domínio que representa. Para que isso seja possível, temos um tipo especial de `AdvisedData`, o `DomainData`. O objeto `AdvisedData` contido em nós `StandardClassNode` é sempre um objeto `DomainData`. Podemos ver no diagrama que a classe `DomainData` tem uma lista contendo os nomes das *tags* que identificam todos os domínios dos quais faz parte. Durante uma busca, o grafo `DomainJoinPointGraph` filtra o resultado da busca utilizando essas *tags* para verificar se as sombras pertencem ou não ao domínio que o grafo representa.

A solução que aplicamos aos grafos de domínios de classe e de instância, por sua vez, não utilizam filtragens o que as torna são tão eficientes quanto o grafo do domínio principal. Esses grafos possuem uma árvore de classes exclusiva e uma quantidade de dados duplicados mínimos, pois referencia os dados contidos no grafo principal. Vimos anteriormente que as árvores de campos e métodos ficam contidas num nó `ClassNode` de forma indireta, através de uma relação com um objeto `AdvisedData`. O motivo dessa indireção é justamente o suporte a domínios de classe e de instância. Esses objetos `AdvisedData` possuem uma estrutura que reflete a estrutura aninhada de domínios de classe e de domínios de instância (vide a Figura 4.3 da Seção 4.1.4). Usamos o padrão Composite [27] para implementar essa estrutura, como mostra a Figura 6.5. Um domínio de classe é representado por um `CompositeDomainData`, cujas

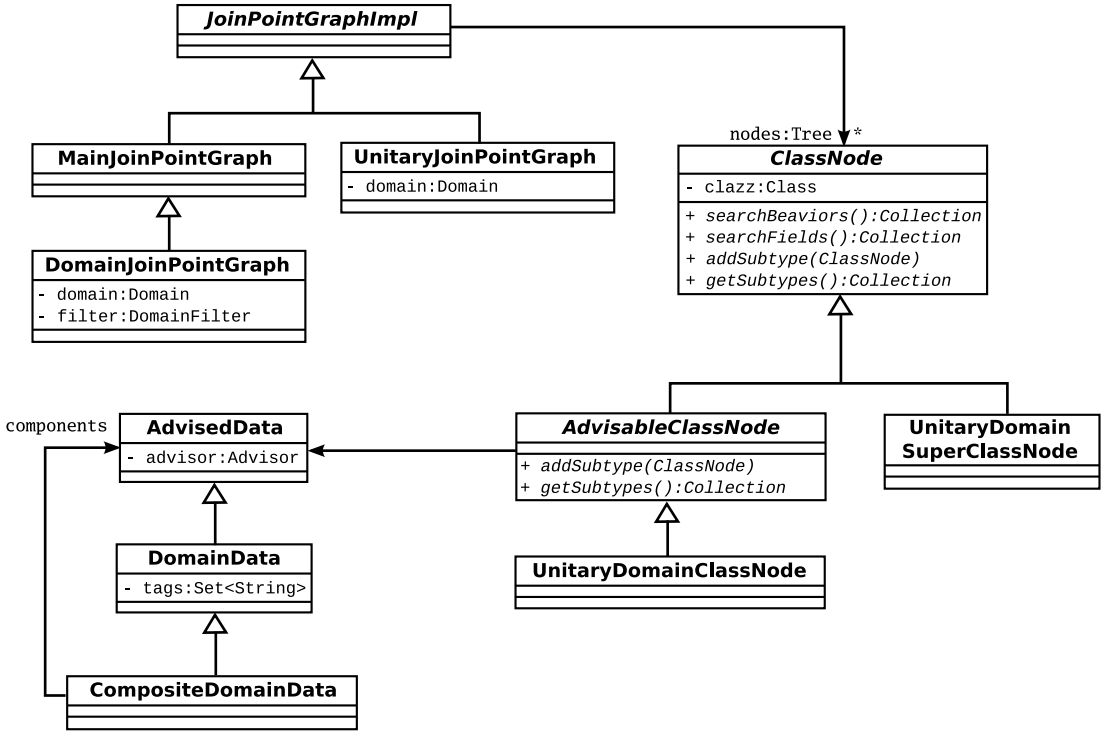


Figura 6.5.: Classes do pacote `org.jboss.aop.joinpont.graph` que fornecem suporte a domínios.

subárvores contêm apenas as sombras associadas a uma classe. As sombras gerenciadas por *advisors* de instâncias pertencem a um domínio de instância e ficam organizadas em uma coleção de objetos `AdvisedData`. Cada um desses objetos contém apenas as sombras pertencentes ao domínio de uma única instância. Sempre que uma busca por nós do tipo `FieldNode` e nós do tipo `BehaviorNode` é realizada no objeto `CompositeDomainData`, esse objeto faz uma busca nas subárvores internas e também delega a mesma busca para os objetos `AdvisedData` que o compõem, unificando os resultados de forma transparente.

Essa estrutura permite que o grafo de um domínio de classe ou de instância referencie apenas o objeto `AdvisedData` que contém as sombras daquele domínio. Quando se trata de um domínio de instância, as únicas sombras que devem estar contidas no grafo são as sombras contidas no objeto `AdvisedData` associado ao *advisor* daquela instância. Assim, criamos um objeto `ClassNode` que servirá como invólucro para o objeto `AdvisedData` e inserimos esse nó `ClassNode` numa árvore de classes, que é a raiz do grafo do domínio de instância. Esse nó é do tipo `UnitaryDomainClassNode`, classe também ilustrada na Figura 6.5. Caso se trate do grafo de um domínio de classe, fazemos a mesma coisa, porém o objeto `UnitaryDomainClassNode` possuirá uma referência ao `CompositeDomainData`, que possui as sombras dos domínios da classe e dos seus subdomínios, correspondentes às instâncias daquela classe.

Como todas sombras desses domínios estão contidas em um único nó de classe, denominamos esses

grafos de `UnitaryJoinPointGraph`. A árvore de classes que compõe esse grafo, porém, não pode conter apenas esse nó. Precisamos suportar buscas que envolvem as superclasses da classe representada. Assim, uma busca por `$instanceof(java.lang.Object)` deve devolver o nó `UnitaryClassNode` contido na árvore. Para isso, utilizamos um nó fantoche, que aponta para o objeto `UnitaryClassNode` como subclasse. Esse nó representa todas as superclasses da classe contida no domínio e é inserido na árvore de classes de `UnitaryJoinPointGraph` com todos os nomes dessas superclasses, além dos nomes das anotações contidas nessas superclasses. O papel de nó fantoche é representado pela classe `UnitaryDomainSuperClassNode`. Note que ele não possui nenhuma informação referente a sombras, já que as super classes não pertencem ao domínio representado pelo grafo.Do mesmo modo, o nó `UnitaryClassNode` não aponta para nós das subclasses, pois essas subclasses não fazem parte do domínio unitário e não devem fazer parte do resultado de uma busca nesse domínio.

7. Instrumentação em Tempo de Execução

Antes de realizarmos este trabalho, o JBoss AOP não fornecia suporte à instrumentação em tempo de execução. Descrevemos a abordagem utilizada pelo JBoss AOP na Seção 4.2.2 na página 59. Essa abordagem consistia na inserção de ganchos em todas as sombras preparadas durante a combinação estática. Assim, quando uma operação dinâmica é executada, bastava adicionar e remover adendos nas cadeias contidas nos *advisors*, processo que denominamos de atualização de cadeias. A simples execução desse processo se reflete automaticamente na execução dos ganchos, que utilizam as cadeias contidas nos *advisors* para a interceptação dos pontos.

Como vimos no Capítulo 3, essa abordagem altera o fluxo de controle da aplicação ao inserir chamadas a ganchos vazios, desnecessários para a execução dos pontos de junção que não serão interceptados. O custo adicional decorrente pode prejudicar o desempenho do sistema, estimulando o usuário a limitar o número de pontos de junção preparados. A instrumentação em tempo de execução, em contrapartida, encoraja o usuário a preparar todos os pontos de junção de um sistema, pois isso não afetará o desempenho do mesmo na ausência de aspectos.

A segunda parte do nosso trabalho consiste em implementar a instrumentação em tempo de execução no JBoss AOP, para que possamos inserir e remover chamadas aos ganchos conforme a presença ou ausência de adendos nas sombras do sistema base durante a execução do programa. Para que isso seja possível, é preciso atualizar os *bytecodes* das classes carregadas em tempo de execução, processo denominado *hot swap*.

Vimos na página 35 da Seção 3.1.2 que há diversas técnicas para realizar o *hot swap*. Contudo, uma dessas técnicas se destaca em relação as demais, que é a API *hot swap* da JVMTI (*Java Virtual Machine Tools Interface*). Essa API é a única que faz parte da especificação da linguagem Java e pode garantir portabilidade à nossa solução. Por esse motivo, decidimos utilizar a JVMTI para implementar a instrumentação em tempo de execução no JBoss AOP.

Vimos na Seção 2.5.1 que, para utilizar a JVMTI, é necessário implementar um agente. Através do método `premain`, o agente tem acesso à classe `Instrumentation`. Essa classe disponibiliza o método `redefineClasses`, através do qual é possível redefinir os *bytecodes* de uma ou mais classes. Esse é o meio que utilizamos para realizar o *hot swap* na nossa implementação. Como o JBoss AOP já provê um agente JVMTI, responsável pela instrumentação em tempo de carga, decidimos utilizar esse mesmo agente para acessar a funcionalidade de *hot swap* da JVMTI. Além disso, implementamos a instrumentação em tempo de execução no JBoss AOP como sendo uma funcionalidade opcional. Isso nos permitiu conduzir experimentos a fim de comparar o desempenho do JBoss AOP com e sem o uso de *hot swap*. Para habilitar o suporte a *hot swap*, basta passar um parâmetro adicional para o agente, como exemplificado abaixo:

```
-javaagent:jboss-aop-jdk50.jar=-hotSwap
```

O uso da instrumentação em tempo de execução no JBoss AOP resultará em alterações no funcionamento dessa ferramenta. Em primeiro lugar, a preparação de sombras deve funcionar de forma diferente. Queremos que, assim como no AspectWerkz, os ganchos vazios adicionados ao código do sistema base não sejam chamados durante a execução do sistema. Só assim poderemos garantir que o fluxo de controle não é alterado na ausência de adendos. Em contrapartida, durante alterações dinâmicas queremos não apenas atualizar as cadeias de adendos associadas às sombras afetadas por tais alterações. Queremos também instrumentar o código das sombras de junção afetadas, substituindo as mesmas por chamadas aos ganchos correspondentes, de forma a permitir a execução correta dos adendos nesses pontos. Informações sobre a nossa implementação desse novo mecanismo serão vistas nas seções seguintes.

A nossa solução completa pode ser vista no diagrama da Figura 7.1 na próxima página. Como queremos que a instrumentação em tempo de execução coexista com a solução anterior, utilizamos o padrão Strategy [27]. A estratégia de implementação da programação orientada a aspectos dinâmica é representada pela interface `DynamicAOPStrategy`. Essa estratégia é responsável por prover:

- um algoritmo de classificação de pontos de junção: que indicará se, durante a preparação das sombras, essas devem ser substituídas pelos ganchos correspondentes ou não.

- uma instância de `InterceptorChainObserver`: esse objeto será notificado das atualizações de cadeias, decorrentes de operações dinâmicas. No caso da instrumentação em tempo de execução, isso culminará no *hot swap* dos *bytecodes* das sombras afetadas.

- e um método *callback* para a notificação da finalização da execução de uma operação de POA dinâmica: esse método permitirá a atomicidade da instrumentação em tempo de execução, como veremos adiante.

Existem duas implementações dessa estratégia. A classe `LoadInterceptedClassesStrategy` mantém o comportamento original do JBoss AOP. Analisando o algoritmo descrito na Seção 3.1.1, é possível deduzir como tal classe é implementada. Primeiramente, ela deverá prover um algoritmo de classificação que indique que sombras preparadas devem ser substituídas pelos ganchos gerados durante a transformação. Além disso, essa estratégia não precisa ser notificada da atualização das cadeias de adendos, pois essa alteração será o suficiente para a efetivação de uma operação dinâmica, visto que os ganchos que fazem chamadas aos adendos dessa cadeia já estão inseridos no local da sombra instrumentada. Portanto, ela não fornece observador de cadeias algum. Pelo mesmo motivo, a notificação realizada através do método *callback* deve ser ignorada por essa estratégia. Por outro lado, a classe `HotSwapStrategy` implementará a solução de instrumentação em tempo de execução. Nas próximas seções, veremos cada uma das partes que compõem essa estratégia. Ao final, na Seção 7.4, reveremos esse diagrama a fim de mostrar uma visão completa de como as peças se encaixam na sua implementação.

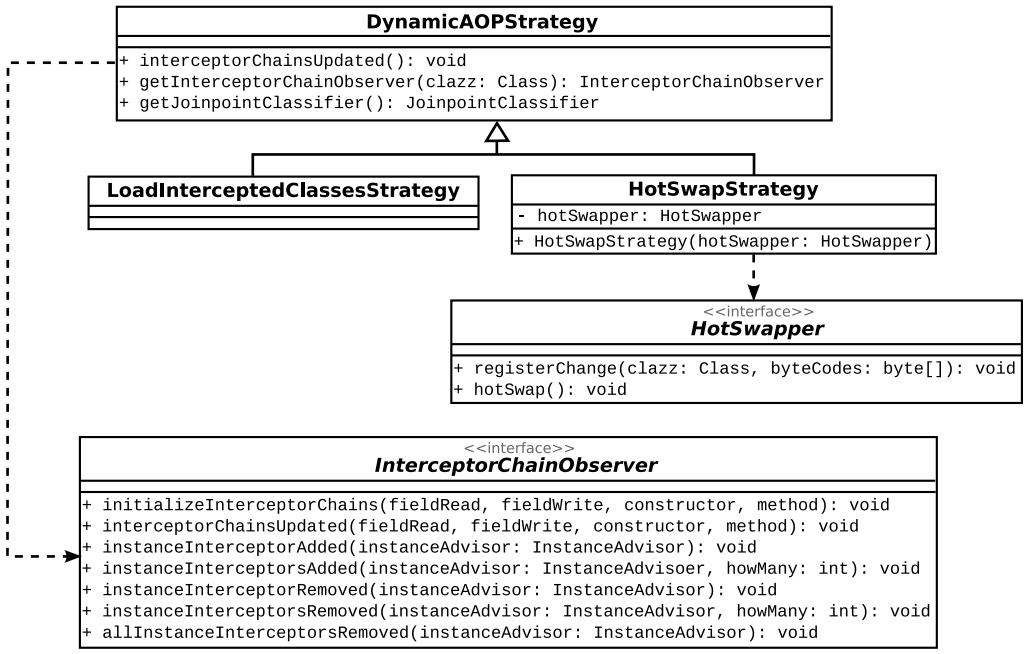


Figura 7.1.: Arquitetura da implementação da instrumentação em tempo de execução no JBoss AOP.

7.1. Classificação de Sombras de Junção

Para que possamos implementar a instrumentação em tempo de execução, precisamos alterar a forma como sombras de junção são classificadas.

A forma como as sombras são classificadas devem atender a algumas restrições da linguagem Java, que se refletem na etapa de preparação.

7.1.1. Preparação de Sombras e Hot Swap

Por questões de segurança, o *hot swap* em Java apresenta algumas restrições. Isso impede a aplicação de mudanças arbitrárias nos *bytecodes* de uma classe carregada, já que a alteração da assinatura de uma classe[1] não é permitida. Entretanto, a instrumentação de uma sombra de junção acarreta na adição de métodos e de campos auxiliares à classe que contém tal sombra. Devido às restrições do *hot swap*, essas alterações precisam ser realizadas antes de a classe ser carregada.

Sendo assim, para que um ponto de junção seja passível de interceptação, é preciso que todos os

[1] A assinatura de uma classe é composta pela lista dos campos que ela contém, bem como pelo conjunto de todas as assinaturas de todos os seus métodos e construtores. Afirmar que a assinatura de uma classe não pode ser alterada significa que não é permitido adicionar ou remover campos, métodos e construtores; alterar o tipo ou o nome de campos; ou alterar a assinatura de construtores e métodos.

campos e métodos necessários para a instrumentação de sua sombra sejam adicionados antes de os *bytecodes* serem carregados pelos *class loaders*. Isso deve ser realizado durante a transformação que, como vimos na Seção 2.5 na página 20, pode ocorrer em tempo de compilação ou de carga. É por esse motivo que as ferramentas vistas na Seção 3.1 adicionam ganchos antes que o sistema seja executado.

A única exceção está na implementação mais recente do PROSE [62]. Na Seção 3.1.4 na página 36, vimos que é possível instrumentar métodos sem a utilização de métodos ou campo auxiliares. O PROSE substitui o corpo de um método na íntegra quando adiciona e remove adendos. Para isso, ele precisa armazenar o corpo original do método, já que o corpo de um método instrumentado é gerado a partir desse corpo original. Essa implementação do PROSE dispensa, portanto, a preparação dessas sombras, evitando o custo decorrente dessa etapa. Em troca desse custo, existe o custo adicional de armazenar os `bytecodes` que compõem os corpos dos métodos do sistema base.

Enquanto que essa solução poderia ser aplicada também à instrumentação de construtores, seria difícil estendê-la à interceptação de outros tipos de ponto de junção, como execução de construtores e acessos a campos. Tome como exemplo pontos do tipo leitura de campos. Nesse caso, dispensar o uso de ganchos para esses pontos resultaria em adicionar chamadas a adendos em todas as sombras que lêem o valor de um campo a ser interceptado, o que, por sua vez, resultaria no armazenamento dos corpos originais de todos os métodos e construtores que contiverem essas sombras instrumentadas. Além disso, diferentemente da interceptação de métodos, esse tipo de instrumentação não poderia tratar o corpo dos métodos que acessam o campo como um bloco único. É preciso instrumentar apenas os trechos do corpo que fizerem a leitura do valor do campo em questão. Isso aumentaria a complexidade da solução apresentada em [62], caso fosse aplicada a esses tipos de pontos de junção.

Podemos afirmar, com isso, que atualmente a preparação de sombras de junção é necessária para a implementação da combinação dinâmica numa ferramenta que inclua interceptação de leituras e escritas de campos, como é o caso do JBoss AOP. Implementar a solução apresentada em [62] no JBoss AOP está fora do escopo desse trabalho, que busca avaliar soluções aplicáveis a todos os tipos de pontos de junção.

Dessa forma, a preparação é um passo que deve permanecer como parte da combinação dinâmica no JBoss AOP, tanto para a instrumentação em tempo de execução quanto para a implementação tradicional dessa ferramenta. Na implementação da instrumentação em tempo de execução, é preciso adicionar os campos auxiliares e ganchos durante a transformação inicial do sistema, devido às restrições impostas pelo *hot swap*. Também manteremos a implementação antiga de forma plugável. Nesse caso, a preparação deverá executar como antes, adicionando chamadas a ganchos vazios ao fluxo de controle de todas as sombras que casarem com quaisquer *pointcuts* configurados no sistema.

7.1.2. Algoritmos de Classificação de Sombras

No JBoss AOP, a preparação é executada pelos transformadores. Vimos que esses transformadores analisam as sombras de um tipo específico, uma a uma, verificando se elas devem ou não ser instrumentadas, e realizando essa instrumentação caso necessário. Para isso, em sua versão original, os transformadores iteravam pelos *pointcuts* contidos no `AspectManager`, procurando por um *pointcut*

que casasse com a sombra sendo analisada. Chamamos esse algoritmo de **algoritmo de classificação**, onde a sombra é classificada com o intuito de definir se ela deverá ou não ser instrumentada. Caso esse *pointcut* fosse encontrado, o gancho necessário para a interceptação daquela sombra seria adicionado, e os *bytecodes* da própria sombra seriam substituídos por uma chamada ao gancho, responsável por executar todos os adendos associados à sombra, assim como por executar o código original da própria sombra instrumentada.

Queremos quebrar a transformação em duas partes: a geração de ganchos; e a substituição das sombras de junção por chamadas aos ganchos correspondentes. Assim, somente as sombras que serão interceptadas deverão ser afetadas pelos dois passos. As sombras que casam com *pointcuts*, mas não serão inicialmente interceptadas, deverão ter apenas os seus ganchos inseridos (nesse caso, o *pointcut* é uma expressão de preparação ou um *pointcut* declarado que não foi referenciado por uma associação). Para que os transformadores possam dividir a instrumentação nessas duas etapas, é preciso distinguir as sombras que serão de fato interceptadas por adendos das que serão somente preparadas. Isso requer um novo algoritmo de classificação de sombras, que indicará quando uma sombra de junção deve ter apenas o seu gancho criado, bem como quando ela deve não só ter esse gancho gerado como deve ser substituída por uma chamada a ele. Esse algoritmo classifica, portanto, um ponto de junção em três categorias diferentes: não instrumentar(`NOT_INSTRUMENTED`), preparar (`PREPARED`) e substituir (`WRAPPED`). Esse novo algoritmo de classificação de sombras de junção deve ser utilizado pelos transformadores para implementar a instrumentação em tempo de execução.

O novo algoritmo de classificação de sombras, que denominamos de `FULL_CLASSIFY`, pode ser visto na página seguinte.

Note que o *hot swap* pode não estar disponível se o usuário estiver utilizando uma máquina virtual mais antiga (versão 1.4 ou anterior) ou, simplesmente, pode não ser a escolha do usuário. Quando a instrumentação não puder ser realizada em tempo de execução, executar o algoritmo sugerido seria desnecessário, inclusive porque ele apresenta custo maior em relação ao algoritmo original, que classificava cada sombra em apenas duas categorias (instrumentar ou não instrumentar). Na nomenclatura de classificação que utilizamos em nosso algoritmo, isso seria equivalente às classificações `WRAPPED` (instrumentar completamente, gerando os ganchos e substituindo as sombras por chamadas aos mesmos) e `NOT_INSTRUMENTED` (não instrumentar a sombra, que conseqüentemente não será interceptada em tempo de execução). O algoritmo original de classificação, que denominamos de `SIMPLE_CLASSIFY`, pode ser visto a vpageref[a seguir]alg:simpleClassify.

7.1.3. Estratégia de Classificação

Dado que queremos fornecer um novo algoritmo de classificação e, ao mesmo tempo, manter o algoritmo antigo, implementamos esses algoritmos usando o padrão *Strategy* [27]. A estratégia a ser utilizada durante a transformação é recebida pelo construtor de `Instrumentor`, a fachada para o pacote de instrumentação `org.jboss.aop.instrument`. Essa classe, que é responsável por realizar a combinação de aspectos delegando tal tarefa aos transformadores, passa aos transformadores a estratégia de classificação.

Algoritmo 7 FULL_CLASSIFY(*shadow*)

```
1: classification ← NOT_INSTRUMENTED
2: for all pointcutInfo ∈ AspectManager do
3:    if classification = PREPARED and pointcutInfo.binding = NIL then
      /* evitar casamentos que não levarão a um resultado novo */
4:       continue for loop
5:    end if
6:    if pointcutInfo.pointcut → matches(shadow) then
      /* um pointcut que casa com a sombra foi encontrado */
7:       if pointcutInfo.binding = NIL then
         /* não existe associação que referencie esse pointcut: apenas preparar */
8:          classification ← PREPARED
9:       else
         /* uma associação referencia esse pointcut: instrumentar a sombra */
10:         classification ← WRAPPED
11:         break for loop
12:      end if
13:   end if
14: end for
15: return classification
```

Algoritmo 8 SIMPLE_CLASSIFY(*shadow*)

```
1: for all pointcutInfo ∈ AspectManager do
2:    if pointcutInfo.pointcut → matches(shadow) then
      /* se um pointcut casa com a sombra, a classificação indica que a instrumentação completa deve ser feita */
3:       return WRAPPED
4:    end if
5: end for
6: return NOT_INSTRUMENTED
```

Uma conseqüência dessa solução é que os transformadores não precisam estar cientes de que têm de agir de formas diferentes comforme o algoritmo de combinação que estiver sendo utilizado. Eles apenas delegam a classificação das sombras para essa estratégia que indicará, para cada sombra, se ela não deve ser alterada (`NOT_INSTRUMENTED`), deve ser preparada (ter o gancho correspondente a ela apenas gerado, equivalente à classificação `PREPARED`) ou substituída (ter o gancho correspondente gerado e ser substituída por uma chamada ao mesmo, equivalente à classificação `WRAPPED`).

A arquitetura dessa solução é detalhada na Figura 7.2. A classe que representa a estratégia é `JoinpointClassifier`. Ela define a assinatura dos métodos de classificação. Todos eles delegam a sua execução para o método abstrato `classifyJoinpoint`, que contém o algoritmo de classificação.

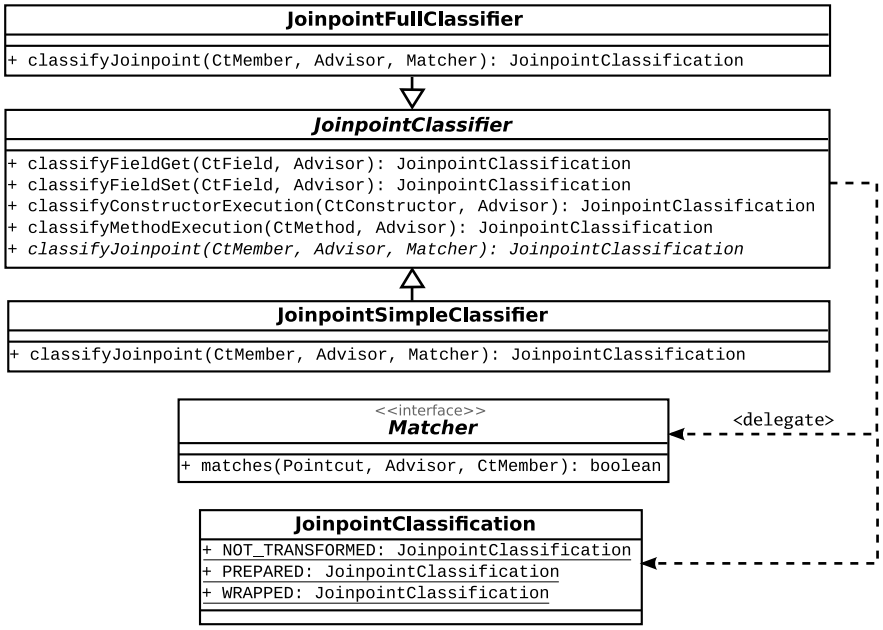


Figura 7.2.: Estratégia de Classificadores de Sombras de Junção.

A classe `JoinpointFullClassifier` implementa tal método usando o primeiro algoritmo que vimos, `FULL_CLASSIFY`, que verifica se o *pointcut* está associado a um ou mais adendos. Já a estratégia `JoinpoinSimpleClassifier` implementa o algoritmo mais antigo, `SIMPLE_CLASSIFY`.

Para possibilitar a implementação de um único algoritmo para todos os tipos de sombras (leituras e escritas de campos, execução de construtores e de métodos, chamadas de métodos e construtores), a interface `JoinpointClassifier.Matcher` é utilizada. O casamento de uma sombra com um *pointcut* é executado por um objeto desse tipo, através do seu método `matches`. Esse objeto é passado como argumento para `classifyJoinpoint`. Cada método público de classificação de `JoinpointClassifier` faz uso de uma instância de `Matcher` apropriada para o tipo de ponto que será classificado. Por exemplo, o método `classifyFieldRead` utiliza uma instância de `Matcher` que casa *pointcuts* com sombras de leitura de campo. Essa instância é passada para o método `classifyJoinpoints`, que

contém o algoritmo de classificação e é independente do tipo de sombra a ser classificada.

Com essa estratégia, obtivemos uma forma efetiva de classificar as sombras de junção de forma dependente da estratégia de programação orientada a aspectos dinâmica utilizada. Sempre que a instrumentação em tempo de execução for a estratégia atual, o algoritmo `FULL_CLASSIFY` deverá ser utilizado. Caso contrário, o algoritmo `SIMPLE_CLASSIFY` é a escolha a ser utilizada.

7.2. Atualização de Cadeias de Adendos

No item anterior, vimos como tratar as sombras de junção que não serão interceptadas inicialmente, mas que casam com uma expressão *pointcut* contida no `AspectManager`. Essas sombras são preparadas pelos transformadores e podem passar a ser interceptados durante a execução do sistema através de uma operação de POA dinâmica.

Como vimos anteriormente, o JBoss AOP atualiza as cadeias dos adendos para efetivar uma operação de combinação dinâmica. Na nossa implementação de instrumentação em tempo de execução, a atualização de cadeias não é mais suficiente. Quando uma cadeia previamente vazia passa a conter adendos, é preciso substituir os *bytecodes* da sombra correspondente a essa cadeia, adicionando uma chamada ao gancho que executará a cadeia de adendos. De forma análoga, é preciso reverter o processo de substituição das sombras cujos pontos de junção não serão mais interceptados por adendos. Para isso, basta substituir a chamada ao gancho pelo código original da sombra. Chamamos esse processo de **reversão de substituição**, ou *unwrapping* em inglês.

Observe que estamos implementando a instrumentação em tempo de execução como uma opção, mantendo a abordagem anterior, sem *hot swap*, também disponível. Assim, queremos que as classes que executam a instrumentação em tempo de execução tenham uma arquitetura plugável. Para isso, foi criada a interface `InterceptorChainObserver`, que implementa o padrão Observer [27]. Essa solução pode ser vista no diagrama d a Figura 7.1 na página 105. Instâncias dessa interface são atribuídas a todos os *advisors* do sistema, para serem notificadas das atualizações de cadeias que ocorrerem durante a execução do sistema.

A interface `InterceptorChainObserver` possui um método para notificação de atualização de cadeias de classes contidas em um `ClassAdvisor`, `interceptorChainsUpdated`, além de métodos para a notificação de mudanças nas cadeias contidas em objetos *InstanceAdvisor* (todos os métodos cujo nome começa com `instanceInterceptor`, além do método `allInstanceInterceptorsRemoved`).

Note que esse mecanismo deve estar integrado à nossa implementação de atualização de cadeias de adendos, descrita no Capítulo 5. Para que pudéssemos implementar a nossa solução, criamos a classe `JoinPointManager`, uma estratégia responsável por atualizar as cadeias quando uma operação dinâmica ocorre (vide a Figura 5.1 na página 62). Há duas versões dessa estratégia: `AdvisedJoinPointManager` e `IndexedJoinPointManager`. A primeira delega a atualização das cadeias para os *advisors*. Cada um desses irá notificar o `InterceptorChainObserver` da atualização de cadeias que executou. Por outro lado, a estratégia `IndexedJoinPointManager` faz a atualização das cadeias sem passar por *advisors*. Observe que nesse momento é também preciso notificar todos os objetos `InterceptorChainObserver`

associados com os *advisors* cujas cadeias foram atualizadas. Sendo assim, a classe `IndexedJoinPointManager` é também responsável por fazer essa notificação.

A informação reunida por esses observadores deverá ser transmitida para os transformadores, que executarão a instrumentação das sombras afetadas em tempo de execução, como veremos a seguir.

7.3. Hot Swap

Vimos até agora que foi preciso dividir a instrumentação em duas etapas (geração de ganchos e substituição), o que requer uma nova classificação de sombras de junção, e criamos uma arquitetura de observadores de cadeias de adendos. O próximo passo consiste em utilizar toda essa estrutura que foi criada para executar o *hot swap*.

Com as informações recebidas pelos observadores de pilhas de adendos, os transformadores poderão substituir as sombras de junção e reverter essa substituição quando necessário. Como decorrência de uma operação dinâmica, diversas cadeias de adendos podem ser afetadas. Para realizar a instrumentação em tempo de execução de forma atômica, queremos aguardar o término de todas as alterações de cadeias decorrentes de uma operação dinâmica antes de executar a instrumentação. Por esse motivo, a estratégia `DynamicAOPStrategy` possui o método *callback* `interceptorChainsUpdated`. Esse método é chamado somente após a execução de uma operação dinâmica. Quando isso ocorre, a nossa estratégia `HotSwapStrategy` (veja a Figura 7.1 na página 105) reúne as informações coletadas pelos observadores de cadeias para repassá-las aos transformadores. Isso é feito através do método cujo nome também é `interceptorChainsUpdated`, que foi adicionado à fachada `Instrumentor`. Esse método notifica o `Instrumentor` sobre as sombras que passaram a ser interceptadas (ou seja, suas cadeias de adendos deixaram de ser vazias), e sobre as sombras que deixaram de sê-lo (isto é, suas cadeias de adendos passaram a ser vazias). De posse dessa informação, as operações de substituição e reversão são delegadas para os transformadores adequados.

Sabemos que, para a manipulação de *bytecodes*, o JBoss AOP utiliza a biblioteca Javassist [37]. Essa biblioteca armazena informações dos *bytecodes* em estruturas de dados próprias e permite a manipulação dessas estruturas. É interessante notar que não existe uma forma fixa de aplicar essas alterações. É possível sobrescrever os arquivos `.class` com novas versões (compilação através de `aopc`), alterar os *bytecodes* em tempo de carga, ou utilizar o *hot swap*. Dessa forma, como as alterações decorrentes da instrumentação serão efetivadas é totalmente transparente para os transformadores, o que torna a sua implementação menos complexa. Assim, para realizar a instrumentação em tempo de execução, os transformadores fazem as alterações da mesma forma que o fazem durante a transformação inicial do sistema. A notificação dos transformadores funciona como uma reclassificação de sombras de junção. A execução do *hot swap* fica a cargo do próprio método `interceptorChainsUpdated` da classe `HotSwapStrategy`, que deve aplicar as alterações realizadas pelos transformadores. Para tanto, ele extrai essas alterações das estruturas fornecidas pelo Javassist e as repassa para a interface de *hot swap* da JVMTI.

7.4. Solução Final

A arquitetura da solução foi apresentada na Figura 7.1 na página 105. Vimos que a abordagem utilizada para executar a combinação dinâmica foi implementada na forma de estratégias, que devem implementar a interface `DynamicAOPStrategy`.

Existem duas implementações dessa estratégia. Uma delas, `LoadInterceptedClassesStrategy`, mantém a implementação original do JBoss AOP. A outra, `HotSwapStrategy`, possui a nossa implementação de instrumentação em tempo de execução. Uma vez que vimos em detalhes as mudanças necessárias para essa implementação, é fácil concluir o que essa estratégia faz:

- fornece o algoritmo `JoinpointFullClassifier` para ser utilizado pelos transformadores (algoritmo que classifica os pontos de junção em: não instrumentar, apenas gerar ganchos, ou substituir), garantindo que somente as sombras de junção que serão de fato interceptadas serão substituídas;
- fornece observadores que armazenam as informações de quais sombras de junção passaram ou deixaram de ser interceptadas após uma operação de POA dinâmica;
- ao ser notificada da ocorrência de uma operação de POA dinâmica, reúne os dados obtidos por todos os observadores e os envia para o método `Instrumentor.interceptorChainsUpdated`. Efetiva a instrumentação realizada pelos transformadores através do *hot swap* fornecido pela JVMTI.

Veja na Figura 7.3 na próxima página como isso funciona na prática. O diagrama de colaboração ilustra a interação entre as classes em dois momentos: durante a transformação inicial do sistema, quando os transformadores utilizam o `JoinPointFullClassifier` para realizar a instrumentação, e durante uma operação dinâmica, quando o `Instrumentor` invoca o `HotSwapper` para substituir as sombras de junção ao ser notificado pela `HotSwapStrategy` de que as cadeias de interceptadores foram modificadas.

A segunda implementação da estratégia `DynamicAOPStrategy` é a classe `LoadInterceptedClasses-Strategy`. Como sabemos, o objetivo dessa classe é apenas implementar o comportamento visto na Seção 3.1.1 na página 27:

- fornece `JoinpointSimpleClassifier` como algoritmo de classificação de sombras, garantindo que as sombras preparadas serão substituídas por chamadas a ganchos vazios;
- fornece um observador nulo para as cadeias de adendos, uma vez que não é preciso observar essas cadeias nessa abordagem;
- ignora as notificações de ocorrências de operações de POA dinâmica, pois não é preciso tomar nenhuma ação adicional quando elas ocorrem.

Veja agora na Figura 7.4 na página 114 como essa estratégia influencia a execução do JBoss AOP. No diagrama de colaboração, vemos que durante a transformação inicial do sistema os transformadores utilizam o algoritmo de classificação `JoinPointSimpleClassifier`, que forçará a instrumentação completa

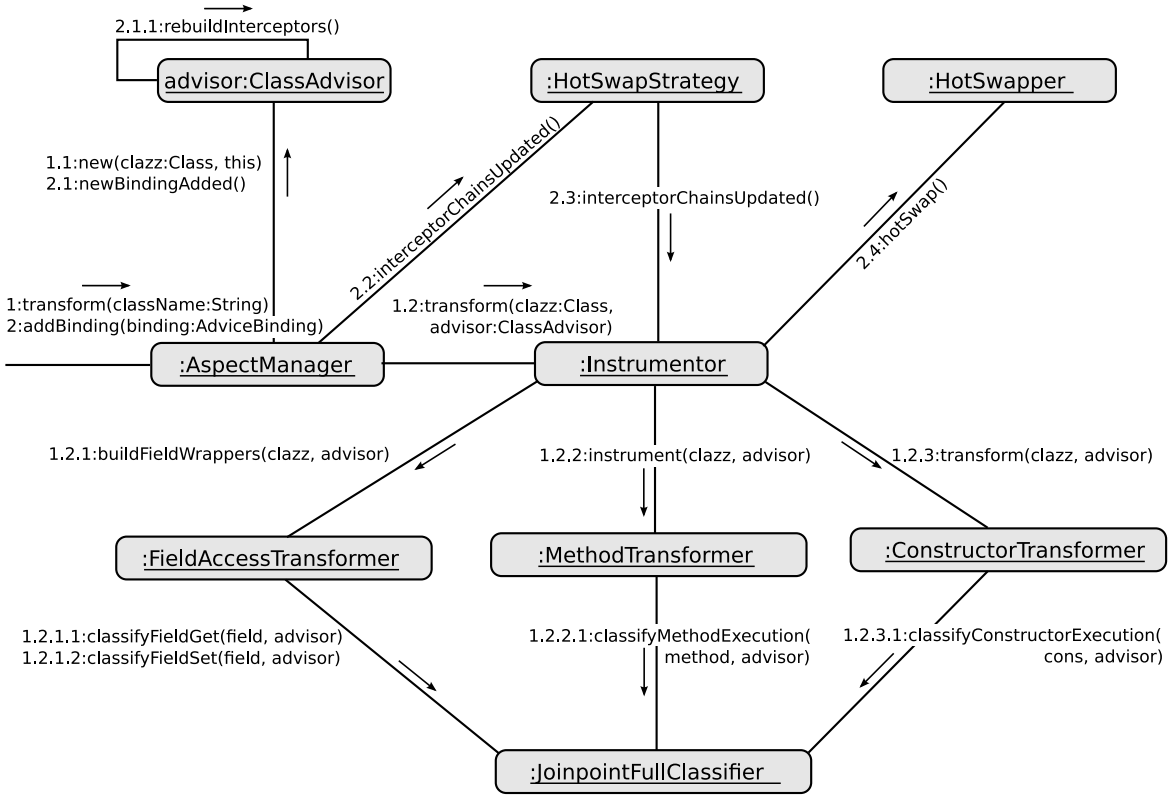


Figura 7.3.: Diagrama de colaboração ilustrando as interações entre as classes no modo de instrumentação em tempo de execução (*hot swap*).

das sombras preparadas. Já durante a execução de uma operação dinâmica, a única ação executada é a atualização de cadeias de adendos. Não há notificações a observadores e a notificação de término da operação, `interceptorChainsUpdated`, feita à estratégia `LoadInterceptedClassesStrategy`, não resulta em nenhuma ação adicional.

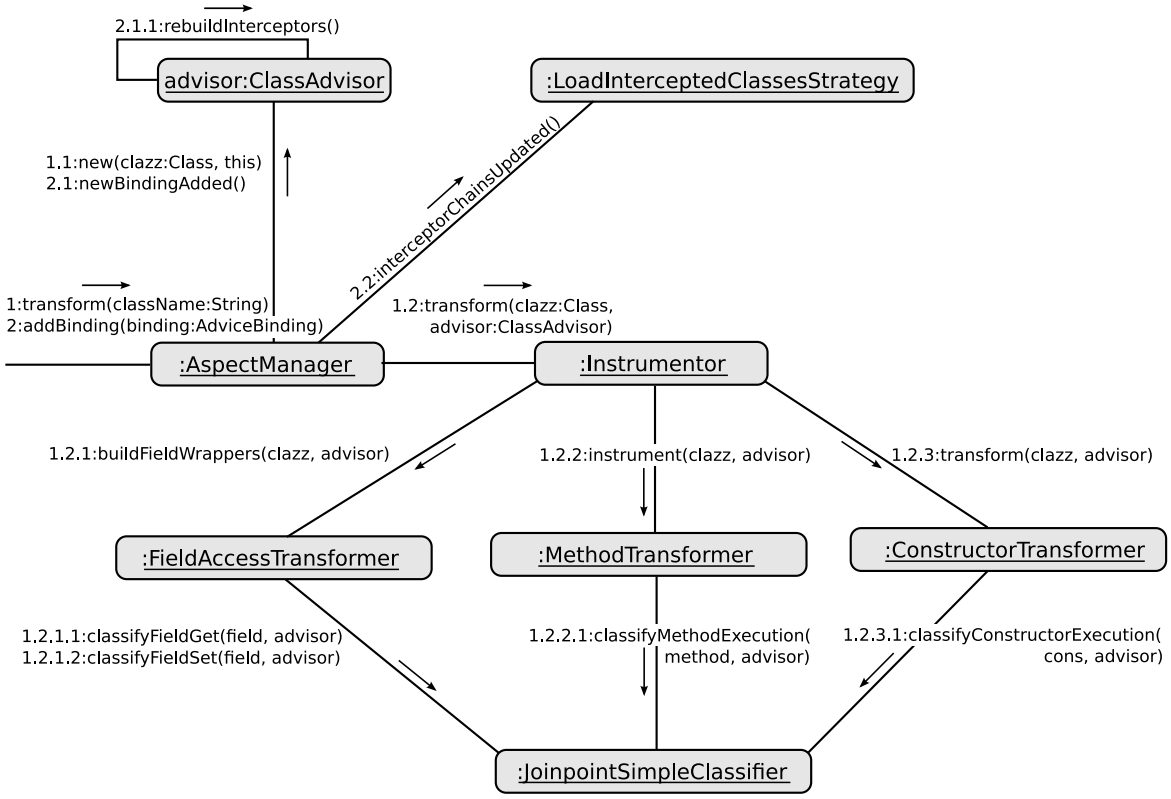


Figura 7.4.: Diagrama de colaboração ilustrando as interações entre as classes no modo de instrumentação tradicional.

8. Avaliação de Desempenho

Realizamos testes de desempenho a fim de quantificar o impacto da abordagem proposta no desempenho da combinação dinâmica de aspectos. Neste capítulo apresentaremos os resultados obtidos e faremos uma breve análise dos mesmos.

Começamos apresentando o ambiente onde os testes foram realizados. Após, apresentaremos os experimentos que conduzimos para avaliar o desempenho da atualização de cadeias seguindo as duas opções disponíveis: a implementação original do JBoss AOP; e a nossa implementação, com o grafo de sombras. Na terceira seção, veremos os testes que avaliam os benefícios da instrumentação em tempo de execução.

8.1. Ambiente de Testes

Todos os testes aqui apresentados foram realizados em um computador com processador Pentium Centrino Core 2 Duo 2.0Ghz e 2GB de memória RAM. O sistema operacional que utilizamos é o Linux, distribuição Fedora 6.

Durante a execução dos testes, o computador não estava operando em modo gráfico, nem possuía conexões com a internet, pois queríamos evitar ao máximo que outros processos interferissem nos nossos resultados.

Os testes foram executados em ambiente Java, onde utilizamos a máquina virtual *Sun Java SE Development Kit* (*JDK*), versão 5.0.

Escolhemos o projeto JHotDraw [40] como caso de teste. O JHotDraw é um arcabouço para o desenvolvimento de interfaces gráficas para usuário que foi criado em 2000, pelo autores Erich Gamma e Thomas Eggenschwiler. No início, se tratava de um exercício de desenho. Com o tempo, esse projeto cresceu na comunidade de *software* livre e hoje é utilizado por outras aplicações [36, 42, 43]. Em 2004, foi criada uma versão refatorada desse projeto utilizando-se aspectos. O objetivo da pesquisa era identificar os cenários comuns em um sistema orientado a objetos que podem ser encapsulados em aspectos. Desde a criação desse projeto, chamado AJHotDraw [1, 22, 23, 56], surgiram diversos trabalhos sobre programação orientada a aspectos que utilizam o JHotDraw e o AJHotDraw como casos de teste [9, 11, 12, 21, 46].

8.2. Atualização de Cadeias de Adendos

Todos os testes da atualização de cadeias de adendos que realizamos consistiram nas três etapas:

- preparar todas as sombras do JHotDraw (chamadas, execuções e acessos a campos);
- forçar a carga das classes de 10 em 10 unidades;
- a cada carga de classes, realizamos 140 adições e remoções dinâmicas de associações.

Cada uma das 140 associações adicionadas possui um *pointcut* diferente. Dentre os *pointcuts* utilizados, é possível encontrar expressões que especificam o tipo alvo (construção *type* da tabela 6.1 na página 72) com e sem curingas. Também utilizamos construções `$instanceof` e expressões do tipo *package* (a sintaxe dessas construções pode ser também encontrada na tabela 6.1). O tipo de sombras de junção com o qual esses *pointcuts* casam é também variado, de modo que incluímos *pointcuts* para casar com todos os tipos suportados pelo JBoss AOP: execução de métodos e construtores, leitura e escrita de campos, e chamadas a métodos e construtores. A lista completa desses *pointcuts* está disponível no Apêndice C na página 160.

Note que cada operação de adição dinâmica é seguida por uma operação de remoção, pois queremos que o número de associações presentes no sistema seja sempre constante durante essas operações.

Realizamos os testes de desempenho de atualização de cadeias em três cenários distintos, cada qual com um número diferente de associações presentes: 0, 6 e 12. Cada uma dessas associações continha 5 adendos, o que implica que os cenários têm 0, 30 e 60 adendos, respectivamente.

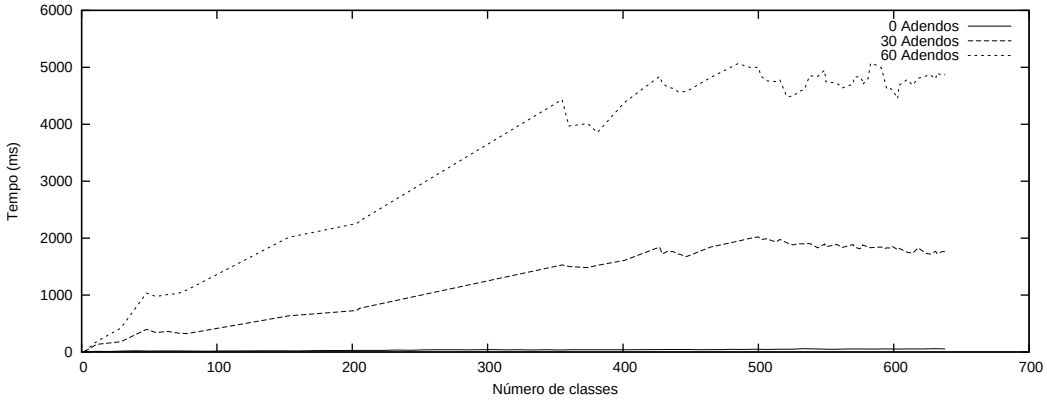
Medimos o tempo decorrido durante a adição e a remoção das associações em todos os cenários, acompanhando como o desempenho varia à medida que o número de classes carregadas cresce.

8.2.1. Adição de Associações

Começamos nosso estudo de desempenho com uma avaliação das operações de adição no modo de casamento padrão (esse é modo habilitado quando a variável de sistema `jboss.aop.matching` é configurada com o valor `"standard"`). O gráfico da Figura 8.1 mostra o desempenho médio dessas operações nos três cenários diferentes: 0, 30 e 60 adendos configurados no sistema.

O modo padrão utiliza a solução de atualização de cadeias descrita na Seção 4.2.2 na página 59. Quando uma associação é adicionada em tempo de execução, esse algoritmo reconstrói todas as cadeias das classes que casam com o *pointcut* da associação (casamento *soft*). Durante a reconstrução de uma cadeia, todos os *pointcuts* presentes no sistema são casados com a sombra correspondente. Quanto mais associações estiverem presentes durante uma operação dinâmica de adição, mais *pointcuts* estarão presentes no sistema, e mais casamentos serão realizados para reconstruir as cadeias de interceptadores das classes envolvidas. À medida que mais classes são carregadas, o impacto que o número de associações presentes implica no desempenho da operação de adição dinâmica se torna mais evidente.

Com os dados da tabela da Figura 8.1b, é possível quantificar esse impacto. Quando há 0 adendos, o tempo de duração médio está entre 0,183 e 55 milissegundos. Porém, ao passarmos para 30 adendos, o desempenho chega a ultrapassar 2000 milissegundos. No cenário com 60 adendos, o custo se torna muito mais alto. Chegamos a ter custo médio superior a 5000 milissegundos para 500 classes carregadas, o que representa um aumento de 11.222,43% em relação ao primero cenário e de 210% em relação ao



(a) Desempenho da adição de associações no modo de casamento padrão (`jboss.aop.matching=standard`).

0 Associações/ 0 Adendos

Classes Carregadas		2	105	203	305	407	507	603	638
Tempo (ms)	Média	0,183	19,278	28,686	42,393	38,839	44,166	50,589	54,727
	Desvio Padrão	0,409	33,724	67,897	104,443	86,483	93,222	112,132	123,651
	Mínimo	0,009	0,806	1,227	1,796	1,111	0,555	0,657	0,669
	Máximo	8,445	482,155	855,424	1105,519	993,272	763,499	829,088	1296,715

6 Associações/ 30 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	5,216	322,028	727,655	1530,512	1609,095	2020,208	1796,850	1761,417
	Desvio Padrão	12,172	432,110	1016,333	2227,884	2329,264	1981,290	2603,519	2471,450
	Mínimo	0,029	0,370	0,906	1,620	1,721	1,330	0,697	0,708
	Máximo	80,026	1586,485	3423,263	9873,709	8698,776	12198,864	9402,761	8600,213
% Maior que 0 Adendos:		2750,27	1570,44	2436,62	3510,29	4042,99	4474,12	3451,86	3118,55

12 Associações/ 60 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	7,384	1090,690	2255,715	4433,729	4381,019	5000,666	4451,749	4873,016
	Desvio Padrão	11,960	1484,845	3215,629	6125,422	6178,586	6920,742	6179,134	6949,742
	Mínimo	0,031	0,379	0,936	1,403	1,647	0,953	0,735	0,757
	Máximo	48,636	5572,566	11553,697	19431,130	23146,432	24352,746	20853,404	24033,782
% Maior que 0 Adendos:		3934,97	5557,69	7763,47	10358,63	11179,95	11222,43	8699,84	8804,23

(b) Duração média das operações de adição dinâmica no modo `jboss.aop.matching:standard`. Todos os tempos são expressos em milissegundos.

Figura 8.1.: Tempo médio de duração das operações de adição dinâmica no modo `jboss.aop.matching:standard`.

segundo cenário. Note que estamos nos referindo ao tempo médio, apenas. Na Figura 8.1b, é possível ver que uma única operação chega a durar mais de 24 segundos para completar.

Quando os testes são executados no modo de casamento indexado, por outro lado, o número de associações presentes no sistema não causa impacto no desempenho das operações de adição dinâmica. Na solução proposta pelo Capítulo 5 na página 61, somente adicionamos interceptadores às cadeias afetadas após realizar uma busca em gráfico. Portanto, se, durante a adição, houver outras associações no sistema que afetam essas cadeias, não haverá alteração no desempenho. Esse fato pode ser comprovado no gráfico da Figura 8.2a na página seguinte, que ilustra o desempenho das operações de adição em três cenários com número variável de adendos. Como é possível notar, não há variações significativas quando o número de associações presentes no sistema aumenta.

No gráfico da Figura 8.3 na página 120 é possível notar que o impacto principal de uma adição no modo indexado é causado pela busca no grafo. Esse gráfico nos permite concluir que não há operações auxiliares impactantes, de modo que o desempenho da busca do grafo se reflete significativamente no desempenho da operação de adição dinâmica como um todo. Isso valida o grafo como um dos principais focos de nossa atenção no desenvolvimento desse trabalho.

Ao compararmos as tabelas da Figuras 8.1b e 8.2b, vemos que a independência do modo indexado quanto ao número de associações existentes resulta em uma diferença imensa no desempenho. Quando todas as classes estão carregadas, o modo indexado tem desempenho médio $96,4\%$ superior ao modo padrão no cenário com 60 adendos. No cenário com 30 adendos, a superioridade do modo indexado se mantém, cujo desempenho é $92,7\%$ melhor do que no modo padrão quando todas as classes foram carregadas. No entanto, ao compararmos o desempenho médio no cenário com 0 adendos, é possível observar que o modo de casamento indexado possui desempenho médio inferior ao desempenho do modo de casamento padrão na ausência de adendos. Note que o cenário com 0 adendos é o melhor caso do modo de casamento padrão. O gráfico da Figura 8.4 na página 120 evidencia essa diferença.

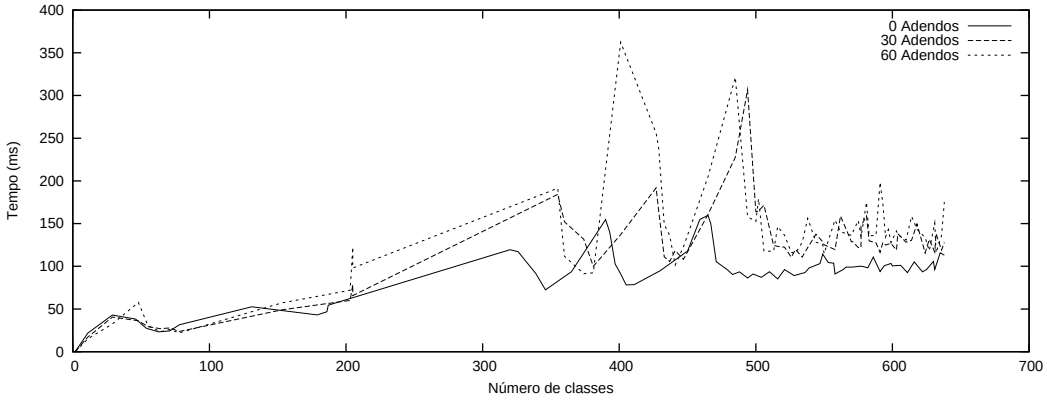
Apesar do que vemos na Figura 8.4, o resultado obtido não significa que a nossa proposta é menos eficiente que a solução anterior no cenário com 0 adendos. Há casos de busca no grafo que são consideravelmente mais lentos, o que afeta a duração média das operações de adição. Esses casos de busca são aqueles que envolvem expressões com construções do tipo `call`. Nesses cenários, é possível afirmar que o modo indexado que propomos neste trabalho não conseguiu superar o desempenho do modo de casamento padrão, como veremos adiante.

8.2.1.1. Desempenho por Tipo de Pointcuts

Nesta seção dividimos o resultado em categorias, conforme o tipo de *pointcuts*. Classificamos os *pointcuts* em três tipos: acesso a campos (*pointcuts* `field`), execução (*pointcuts* `execution`) e chamada (*pointcuts* `call`). Veremos que os resultados obtidos com o grafo para cada um desses tipos foi diferente.

Começamos avaliando o desempenho das operações de adição que envolvem *pointcuts* do tipo acesso a campos. Essas operações tiveram desempenho superior no modo indexado, mesmo quando comparamos com o melhor caso do modo padrão, o cenário com 0 adendos.

Dentre as expressões do tipo acesso a campos, há um grupo que teve um desempenho visivelmente



(a) Desempenho da adição de associações no modo `jboss.aop.matching:indexed`.

0 Associações/ 0 Adendos

Classes Carregadas		2	78	187	320	401	504	600	638
Tempo (ms)	Média	0,400	31,497	54,447	119,483	91,185	87,063	100,583	113,101
	Desvio Padrão	0,641	54,772	108,807	229,920	171,565	145,339	178,376	235,340
	Mínimo	0,058	0,049	0,049	0,090	0,100	0,093	0,089	0,095
	Máximo	14,838	456,398	687,319	1707,609	1342,266	1236,983	1740,428	2385,482

6 Associações/ 30 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	0,410	23,755	60,098	184,306	137,148	163,858	118,921	128,102
	Desvio Padrão	0,386	33,878	123,935	347,126	264,915	313,311	191,653	252,853
	Mínimo	0,062	0,048	0,057	0,104	0,108	0,159	0,155	0,155
	Máximo	2,201	189,509	772,218	1541,435	1652,636	1630,206	947,746	2208,812
% Maior que 0 Adendos:		2,5	-24,58	10,38	54,25	50,41	88,21	18,23	13,26

12 Associações/ 60 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	0,388	21,939	71,209	191,422	148,912	152,588	141,239	175,214
	Desvio Padrão	0,347	29,903	160,828	403,720	367,888	386,107	335,463	447,782
	Mínimo	0,055	0,051	0,077	0,113	0,127	0,124	0,114	0,097
	Máximo	1,982	138,144	1216,453	2206,402	3064,445	3194,360	3444,713	2913,536
% Maior que 0 Adendos:		-3	-30,35	30,79	60,21	63,31	75,26	40,42	54,92

(b) Duração média das operações de adição dinâmica no modo `jboss.aop.matching:indexed`. Todos os tempos são expressos em milissegundos.

Figura 8.2.: Tempo médio de duração das operações de adição dinâmica no modo de casamento indexado (`jboss.aop.matching=indexed`).

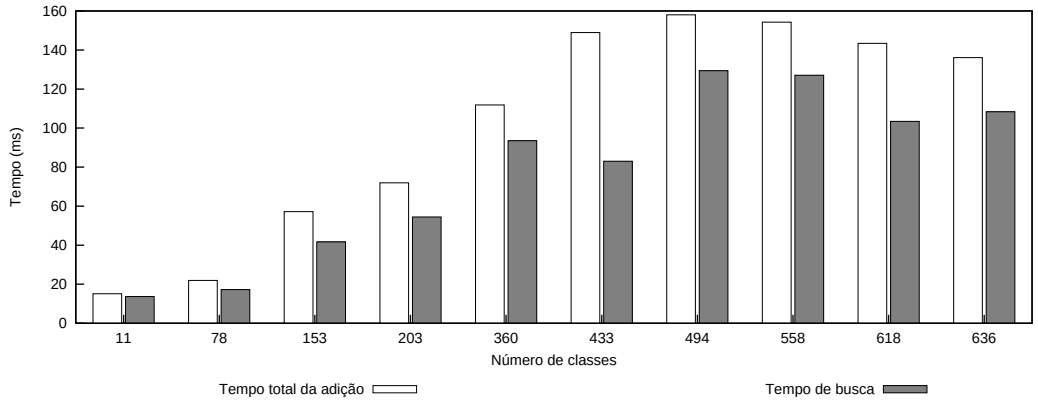


Figura 8.3.: Tempo de duração médio de operações de adição no modo indexado comparado com a parcela da duração da busca no grafo.

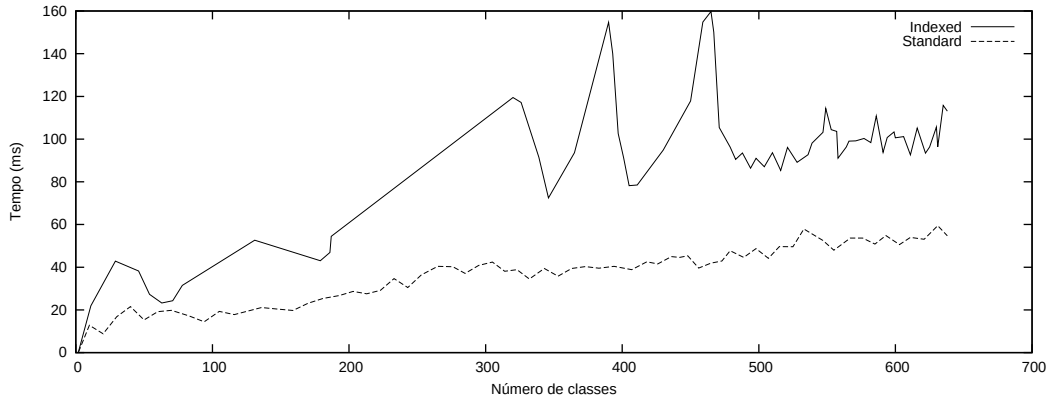


Figura 8.4.: Comparação do desempenho médio das operações de adição modo indexado com o modo padrão (no cenário com 0 adendos).

inferior. Esse é o grupo que possui construções `$instanceof{*}` e `$instanceof{java.lang.Object}`. Essas construções são ambas equivalentes a utilizar um curinga `'*'` para denotar os tipos alvo nos *pointcuts*. Apesar de ser improvável um usuário utilizar tais expressões (o uso do curinga é muito mais claro e direto), achamos válido incluir *pointcuts* nesse formato porque ele representa o pior caso de busca na árvore de nós `ClassNode`. Expressões com a construção `$instanceof` resultam em um nível extra de busca no grafo: não é suficiente fazer uma busca na árvore de nós `ClassNode`; é preciso também percorrer as subclasses das classes encontradas de maneira recursiva, replicando a busca nesses nós. Isso resulta em uma sobrecarga, sendo que o pior caso é quando temos que encontrar todas as subclasses de todas as classes, o que equivale às construções `$instanceof{*}` e `$instanceof{java.lang.Object}`.

Os gráficos da Figura 8.5 comparam o desempenho médio da adição de associações com *pointcuts* desse tipo nos modos indexado e padrão (ambos no cenário com 0 associações). Dividimos os *pointcuts*

dessa classificação em dois grupos: com e sem expressões `$instanceof`. Vemos nos gráficos que o modo indexado apresenta um desempenho visivelmente superior, mesmo na presença de construções `$instanceof`. Os dados da Figura 8.5 também mostram que há uma visível sobrecarga com a presença de construções `$instanceof`, tanto no modo de casamento padrão quanto no modo indexado.

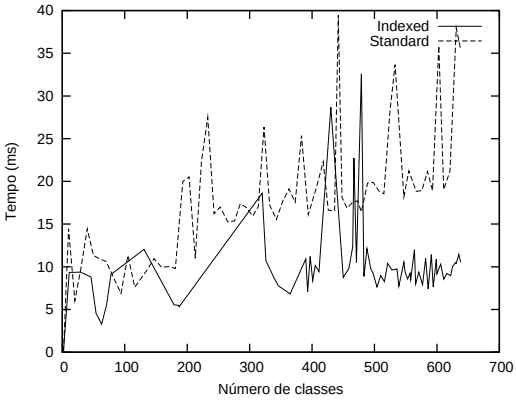
O desempenho médio de *pointcuts* do tipo execução em função do número de classes carregadas pode ser visto na Figura 8.6. Pelos mesmos motivos apresentados anteriormente, dividimos esses *pointcuts* em dois grupos, um sem construções `$instanceof` e outro com essas construções. Vemos na Figura 8.6 que eles apresentam desempenho também superior aos cenários respectivos no modo de casamento padrão. Novamente, é possível ver que as construções `$instanceof` causam impacto nos dois modos de casamento, padrão e indexado.

Ao compararmos as tabelas das Figuras 8.5b e 8.6b, podemos notar que a busca por execuções é um pouco menos eficiente que a busca por acesso a campos. Essa pequena diferença se deve ao tamanho das subárvores de campos `FieldNode`. Essas subárvores contêm um número de nós menor do que as subárvores de `BehaviorNode`, visto que o número de campos presentes nas classes do `JHotDraw` é visivelmente inferior ao número de construtores e métodos.

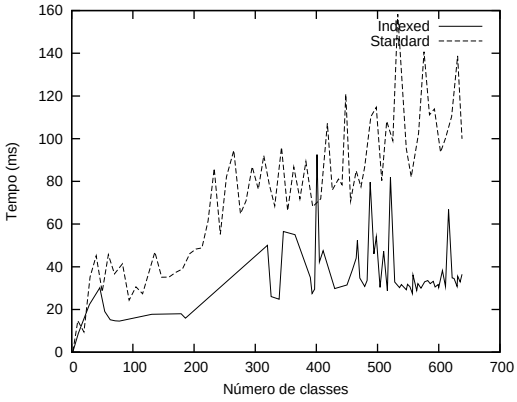
Até o momento, estudamos casos em que o modo de casamento indexado pelo grafo superou o modo padrão em termos de desempenho. Assim, o motivo do desempenho inferior visto no gráfico da Figura 8.4 na página precedente está no desempenho dos cenários que envolvem o terceiro e último tipo de *pointcuts*: chamada. A causa desta queda no desempenho se encontra na própria estrutura do grafo. Para realizar uma busca por chamadas o grafo executa diversas buscas em três níveis do grafo: a subárvore de nós `ClassNode`, as subárvores de nós `BehaviorNode`; e as subárvores de chamadas.

Os gráficos da Figura 8.7 ilustram a diferença que existe no desempenho da adição de associações envolvendo *pointcuts* do tipo chamada. Nos gráficos da figura, é possível observar como o modo padrão é mais eficiente do que o casamento indexado quando as associações possuem um *pointcut* desse tipo. Observe que dividimos também os *pointcuts* de chamada em dois grupos: um é formado por *pointcuts* simples, que possuem somente a expressão `call`; o outro é o grupo de *pointcuts* compostos, formados por uma construção `call` seguida por uma construção restritiva `within` ou `withincode`. Há uma visível melhora no desempenho quando adicionamos essas construções restritivas.

Essa diferença é também causada pela estrutura do grafo. Ao contrário do que se poderia imaginar, a construção `call` é vista como auxiliar na busca, já que primeiro o grafo busca pelos métodos e construtores que fizeram a chamada para depois encontrar as chamadas que eles realizaram. Assim, são as expressões `within` e `withincode` que determinam a chave de busca por um objeto `BehaviorNode`, enquanto que as expressões `call` são utilizadas dentro dos nós `BehaviorNode` para encontrar chamadas a construtores e a métodos. Quando não há restrições do tipo `within` e `withincode`, o desempenho da busca no grafo é inferior ao desempenho da busca pelos demais tipos de *pointcut*. O grafo irá buscar por todos os nós `ClassNode` na árvore de classes. Para cada nó encontrado, uma busca por todos os métodos e construtores será efetuada na subárvore de métodos e construtores desse nó. Finalmente, para cada nó `BehaviorNode` encontrado nessa busca, o grafo fará uma busca utilizando a expressão contida dentro da construção `call` do *pointcut*. Isso significa que, quando não há restrições indicando quais os métodos e construtores que fizeram a chamada, o número de buscas nas subárvores de chamadas



(a) Desempenho da adição de associações com *pointcuts* de acesso a campos.



(b) Desempenho da adição de associações com *pointcuts* de acesso a campos que utilizam a construção `$instanceof`.

Modo de casamento padrão (`jboss.aop.matching=standard`)

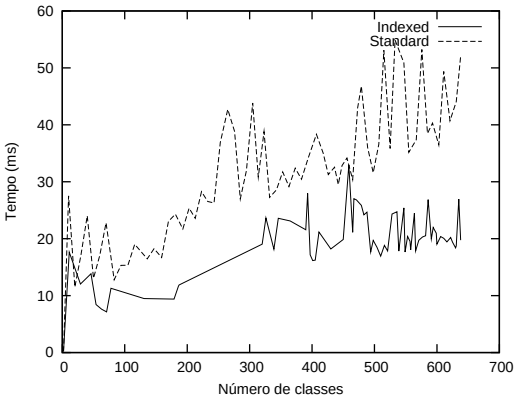
		pointcut `field` **sem** `$instanceof`				*pointcut* `field` **com** `$instanceof`			
Classes Carregadas		2	203	407	603	2	203	407	603
Tempo (ms)	**Média**	0,112	20,524	19,279	35,860	0,167	48,264	71,810	93,665
	Desvio Padrão	0,065	69,245	40,468	111,970	0,082	69,763	92,264	125,985
	Mínimo	0,033	1,227	2,352	0,805	0,078	2,669	5,595	5,818
	Máximo	0,290	509,917	191,949	754,538	0,321	370,438	449,425	608,392

Modo de casamento indexado (`jboss.aop.matching:indexed`)

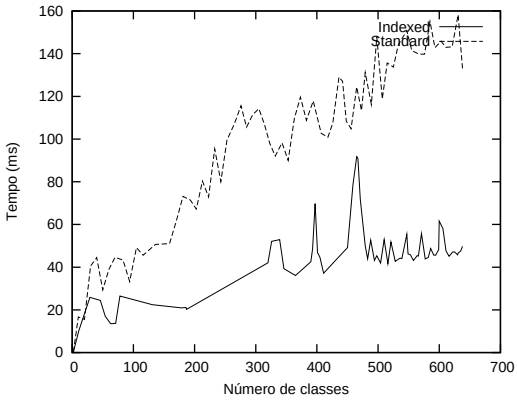
		pointcut `field` **sem** `$instanceof`				*pointcut* `field` **com** `$instanceof`			
Classes Carregadas		2	187	405	600	2	187	405	600
Tempo (ms)	**Média**	0,159	5,304	10,128	9,178	0,210	16,168	42,201	30,306
	Desvio Padrão	0,085	13,407	30,329	24,572	0,145	22,771	126,030	47,660
	Mínimo	0,058	0,049	0,095	0,089	0,062	0,056	0,584	0,557
	Máximo	0,410	75,794	169,595	100,526	0,529	88,163	791,250	166,371

(c) Tabela comparativa sobre o desempenho da adição de associações com *pointcuts* de acesso a campos.

Figura 8.5.: Desempenho da adição de associações com *pointcuts* de acesso a campos nos modos de casamento padrão e indexado.



(a) Desempenho da adição de associações com *pointcuts* de execução.



(b) Desempenho da adição de associações com *pointcuts* de execução que utilizam a construção `$instanceof`.

Modo de casamento padrão (`jboss.aop.matching=standard`)

		pointcut execution **sem** $instanceof				*pointcut* execution **com** $instanceof			
Classes Carregadas		2	203	407	603	2	203	407	603
Tempo (ms)	**Média**	0,163	24,360	38,342	36,456	0,247	67,222	102,646	145,802
	Desvio Padrão	0,167	78,243	112,184	84,726	0,172	87,429	148,374	207,957
	Mínimo	0,033	1,495	2,420	0,897	0,077	3,245	5,388	5,568
	Máximo	0,937	855,424	993,272	531,178	0,831	325,047	860,085	829,088

Modo de casamento indexado (`jboss.aop.matching:indexed`)

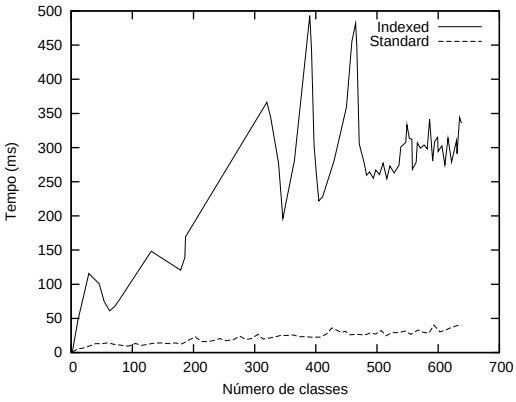
		pointcut execution **sem** $instanceof				*pointcut* execution **com** $instanceof			
Classes Carregadas		2	187	405	600	2	187	405	600
Tempo (ms)	**Média**	0,225	11,878	16,231	19,039	0,284	20,232	44,638	61,577
	Desvio Padrão	0,161	53,450	44,998	58,460	0,230	36,856	107,534	185,658
	Mínimo	0,070	0,140	0,163	0,135	0,070	0,183	0,209	0,189
	Máximo	0,922	620,056	240,498	357,771	1,134	181,158	785,750	1632,248

(c) Tabela comparativa sobre o desempenho da adição de associações com *pointcuts* de execução.

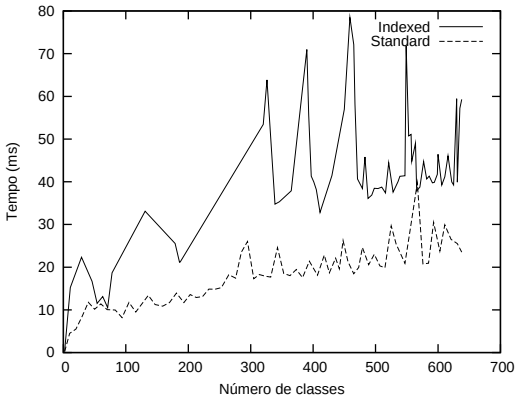
Figura 8.6.: Desempenho da adição de associações com *pointcuts* de execução nos modos de casamento padrão e indexado.

dos nós `BehaviorNode` é consideravelmente maior. Essa diferença pode ser vista ao compararmos os dois gráficos da Figura 8.7.

Idealmente, gostaríamos de fazer uma busca direta pelo nó `BehaviorNode` que representa o método ou construtor chamado quando não houver restrições ***`within`*** e ***`withincode`*** em um *pointcut* de chamada. A partir dos nós encontrados nessa busca, obteríamos as sombras que realizam as chamadas em tempo constante, sem a necessidade de realizar mais uma busca. O motivo pelo qual o grafo não se comporta dessa maneira é o suporte a domínios. A estrutura atual garante que cada objeto `AdvisedData` contém apenas dados sobre sombras pertencentes a um domínio. Uma vez que uma sombra de chamada pertence ao domínio do objeto chamador, um nó `BehaviorNode` deve conter as sombras das chamadas que realiza, ao invés de conter as sombras das chamadas feitas ao construtor ou método que representa.



(a) Desempenho da adição de associações com *pointcuts* de chamada.



(b) Desempenho da adição de associações com *pointcuts* simples de chamada.

Modo de casamento padrão (`jboss.aop.matching=standard`)

		pointcut `call` **sem** `within/withincode`				*pointcut* `call` **com** `within/withincode`			
Classes Carregadas		2	203	407	603	2	203	407	603
Tempo (ms)	**Média**	0,178	22,945	22,705	30,319	0,198	13,580	18,101	23,770
	Desvio Padrão	0,465	68,012	20,277	60,490	0,639	17,931	21,270	34,863
	Mínimo	0,033	1,364	2,411	0,676	0,036	1,586	1,111	0,657
	Máximo	6,151	537,497	69,812	753,761	8,445	109,713	95,699	183,887

Modo de casamento indexado (`jboss.aop.matching:indexed`)

		pointcut `call` **sem** `within/withincode`				*pointcut* `call` **com** `within/withincode`			
Classes Carregadas		2	187	405	600	2	187	405	600
Tempo (ms)	**Média**	0,859	169,318	222,031	294,323	0,283	21,212	38,162	46,429
	Desvio Padrão	1,099	153,715	159,446	199,423	0,310	52,430	91,381	116,085
	Mínimo	0,436	40,941	86,838	109,124	0,064	0,054	0,115	0,098
	Máximo	14,838	687,319	1183,544	1690,789	1,431	447,493	705,582	993,614

(c) Tabela comparativa sobre o desempenho da adição de associações com *pointcuts* de chamada compostos com uma restrição `within`/`withincode`.

Figura 8.7.: Desempenho da adição de associações com *pointcuts* de chamada nos modos de casamento padrão e indexado.

8.2.2. Remoção de Associações

O gráfico da Figura 8.8 ilustra o desempenho médio das operações de remoção no modo de casamento padrão. A operação de remoção nesse modo é consideravelmente mais veloz do que as operações de adição. O motivo dessa diferença é que não há casamento *soft* envolvido nessa operação. Como mostra o diagrama da Figura 4.5 na página 60, o JBoss AOP utiliza a coleção de *advisors* contida no `AdviceBinding` a ser removido para decidir quais classes terão as suas cadeias reconstruídas. Exceto por isso, a operação de remoção tem o mesmo algoritmo que a de adição, onde todas as cadeias dos *advisors* são reconstruídas. Assim, a diferença entre essa figura e a Figura 8.1 na página 117 evidencia o impacto que o casamento *soft* pode causar no algoritmo de atualização de cadeias no modo de casamento padrão.

Vemos na tabela da Figura 8.8b que a remoção, assim como a adição, não escala bem quando aumentamos o número de adendos. Essa é uma característica do modo de casamento padrão, que reconstrói as cadeias durante essas operações, casando todos os *pointcuts* no sistema com as sombras de cada uma dessas cadeias.

O gráfico da Figura 8.9 na página 127 ilustra a remoção no modo indexado. Esse modo remove os interceptadores criados por uma associação sem realizar operações de busca ou de reconstrução de cadeias inteiras (veja o diagrama da Figura 5.3 na página 68).

Apesar de não reconstruir cadeias, esse algoritmo sofre um impacto à medida que aumenta o número de associações no sistema. Essa diferença pode ser vista no gráfico da Figura 8.9. A causa de tal diferença está no estado das cadeias quando da execução de interceptadores. Essas cadeias são objetos `ArrayList` de Java e, para a remoção, utilizamos chamadas ao método `removeAll`. Dada a natureza das listas, a remoção feita em cadeias que só contêm os adendos a serem removidos deve ser mais rápida do que em cadeias que possuem diversos outros interceptadores. Apesar dessa diferença, é possível notar que o modo de casamento indexado não sofre alterações bruscas entre os três cenários. O mesmo não pode ser afirmado em relação ao modo padrão. A tabela da Figura 8.8b mostra que a presença de 60 adendos no sistema pode causar um impacto de mais de 26.000% no desempenho da operação de remoção, quando comparada com a mesma operação na ausência de adendos.

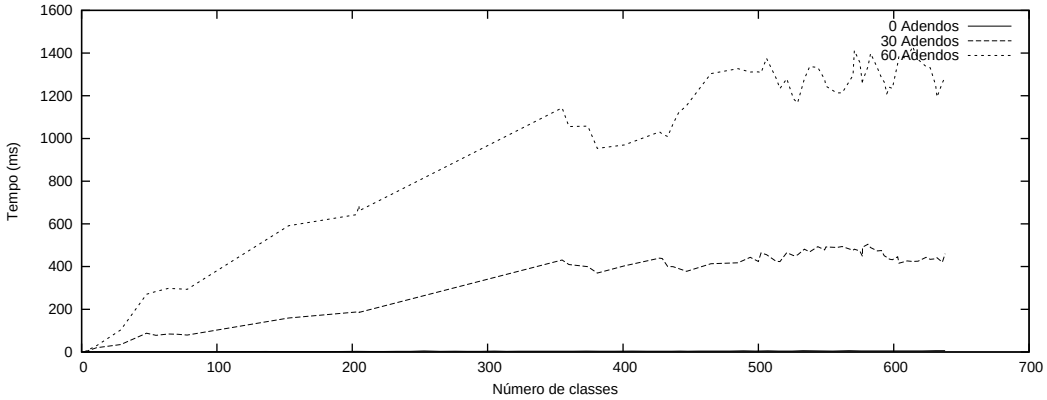
8.2.3. Custo adicional do Modo de Casamento Indexado

Até o momento vimos que o modo de casamento indexado tem desempenho superior ao modo padrão sempre que houver associações presentes no sistema. Contudo, há um custo envolvido no modo indexado. Como esse modo utiliza o grafo, temos o custo adicional do espaço que o grafo ocupa na memória, bem como o custo do tempo de inserção.

Quanto ao custo adicional do espaço que o grafo ocupa na memória, os nossos testes mostraram que o grafo ocupa 17,6 MB quando as 638 classes tiveram as suas sombras carregadas no grafo. Considerando-se que estamos incluindo sombras de chamadas além de execuções e acessos a campos, acreditamos que esse número é satisfatório, já que representa em média menos de 5% do total de memória ocupado pela nossa aplicação de testes no instante em que fizemos a medida (após as 638 classes terem sido carregadas).

Em relação ao tempo de duração de inserção de sombras no grafo, a tabela na página 128 mostra o tempo médio de duração dessa operação para os três tipos de sombras: leitura ou escrita de campo, execução de método ou construtor; e chamada a método ou construtor. Note que dividimos essas sombras em duas categorias: com e sem inserção de classes. Quando o grafo executa a inserção de uma sombra cuja classe à qual ela pertence já está no grafo, dizemos que essa operação ocorreu sem inserção de classe. Essas são as inserções de sombras que mostramos nas três primeiras colunas da tabela abaixo. As colunas restantes, na categoria "com inserção de classe", mostram o tempo da duração de adição de sombras quando foi preciso inserir no grafo também um `ClassNode` representando a classe à qual a sombra pertence.

Nessa tabela vemos que, em média, as operações sem inserção de classe ocorrem em menos de 1



(a) Desempenho da remoção de associações no modo de casamento padrão (`jboss.aop.matching=standard`).

0 Associações/ 0 Adendos

Classes Carregadas		2	105	203	305	407	507	603	638
Tempo (ms)	Média	0,020	1,750	3,208	4,465	4,086	5,440	5,062	6,305
	Desvio Padrão	0,041	6,348	20,400	26,403	13,107	29,194	15,699	32,913
	Mínimo	0,009	0,010	0,011	0,010	0,010	0,010	0,011	0,010
	Máximo	0,338	127,181	506,109	632,577	143,292	685,210	187,123	760,021

6 Associações/ 30 Adendos

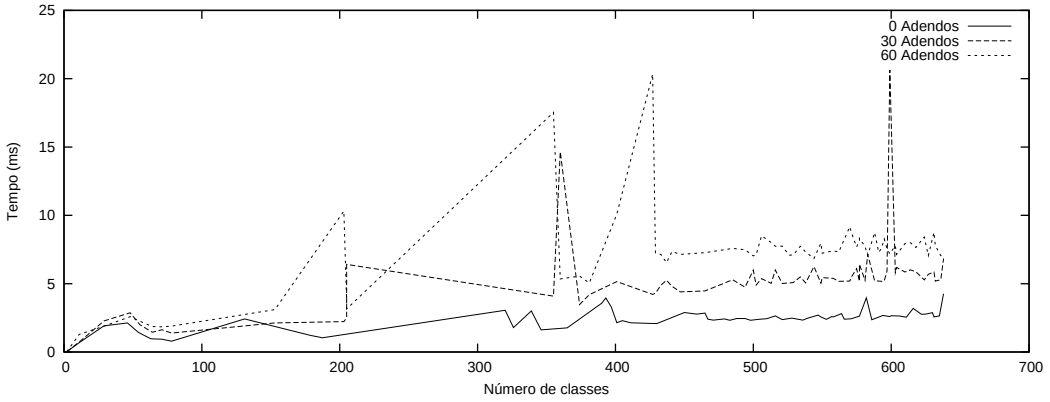
Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	0,197	79,421	187,527	430,635	403,525	423,745	445,901	460,273
	Desvio Padrão	0,848	224,757	548,806	1277,974	1277,295	1222,693	1308,468	1340,613
	Mínimo	0,009	0,010	0,010	0,011	0,011	0,009	0,010	0,010
	Máximo	6,027	1018,279	2748,450	7574,400	8269,295	6347,608	6946,892	6543,327
% Maior que 0 Adendos:		885	4438,34	5745,6	9544,68	9775,8	7689,43	8708,79	7200,13

12 Associações/ 60 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	0,686	293,323	643,205	1143,295	969,900	1312,028	1350,080	1289,066
	Desvio Padrão	3,337	850,299	1889,572	3400,367	2774,492	3930,434	4046,775	3893,053
	Mínimo	0,009	0,010	0,010	0,010	0,011	0,010	0,010	0,010
	Máximo	28,236	3889,726	8483,889	15644,999	11779,734	21760,190	19806,460	20569,215
% Maior que 0 Adendos:		3330	16661,31	19950,03	25505,71	23637,15	24018,16	26570,88	20345,14

(b) Duração média das operações de remoção dinâmica no modo de casamento padrão (`jboss.aop.matching=standard`). Todos os tempos são expressos em milissegundos.

Figura 8.8.: Tempo médio de duração das operações de remoção dinâmica no modo de casamento padrão (`jboss.aop.matching=standard`).



(a) Desempenho da remoção de associações no modo de casamento indexado (`jboss.aop.matching=indexed`).

0 Associações/ 0 Adendos

Classes Carregadas		2	78	187	320	401	504	600	638
Tempo (ms)	Média	0,018	0,794	1,030	3,050	2,148	2,388	2,652	4,269
	Desvio Padrão	0,013	2,026	2,784	9,293	8,445	9,160	10,029	36,321
	Mínimo	0,012	0,014	0,011	0,016	0,014	0,014	0,012	0,015
	Máximo	0,117	28,354	23,312	116,890	159,975	148,251	129,995	903,079

6 Associações/ 30 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	0,024	1,384	2,224	4,088	5,156	5,996	5,815	6,844
	Desvio Padrão	0,028	3,649	5,443	12,801	17,395	23,353	19,656	24,292
	Mínimo	0,012	0,014	0,023	0,016	0,016	0,014	0,016	0,015
	Máximo	0,206	29,328	31,517	110,386	123,056	224,350	128,020	192,045
% Maior que 0 Adendos:		33,33	74,31	115,92	34,03	140,04	151,09	119,27	60,32

12 Associações/ 60 Adendos

Classes Carregadas		2	78	203	355	401	500	603	638
Tempo (ms)	Média	0,024	1,894	3,652	5,343	10,076	7,039	7,779	7,040
	Desvio Padrão	0,027	5,028	9,536	18,072	39,980	24,025	28,016	23,244
	Mínimo	0,012	0,015	0,015	0,016	0,016	0,015	0,017	0,016
	Máximo	0,190	31,437	71,083	135,775	280,453	177,998	223,285	141,096
% Maior que 0 Adendos:		33,33	138,54	254,56	75,18	369,09	194,77	193,33	64,91

(b) Duração média das operações de remoção dinâmica no modo de casamento indexado (`jboss.aop.matching=indexed`). Todos os tempos são expressos em milissegundos.

Figura 8.9.: Tempo médio de duração das operações de remoção dinâmica no modo de casamento indexado (`jboss.aop.matching=indexed`).

		Sombras Registradas No Grafo					
		sem inserção de classe			com inserção de classe		
		Acesso a Campo	Execução	Chamada	Acesso a Campo	Execução	Chamada
Tempo (ms)	Média	0.037	0.092	0.883	3.696	2.334	-
	Desvio Padrão	2.121	4.296	21.065	27.527	19.481	-
	Mínimo	0.007	0.016	0.023	0.046	0.059	-
	Máximo	1800.867	1978.201	1988.409	1815.482	1506.810	-

milissegundo. Já as operações com inserção de classe levam mais tempo mas, ainda assim, o seu tempo médio não ultrapassa 4 milissegundos.

8.3. Instrumentação em Tempo de Execução

Para realizar os testes de instrumentação em tempo de execução escolhemos uma das aplicações exemplo inclusas no projeto JHotDraw para medir o desempenho de algumas operações. Preparamos todas as execuções de método e construtor dessa aplicação, bem como todas as leituras e escritas de campo. Comparamos o desempenho dessas operações com o *hot swap* habilitado e desabilitado, comprovando o impacto que a chamada de ganchos vazios pode causar. Também realizamos as mesmas medidas na aplicação isolada, sem o uso do JBoss AOP, verificando que o desempenho com o *hot swap* habilitado é similar ao desempenho da aplicação isolada.

A aplicação que escolhemos para testar é o SVG, um editor de vetores gráfico que possui operações simples de criação de objetos elipse, retângulo, entre outros. Medimos o tempo das seguintes operações:

- criação de objeto retângulo;
- criação de objeto elipse;
- criação de um segmento de vetor *path*;
- criação de uma caixa de texto;
- uma operação de cópia de um retângulo;
- uma operação de recorte de um retângulo;
- uma operação de cola de um retângulo.

Os objetos retângulo e elipse foram criados com um clique duplo, de modo que a interação com o usuário tivesse o menor impacto possível nas medições realizadas. Por esse motivo, todos os objetos criados têm sempre a mesma dimensão padrão, atribuída pelo SVG. Da mesma forma, os objetos de texto possuem sempre o mesmo texto na criação. Quanto à criação segmentos de vetor, essa foi feita com um único clique em uma posição qualquer do espaço disponibilizado pelo documento tamanho padrão do SVG. Medimos cada uma dessas operações 50 vezes.

As operações de cópia, recorte e cola, foram realizadas em duas etapas. A primeira delas consistiu em copiar um objeto e colá-lo 50 vezes. A segunda etapa, em recortar um objeto retângulo e colá-lo 50 vezes. Por esse motivo, a operação de cola foi medida 100 vezes enquanto que as de cópia e cola foram medidas 50 vezes cada uma.

	Retângulo	Elipse	Texto	Seg. de Vetor	Recortar	Copiar	Colar
Média	4,315	3,581	1,780	3,419	14,946	7,007	7,125
D.P.	1,295	0,247	0,732	0,130	3,346	0,991	2,858
Mínimo	0,356	3,212	1,490	3,242	9,636	3,380	2,965
Máximo	9,462	4,595	5,303	3,792	33,941	9,798	20,502
I.C.	[3,956, 4,674]	[3,513, 3,650]	[1,577, 1,983]	[3,383, 3,455]	[14,019, 15,873]	[6,732, 7,282]	[6,579, 7,672]

(a) Tabela de referência. Mostra o desempenho de operações no SVG sem JBoss AOP.

	Retângulo	Elipse	Texto	Seg. de Vetor	Recortar	Copiar	Colar
Média	4,481	3,721	1,773	3,532	15,648	7,147	7,317
D.P.	1,336	0,449	0,565	1,061	3,220	1,571	3,894
Mínimo	0,348	2,381	1,529	0,602	10,718	3,380	3,049
Máximo	8,614	5,925	5,542	9,537	25,616	15,470	34,283
I.C.	[4,111, 4,851]	[3,597, 3,845]	[1,616, 1,929]	[3,238, 3,826]	[14,755, 16,540]	[6,711, 7,582]	[6,576, 8,058]
Referência	3,84 %	3,89 %	-0,42 %	3,29 %	4,69 %	1,99 %	2,68 %

(b) Desempenho de operações do SVG com sombras preparadas com *hot swap*.

	Retângulo	Elipse	Texto	Seg. de Vetor	Recortar	Copiar	Colar
Média	8,276	6,276	2,061	6,275	19,046	9,202	14,335
D.P.	2,405	0,314	0,131	0,132	1,874	1,320	9,373
Mínimo	0,751	5,633	1,894	6,035	17,223	7,787	5,715
Máximo	20,353	6,906	2,399	6,621	27,119	15,468	99,571
I.C.	[7,610, 8,943]	[6,189, 6,363]	[2,025, 2,097]	[6,239, 6,312]	[18,527, 19,566]	[8,836, 9,568]	[12,551, 16,120]
Referência	91,80 %	75,23 %	15,77 %	83,53 %	27,43 %	31,32 %	101,18 %

(c) Desempenho de operações do SVG com sombras preparadas sem *hot swap*.

Figura 8.10.: Desempenho de algumas operações da aplicação SVG em três cenários: ausência de JBoss AOP; sombras de execução e acesso a campo preparados com *hot swap*; e a mesmas sombras preparadas sem *hot swap*.

As tabelas da Figura 8.10 mostram o resultado das medições efetuadas. Nelas, podemos ver que o tempo médio de desempenho dessas operações não sofre alterações consideráveis quando preparamos as sombras no modo *hot swap*. Porém, quando preparamos as sombras para operações de combinação dinâmica sem habilitar o *hot swap*, o desempenho dessas operações sofre o custo adicional das chamadas aos ganchos vazios. A linha de referência das tabelas mostra qual o impacto no desempenho em porcentagem. Nela, podemos ver que o custo adicional das chamadas a ganchos vazios torna a operação de colar retângulos 101,18% mais lenta, quando comparado ao desempenho das operações sem JBoss AOP. Por outro lado, ao prepararmos as sombras com o *hot swap* habilitado, não há custo de chamadas a ganchos vazias e impacto no desempenho é muito baixo, de 2,68% apenas.

Apesar de termos obtido resultados tão promissores na ausência de aspectos em um sistema preparado para a combinação dinâmica, o uso do JBoss AOP imprime um custo adicional em relação à memória que a aplicação ocupa. O uso de *hot swap* não muda o fato de que alterações nos *bytecodes* de uma classe acarretarão em maior espaço ocupado na memória. Como vimos na Seção 7.1.1 na página 105, é preciso adicionar os ganchos e campos auxiliares à interceptação antes de as classes serem carregadas. Os nossos estudos mostraram que, em média, o tamanho de uma classe aumenta 178,08% após a transformação realizada pelo JBoss AOP com o *hot swap* habilitado. Apesar de ser um número alto, o resultado

pode ser pior na ausência de *hot swap*. Os nossos testes utilizando preparação de todas as sombras de execução e acesso a campos sem o *hot swap* habilitado mostraram um aumento médio de 292,66% no tamanho das classes durante a transformação. A diferença grande entre esses resultados se deve às chamadas aos ganchos que são adicionadas apenas quando o *hot swap* está desabilitado.

8.4. Conclusão

Vimos que o desempenho a nossa abordagem de atualização de cadeias, habilitada quando o JBoss AOP é executado no modo de casamento indexado, possui em geral um desempenho muito superior ao desempenho da solução original. Na presença de adendos, a superioridade da nossa solução é indiscutível, quando comparada ao aumento no tempo decorrido em uma operação nos cenários com 30 e 60 adendos. Além disso, vimos que, na ausência de associações e adendos, o modo indexado é superior ao modo padrão para a adição de associações envolvendo *pointcuts* do tipo acesso a campos e execução. Quanto aos *pointcuts* do tipo chamada, ele tem um desempenho inferior. Felizmente, trata-se de uma diferença que não chega a 300ms nas medidas de duração média. O ganho com o modo indexado, por outro lado, pode chegar a medir segundos, como mostraram os resultados obtidos por nossos testes.

Quando à instrumentação em tempo de execução, ela realmente fornece garantias quanto ao desempenho de sombras preparadas que não serão instrumentadas. A instrumentação em tempo de execução afetará o fluxo de controle de uma sombra somente quando houver interceptadores na cadeia daquela sombra. Apesar disso, há ainda o custo inerente do aumento do espaço ocupado na memória pela aplicação instrumentada, como pudemos observar.

Podemos concluir, com todos esses testes, que o nosso trabalho cumpriu o papel a que se propôs, no sentido de buscar resultados melhores no desempenho da combinação dinâmica como um todo.

9. Trabalhos Relacionados

Neste capítulo faremos uma breve apresentação dos trabalhos relacionados ao nosso estudo. Começamos

9.1. Projetos Relacionados

Como projetos relacionados, podemos mencionar o AspectWerkz, o JasCO e o PROSE. Todos esses projetos foram estudados em profundidade no Capítulo 3. Por esse motivo, vamos colocar neste capítulo apenas um resumo do que vimos anteriormente:

AspectWerkz essa ferramenta é bem similar ao JBoss AOP. Os aspectos podem ser configurados através de arquivos XML e as operações dinâmicas são fornecidas por uma API. Essa API é formada por uma única classe, o `Deployer`. O AspectWerkz utiliza a instrumentação em tempo de execução para implementar a combinação dinâmica de aspectos;

JasCO o JasCO estende a linguagem Java e suporta operações dinâmicas através de uma API e através de *hot deployment* de aspectos. O JasCO também utiliza instrumentação em tempo de execução para realizar a combinação dinâmica de aspectos;

PROSE essa é uma ferramente desenvolvida com o objetivo de estudar as diversas formas de implementação da combinação dinâmica de aspectos. Assim, há três implementações de combinação no PROSE. A primeira executa o programa em modo de depuração, inserindo *breakpoints* nos pontos onde deve haver chamadas a adendos. Essa é a forma mais lenta das três. A segunda implementação utiliza a JVM *Jikes* para inserir ganchos minimais no programa. Esses ganchos são inseridos utilizando-se a API da própria JVM, garantindo um desempenho superior à versão anterior. Finalmente, a terceira implementação instrumenta *bytecodes* através da ferramenta BCEL, e executa o *hot swap* dos esses *bytecodes* sempre que uma operação dinâmica é executa. O PROSE possui otimizações interessantes, como, por exemplo, o armazenamento das sombras onde há leituras e escritas a campos, o que evita a busca por esses pontos em tempo de execução quando uma operação dinâmica ocorre. Além disso, o PROSE inovou na chamada a ganchos apresentando uma implementação que dispensa a presença de ganchos na interceptação de métodos [62].

Todos os projetos que apresentamos iteram de algum modo pelas sombras de junção, realizando o casamento das mesmas com um *pointcut* quando ocorre uma operação dinâmica.

9.2. Busca de Sombras

Quanto à aplicação de busca de sombras para a identificação dos pontos de junção em tempo de execução, uma abordagem similar foi proposta no trabalho de doutorado de Michael Haupt [58] e implementada num projeto chamado Steamloom.

O Steamloom é uma máquina virtual com suporte à combinação dinâmica, criado com o objetivo de encontrar uma implementação eficiente de combinação dinâmica, através da extensão de uma máquina virtual com suporte à programação orientada a aspectos dinâmica.

A abordagem de busca de sombras implementada pelo Steamloom é uma versão mista do grafo de sombras com o casamento tradicional de sombras. A busca do Steamloom consiste em primeiro extrair a parte do *pointcut* que se refere às classes alvo, fazendo uma busca com essa expressão por uma estrutura de dados que contém informações sobre as classes do sistema base. Isso é similar à primeira etapa da busca que fazemos no grafo, realizada na subárvore de nós `ClassNode` com a expressão ***`type`*** extraída do *pointcut*. A partir desse instante, porém, a busca no Steamloom é totalmente diferente da busca no grafo. Uma vez encontradas as classes, o Steamloom faz o casamento do restante da expressão com todas as sombras contidas na árvore. Quando se trata de *pointcuts* que casam com chamadas ou acesso a campos, o Steamloom utiliza uma estrutura indexadora que permite identificar quais sombras no código base precisam ser alteradas. Isso é similar à otimização realizada pelo PROSE, que armazena informações sobre os pontos onde há acessos a campos. Note que, caso tivéssemos implementado algo similar no nosso trabalho, o resultado dessa busca seria útil para a instrumentação em tempo de execução e não, como poderia parecer inicialmente, para a atualização das cadeias de adendos. Afinal, é a fase de instrumentação que terá que descobrir a localização das sombras no sistema base que precisam ter seus *bytecodes* alterados.

Assim, é possível ver que o grafo inova por ser a única estrutura de dados que utiliza busca para encontrar todos os elementos, e por ser a única estrutura de dados compatível com uma estrutura hierárquica de domínios, onde as classes e instâncias fazem partes de domínios, que podem ser utilizados para restringir a busca. Essa é uma nova abordagem que não foi proposta anteriormente.

Além da criação do Steamloom, não há outro trabalho que tenha proposto busca de sombras para substituir o casamento de *pointcuts* durante operações dinâmicas. Contudo, podemos também destacar mais dois textos como trabalhos relacionados.

Em [54], a complexidade das linguagens *pointcut* é analisada e fica provado que a compilação da linguagem do AspectJ é um problema NP-Completo. Isso prova que a seleção estática de todos os pontos de junção de um sistema é NP-Completa, pois necessita da resolução total das expressões *pointcut*.

Outro trabalho [13] propõe a implementação de um banco de dados em Prolog de forma a armazenar informações relevantes sobre os pontos de junção. A saber, essas informações são: a árvore de sintaxe abstraída do programa, a hierarquia de tipos, o histórico completo da execução do programa até o momento presente, e o conteúdo atual do *heap*. Os *pointcuts* são simples *queries* de Prolog, e são reavaliados a cada momento que uma nova informação é adicionada a essa base de dados. Se, durante essa reavaliação, um ponto de junção é identificado por um *pointcut*, o adendo correspondente é executado.

Essa base de dados provê informação adicional do sistema, referente à sua execução, aumentando a precisão das informações selecionáveis por expressões *pointcut*. Isso resulta em uma linguagem *pointcut* mais poderosa. Por outro lado, ele contém mais dados do que a solução que propomos, uma vez que ele armazena toda a informação referente a execução do sistema. Apesar das vantagens resultantes dessa diferença, essa solução ocupa mais espaço na memória, além de ser mais lenta por exigir a reavaliação de todos os *pointcuts* a cada passo de execução do programa.

9.3. Sistemas Auto-adaptáveis e Hot Swap

Uma vasta gama de trabalhos estudam técnicas de implementação de sistemas auto-adaptáveis.

Em [24], é proposta uma forma de controlar as relações entre componentes e objetos, através de uma camada reflexiva. Isso permitiria a substituição de componentes em tempo de execução, de forma comparável com o *hot swap*. O autor compara a solução apresentada com o *hot swap* em Java, concluindo que a vantagem da solução proposta é que ela tem granularidade mais fina, permitindo a troca objeto a objeto, ao invés da troca de classes.

Sato et. al [69] sugere a utilização de *class loaders* que permitam a troca de uma versão de uma classe por outra, carregada por outro *class loader*. Nessa proposta, vemos uma abordagem alternativa, similar ao *hot swap*. As vantagens na permissão da troca de classes carregadas por *class loaders* diferentes são enumeradas pelo autor.

Em [29] e em [30], abordagens da combinação em tempo de execução são avaliadas, em casos específicos. O primeiro trata de sistemas embutidos, que apresentam uma alta restrição quanto à memória ocupada. O segundo, trata de sistemas autônomos auto-adaptáveis, capazes de solucionar erros em si mesmos, além de outras características de auto-adaptabilidade.

Uma forma de realizar *hot swap* sem a JVMTI é apresentada em [17]. Finalmente, em [60], é descrita a implementação da programação orientada a aspectos de forma distribuída. Nesse trabalho, a possibilidade de realizar *hot swap* remotamente é abordada.

Finalmente, os artigos [76] e [75] descrevem a implementação da combinação dinâmica em tempo de execução através de *hot swap* no AspectWerkz e no JasCO, respectivamente.

10. Considerações Finais

Este trabalho apresenta um estudo completo sobre a combinação dinâmica, onde avaliamos a sua implementação em diversas ferramentas, propomos novas soluções para a sua implementação, implementamos essas soluções e avaliamos o seu desempenho.

Neste capítulo, faremos uma apresentação final do nosso trabalho, mostrando os aspectos positivos do mesmo, detalhes sobre a sua implementação e propostas de trabalhos futuros para trabalhar em diversos aspectos da nossa solução, indo além dos resultados de desempenho obtidos até aqui. Desse modo, apresentamos as principais contribuições do nosso trabalho na próxima seção. Na Seção 10.2, mostramos algumas curiosidades sobre a implementação desse trabalho. Concluímos o texto com uma lista de trabalhos futuros, na Seção 10.3.

10.1. Principais Contribuições

Acreditamos que esse trabalho colaborou com as pesquisas na área de combinação dinâmica de aspectos das mais variadas formas. Destacamos as seguintes contribuições:

- no Capítulo 3, apresentamos um estudo da forma como a combinação dinâmica de aspectos é implementada em uma série de ferramentas de programação orientada a aspectos dinâmica. Esse estudo exigiu não somente uma vasta pesquisa sobre artigos na área, como também exigiu uma análise do código de implementação de cada uma das ferramentas mencionadas, o que acreditamos ser uma contribuição única na área;

- até onde sabemos, esse é o primeiro trabalho a fazer uma análise detalhada dos fatores que podem acarretar em impactos no desempenho da combinação dinâmica;

- a combinação dinâmica de aspectos no JBoss AOP foi levada a um novo patamar, como mostram os testes do Capítulo 8;

- até onde sabemos, é o primeiro trabalho que propõe uma estrutura de dados complexa para armazenar informações sobre sombras, a fim de substituir o casamento de *pointcuts* por buscas nessa estrutura. Inclusive, é o primeiro trabalho desse tipo compatível buscas por domínios;

- esse trabalho dá continuidade ao estudo sobre instrumentação em tempo de execução, iniciado em 2004 por Alexandre Vasseur [76] e pelos autores Wim Vanderperren and Davy Suvée [75].

10.2. Implementação

A implementação do nosso trabalho está atualmente adaptada à versão 2.0 CR14 do JBoss AOP e pode se encontrada no *branch* do repositório do JBoss:

`http//anonsvn.jboss.org/repos/jbossas/projects/aop/branches/joinpoint_graph`

A segunda parte do nosso trabalho, instrumentação em tempo de execução com *hot swap*, está disponível também na versão de produtividade (diretório `projects/aop/trunk` do repositório). Quanto à primeira parte, há planos de integração após a entrega da versão 2.0 GA. Por enquanto, apenas a parte que evita reconstrução de todas as cadeias foi adicionada ao JBoss AOP, sendo que o grafo será integrado em um momento futuro.

Algumas curiosidades sobre a implementação do grafo de sombras:

- os 936 testes de unidade da árvore de sombras levam em torno de 1,1 segundos para executar no computador descrito na Seção 8.1 na página 115;
- os 1549 testes de unidade do grafo de sombras (o que inclui os testes da árvore) levam em torno de 6,2 segundos para executar no mesmo ambiente de testes;
- excluindo o código de integração do grafo com o JBoss AOP, o grafo é composto por 88 arquivos Java, dos quais 38 compõem a implementação da árvore de sombra;

A implementação da instrumentação em tempo de execução é bem mais curta e está distribuída pelas classes mencionadas no Capítulo 7. Não por isso, a sua implementação também incluiu diversos desafios, como:

- refatorar todos os transformadores do pacote `org.jboss.aop.instrument`, para separar a instrumentação em duas fases (preparação e substituição);
- adicionar os métodos `unwrap` a esses transformadores, responsável por reverter a substituição de sombras por chamadas a ganchos;
- escrever os testes de unidade.

Em relação ao último item, há apenas 6 classes de testes, sendo que cada uma possui 8 testes de unidade. Todas as classes de testes executam os mesmos testes de unidade, porém utilizando arquivos `jboss-aop.xml` distintos. Assim, verificamos a corretude da instrumentação em tempo de execução quando todas as sombras foram preparadas, quando somente execução de métodos foram preparadas e assim por diante. Cada um desses testes verifica a corretude das operações dinâmicas com o *hot swap* em 8 cenários. Todos esses cenários adicionam e removem associações diversas vezes, verificando se o resultado a cada passo é sempre o esperado. Há cenários que verificam operações dinâmicas realizadas através do `InstanceAdvisor` e outros que verificam a corretude do *hot swap* quando as operações dinâmicas do `AspectManager` são invocadas. Há, inclusive, um cenário que verifica se as chamadas aos ganchos são removidas quando há interceptadores associados somente a uma instância e essa instância é coletada pelo *garbage collector*.

O grande desafio em escrever esses testes foi a verificação da corretude. Primeiramente, tivemos que gerar arquivos `.jar` para os cenários a serem testados, pois era preciso que os testes carregassem cada um dos cenários em um *class loader* à parte. Caso contrário, um cenário afetaria o outro, alterando o resultado dos testes. A cada passo dos cenários, a corretude da instrumentação em tempo de execução é averiguada através do `Javassist`, onde o teste inspeciona os *bytecodes* para verificar se as chamadas aos ganchos foram inseridas e removidas conforme o esperado. O teste também executa os pontos de junção, confirmando que os interceptadores são invocados nos momentos esperados.

O número de verificações que esses testes fazem é tamanho que, ao contrário dos testes do grafo, eles são lentos. Os 48 testes de unidade levam mais de um minuto para executar.

10.3. Trabalho Futuro

Consideramos esse trabalho como um projeto de pesquisa inicial na área de desempenho da combinação dinâmica, pois acreditamos que há espaço para muitas possibilidades de trabalho futuro. Assim, propomos:

otimização do grafo para busca de métodos e construtores: na Seção B.6 do Apêndice B, mostramos uma curiosidade sobre o formato das chaves de busca por métodos e construtores. Infelizmente, abrimos mão de um mecanismo de otimização da árvore, que enxerga nomes de pacotes em chaves, para que ele pudesse ser utilizado para enxergar argumentos de construtores e métodos. Queremos implementar uma versão da árvore configurável, permitindo que ela enxergue mais de um nível de elementos em uma chave. Isso permitirá distinguir tanto nomes de argumentos como os nomes dos pacotes no tipos desses argumentos, o que tornará a busca nas subárvores de nós `BehaviorNode` mais eficiente;

otimização da busca por chamadas: infelizmente, a atualização de cadeias de chamadas utilizando o grafo não superou o desempenho do algoritmo que encontramos originalmente no JBoss AOP. Vimos que isso se deve à estrutura do grafo, que definimos de uma forma que encarece esse tipo de busca, a fim de garantir que uma solução econômica, em termos de espaços na memória, pudesse ser implementada para domínios. Queremos fazer um novo estudo desse aspecto do grafo e tentar encontrar uma nova solução que, mesmo compatível com domínios, possa fornecer um desempenho superior na busca por sombras de chamadas.

implementação do hot swap para sombras do tipo chamadas: não implementamos esse item, devido à complexidade da instrumentação de sombras do tipo chamada no código do JBoss AOP. Contudo, acreditamos que a invocação de ganchos do tipo chamada pode causar um impacto consideravelmente maior do que a chamada de ganchos durante a execução de outros tipos de pontos de junção. Como cada método ou construtor pode chamado mais de uma vez, a tendência é haver um maior número de sombras de junção do tipo chamada do que do tipo execução. Queremos implementar esse tipo de instrumentação e averiguar os resultados do mesmo modo que fizemos com os outros tipos de sombra de junção;

implementação de busca por sombras para a instrumentação em tempo de execução: vi-mos que o PROSE e o Steamloom otimizam as operações de combinação dinâmica envolvendo *pointcuts* do tipo chamada ou acesso a campos. Essa otimização é feita através de uma estrutura de dados que armazena a localização das sombras que realizam as chamadas e que acessam os campos. Queremos estender o grafo para ser utilizado também com esse objetivo. Isso permitiria uma otimização da instrumentação em tempo de execução, quando fosse necessário substituir sombras do tipo chamada e acesso a campos. A nossa solução atual utiliza o código original do JBoss AOP e itera pelos *bytecodes* das classes preparadas, verificando se cada sombra encontrada é uma das chamadas ou acesso a campos que precisam ser substituídos como resultado de uma operação dinâmica. Acreditamos que essa otimização resultaria em uma melhora considerável no desempenho das operações dinâmicas nesses cenários.

adaptação de uma máquina virtual: as soluções que propomos nesse trabalho são portáveis, pois não dependem da máquina virtual onde executam. Essa é uma qualidade que consideramos valiosa. Contudo, sabemos que podemos implementar a nossa solução em código nativo e adaptá-la a uma máquina virtual, o que nos permitirá utilizar estruturas internas da máquina virtual para representar sombras. Com isso, esperamos obter maior eficiência nas operações de combinação dinâmica. Assim, queremos portar a nossa solução para uma máquina virtual, a fim de comparar os resultados obtidos com a solução aqui apresentada.

A. JBoss AOP: Outros Recursos

Na Seção 2.4 na página 10, vimos como declarar adendos e associações no JBoss AOP. Além da associação de adendos a *pointcuts*, existem outros recursos disponíveis no JBoss AOP. Neste apêndice veremos de forma sucinta quais são esses recursos.

A.1. Pointcuts Dinâmicos: `cflow`

Definimos como ***pointcuts* dinâmicos** aqueles que só podem ser casados com um ponto de junção em tempo de execução. Esses *pointcuts* dependem de valores que serão conhecidos apenas durante a execução do programa e, por isso, necessitam ser casados nesse momento. Por terem essa característica, o seu uso incorre naturalmente em um custo de desempenho adicional.

No JBoss AOP existe apenas um tipo de *pointcut* dinâmico: o `cflow`. Esse tipo de *pointcut* permite que se determine quais métodos e construtores devem estar na pilha de execução quando da execução de um ponto de junção. Se essa restrição não for aceita, o ponto de junção não casará com o *pointcut* `cflow`. Com um *pointcut* `cflow`, podemos também definir quais elementos não poderão estar presentes na pilha de execução. Além disso, é possível definir a ordem relativa dos elementos.

Apesar de serem denominados *pointcuts*, no JBoss AOP eles não são declarados através de uma expressão *pointcut* como as que vimos n a Seção 6.1.1.1 na página 71. Eles são declarados na forma de pilhas, o que pode ser feito, assim como ocorre com as outras expressões *pointcut*, tanto em um arquivo xml quanto através de anotações. Vamos nos ater a mostrar como declarar *pointcuts* do tipo `cflow` em um arquivo xml. Veja o exemplo a seguir:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aop>
3      ...
4      <cflow-stack name="noRecursion">
5          <called expr="void POJO->otherMethod()"/>
6          <called expr="int POJO->recursive(int)"/>
7          <not-called expr="int POJO->recursive(int)"/>
8      </cflow-stack>
9
10     <bind pointcut="execution(int POJO->recursive())" cflow="noRecursion>
11         <advice name="anyAdvice" aspect="MyAspect"/>
12     </bind>
```

```
13      ...
14  </aop>
```

Listagem A.1: Exemplo de *pointcut* do tipo `cflow`

Nas linhas 4-8, vemos a declaração da expressão `cflow` que, como dissemos, é feita através de uma declaração de uma pilha. Essa pilha pode conter as *tags* internas `called` e `not-called`, indicando, respectivamente, a presença ou ausência de métodos e construtores na pilha de execução. No exemplo, o *pointcut* `cflow`, chamado `noRecursion`, define que a pilha precisa conter o método `POJO.otherMethod()`. Após esse método ter sido executado, é preciso que o método `POJO.recursive(int)` também tenha sido chamado. Note que isso não indica que `otherMethod` tenha que ter chamado `recursive(int)`. O número de chamadas na pilha de execução entre um elemento e outro pode ser zero ou mais. O *pointcut* `cflow noRecursion` também declara que, uma vez que existe uma chamada de `recursive(int)` na pilha, não pode existir outra após ela, como podemos ver na linha 8.

O nome de uma expressão `cflow` pode ser utilizado na declaração de associações, como foi feito nas linhas 10-12. O atributo opcional `cflow` contém o nome da expressão, `noRecursion`. É possível combinar *pointcuts* `cflow` nesse atributo. Assim como nas expressões convencionais, o uso de `AND`, `OR` e parênteses é permitido dentro desse atributo.

No exemplo acima, a associação indica que as execuções do método `POJO.recursive(int)` deverão ser interceptadas. Por causa do `cflow` presente na linha 10, somente a execuções que estiverem ocorrendo dentro do fluxo de controle do método `POJO->otherMethod()`, e que não forem chamadas recursivas, serão interceptadas.

A.2. Introdução de Interfaces e Mixins

O JBoss AOP, assim como a maioria das ferramentas de POA, suporta o recurso de introdução de *mixins*.

Como já vimos na Seção 2.2 na página 6, uma *mixin* é responsável por responder por métodos que foram introduzidos numa classe do sistema base. Quando um desses métodos for invocado, será o *mixin* que responderá por ele. Para isso, é preciso que o *mixin* possua um método com a mesma assinatura do método introduzido.

No JBoss AOP, a introdução de métodos é feita através de interfaces. Para especificar a lista de métodos que você quer introduzir numa classe, você precisa apresentar uma ou mais interfaces. O JBoss AOP verificará quais métodos da(s) interface(s) fornecidas aquela classe não implementa. Todos esses métodos serão inseridos na classe e sua execução será delegada para o *mixin*. Apesar de parecer uma decisão restritiva exigir o uso de interfaces para a introdução de métodos, essa é uma decisão que facilita a chamada desses métodos. Pois basta fazer um *casting* para a interface para poder invocar os métodos inseridos. Sem o uso de uma interface, isso já não seria possível e estaríamos presos ao uso de reflexão.

Uma introdução de interface com *mixin* pode ser vista no próximo exemplo, retirado do tutorial do JBoss AOP:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aop>
3     ...
4     <introduction class="POJO2">
5        <mixin>
6           <interfaces>
7              java.io.Externalizable
8           </interfaces>
9           <class>POJO2ExternalizableMixin</class>
10          <construction>new POJO2ExternalizableMixin(this)</construction>
11       </mixin>
12    </introduction>
13    ...
14 </aop>
```

Listagem A.2: Exemplo de introdução com *mixin*.

Como mostra o exemplo, a *tag* `introduction` declara uma introdução da interface `Externalizable`, associada ao uso do *mixin* `POJO2ExternalizableMixin`. O atributo `class` da *tag* `introduction` deve conter uma expressão *pointcut* do tipo *type*, cuja sintaxe é mostrada na tabela 6.1 na página 72. No caso, ela é uma expressão simples, sem o uso de curingas ou outras construções, e seleciona apenas a classe de nome `POJO2`. Observe também que é possível colocar mais de uma interface dentro da *tag* interna `interfaces`. A linha 10 mostra como o *mixin* deve ser instanciado. O `this` tem um significado especial dentro dessa *tag*. Ele representa a instância de `POJO2` que delegará as execuções dos métodos da interface `Externalizable` para aquela instância do *mixin*. Essa linha poderia conter algo mais complexo, como uma chamada a um método estático de uma *factory*, por exemplo. Veja, a seguir, o código da classe `POJO2ExternalizableMixin`:

```
1  public class POJO2ExternalizableMixin implements java.io.Externalizable
2  {
3     POJO2 pojo;
4
5     public POJO2ExternalizableMixin(POJO2 pojo)
6     {
7        this.pojo = pojo;
8     }
9
10    public void readExternal(ObjectInput in)
11                          throws IOException, ClassNotFoundException
12    {
13       pojo.stuff2 = in.readUTF();
14    }
```

```
15     public void writeExternal(ObjectOutput out) throws IOException
16     {
17         out.writeUTF(pojo.stuff2);
18     }
19 }
```

Listagem A.3: Implementação de um *mixin*.

A.3. Introdução de Anotações

Outra funcionalidade que apresentaremos é a introdução de anotações. As anotações introduzidas em uma classe, método ou campo podem ser utilizadas por um outro arcabouço. E podem, ainda, ser utilizadas em expressões *pointcut*.

Veja o exemplo de introdução de anotações abaixo:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <aop>
3     ...
4     <annotation-introduction expr="execution(* POJO->*(..))" >
5         @MyAnnotation(value="aop-introduced")
6     </annotation-introduction>
7     ...
8 </aop>
```

Listagem A.4: Exemplo de introdução de anotações

No exemplo dado, a anotação `@MyAnnotation(value="aop-introduced")` é introduzida em todos os métodos da classe `POJO`.

A expressão `"execution(* *->@MyAnnotation(..))"` é um exemplo de *pointcut* que utilizaria a anotação introduzida para selecionar métodos.

A.4. Funcionalidades Avançadas

Além do que já descrevemos neste capítulo, existem inúmeras outras funcionalidades disponibilizadas pelo JBoss AOP. Dentre elas podemos citar a **adição de meta-dados**, largamente utilizada pelo JBoss AS e as **regras de precedência de adendos**. Esse último permite que se defina qual adendo deve ser executado antes, qual deve ser executado por último, etc.

Há, ainda, a possibilidade de definir **regras de acesso**, com o uso de `declare-warning` e `declare-error`. Essas regras podem controlar acesso a métodos e campos. Isso pode ser útil quando um campo `protected` ou `public` só puder ser acessado por um determinado conjunto de classes.

Com o JBoss AOP também é possível escrever a sua própria implementação de *pointcut*, através dos ***pointcut* extensíveis**. É também possível escrever a sua própria regra de `cflow` através dos `cflow` **dinâmicos**.

Aspectos e interceptadores podem ler dados do arquivo `jboss-aop.xml`, se implementarem a interface `org.jboss.aop.XmlLoadable`. Isso permite que se coloque outras *tags* dentro das *tags* `aspect` e `interceptor`. Essas *tags* internas serão repassadas para os aspectos e interceptadores que estiverem sendo declarados.

Há também as introduções de interfaces sem *mixins*. Nesse cenário, o JBoss AOP delega a execução dos métodos introduzidos para a pilha *around* associada a cada um deles.

B. A Árvore de Sombras em Profundidade

Na Seção 6.2 na página 81, apresentamos a árvore de sombras e mostramos os principais algoritmos envolvidos na busca. Contudo, há diversos fatores na implementação da árvore de sombras que contribuem para um bom desempenho e economia de espaço de memória de maneira significativa. Nesse apêndice, iremos mostrar quais são esses fatores e como eles se refletem nos algoritmos de busca da árvore.

Começamos na Seção B.1 apresentando a solução utilizada que um nó possa referenciar um número variável de nós filhos sem acarretar num alto consumo de memória. Na Seção B.2, formalizamos o controle de comprimento, responsável por evitar comparações de caracteres desnecessárias durante uma busca. Em seguida, a Seção B.3 mostra como as chaves são divididas em duas camadas distintas, o que aumenta a granularidade do controle de comprimento, tornando-o mais eficaz. Como conseqüência dessa divisão, é possível implementar o suporte a chaves do tipo `package` de maneira simples, tema que será abordado na Seção B.4. Uma visão geral da arquitetura da árvore é dada na Seção B.5. Por fim, a Seção B.6 mostra que as chaves identificadoras de métodos e construtores precisam seguir um formato específico, para evitar que a busca na árvore de `BehaviorNode` devolva resultados incorretos.

B.1. Referências a Nós Filhos

No início da Seção 6.2 na página 81, mencionamos que os identificadores de subchaves são utilizados por nós internos para identificar os seus filhos. A forma como um nó referencia os nós filhos pode impactar o espaço de memória que a árvore ocupa consideravelmente.

Uma forma ingênua de representar a conexão entre um nó interno e nós filhos é através de um vetor de nós filhos, cujo tamanho seria o número máximo de nós filhos permitido. Com essa implementação, a busca por um nó filho leva tempo constante, utilizando-se o *id* do nó procurado como índice no vetor. Porém, uma vez que o identificador pode ser qualquer caractere pertencente ao alfabeto Σ, incluindo caracteres *unicode*, o número máximo de nós filhos é tão grande quanto o número de caracteres *unicode* existentes, o que torna essa solução impraticável. Ao invés disso, nós utilizamos uma outra abordagem, que faz uso de *flags* com bits para determinar a posição de um nó filho no vetor. Na solução que propomos, cada nó interno é composto de três vetores:

`children` : contém os nós filhos, em ordem crescente de *id*. O comprimento inicial de `children` é dado por $max(C, c)$, onde $C \geq 0$ é uma constante fornecida à árvore de sombras, e c é o número total de nós filhos contidos no nó interno atual. Se o comprimento de `children` atingir o tamanho c

e um novo nó filho precisar ser adicionado, o vetor `children` é realocado para o tamanho $c + I$, onde $I \geq 1$ é também uma constante fornecida à árvore.

μ : um vetor de *flags*, onde cada elemento j, denotado por μ_j, contém uma *flag* de 16 bits. Esses bits representam um intervalo de identificadores possível e determinam a presença ou ausência de cada nó filho possível dentro desse intervalo. Inicialmente, essas *flags* cobrem apenas os identificadores da tabela ASCII. Se um nó filho identificado por um caractere que não está nessa tabela for adicionado, o intervalo de *flags* possíveis é aumentado de forma a incluir esse novo caractere no intervalo de *flags*. Para isso, o vetor μ é realocado[1]. Para obter o valor da *flag* correspondente ao identificador *id*, o seguinte cálculo é realizado:

$$\frac{\mu_{i \div 16}}{2^{i \bmod 16}} \bmod 2 \tag{B.1.1}$$

onde $i = id - offset$ e $offset$ é o primeiro caractere do intervalo representado pelo vetor μ.

λ : é um vetor utilizado para computar o índice correspondente a um *id* no vetor `children`. Esse vetor é composto por *bytes* e seu comprimento é o mesmo comprimento do vetor μ. O valor de cada elemento i do vetor λ é definido pela seguinte equação:

$$\lambda_i = \sum_{j=0}^{i-1} \left(\sum_{k=0}^{15} \left(\mu_j \bmod 2^k \right) \right) \tag{B.1.2}$$

Existem alguns cenários onde um *byte* não será suficiente para armazenarmos essa soma. Se isso ocorrer, esse vetor será substituído em tempo de execução por um vetor de inteiros.

Utilizando esse trio de vetores, a adição de um nó filho, cujo identificador denotamos por *id*, consistirá nos seguintes passos:

1. calcular o valor de i:

$$i = id - offset \tag{B.1.3}$$

onde $offset$ é o primeiro caractere do intervalo representado pelo vetor μ.

2. definir a *flag* correspondente a *id* como positiva, através da operação:

$$\mu_{i \div 16} = \mu_{i \div 16} + 2^{i \bmod 16} \tag{B.1.4}$$

3. adicionar o nó ao vetor `children` através de um algoritmo de inserção linear. A posição de `children` onde esse nó deve ser inserido é dada pela equação:

$$\lambda_{i \div 16} + \sum_{k=0}^{i} \left(\mu_i \bmod 2^k \right) \tag{B.1.5}$$

[1] Estimamos que esse cenário seja bastante incomum, uma vez que a maioria dos programas Java utilizam apenas caracteres ASCII nos nomes de classes, campos e métodos.

Observe na Equação (B.1.2) como o vetor λ funciona como um atalho para o cálculo da Equação (B.1.5). A presença desse vetor tem o intuito de evitar o custo adicional de calcularmos o valor de $\lambda_{i \div 16}$ a cada busca por um nó filho, algo que pode ocorrer durante dezenas de vezes a cada operação de busca no grafo.

Com o uso desse trio de vetores, a busca de um nó filho, cujo identificador é *id*, envolve as seguintes operações:

1. determinar a presença desse nó filho através da Equação (B.1.1);
2. caso a presença seja negativa, a busca pelo nó filho é abortada;
3. caso contrário, o próximo passo será calcular o índice j no vetor `children` que contém o nó procurado. O cálculo de j é dado pela Equação (B.1.5);
4. finalmente, o passo final é devolver o j-ésimo elemento de `children` como resultado da busca.

B.2. Controle de Comprimento

Nesta seção mostraremos o que é o controle de comprimento, como ele pode ser utilizado no estado inicial de uma comparação e como ele pode evitar comparações que fatalmente resultarão em fracasso durante uma busca.

Definição Dada uma instância de um algoritmo de busca na árvore, definimos o estado inicial da comparação que esse algoritmo irá executar como $\mathcal{S}(T, n, j, searchKey, skp, i)$, onde:

T é a árvore $T = (N, r, E, K, V, \delta)$ na qual a busca está sendo realizada;

n é o nó $n \in N$ onde a próxima comparação de caracteres da busca irá ser executada;

j é o índice do primeiro caractere da subchave contida em n que será comparado;

searchKey é a chave de busca;

skp é a subchave de busca que está atualmente sendo comparada pelo algoritmo de busca;

i é o índice do primeiro caractere de skp que o algoritmo de busca irá comparar.

Dizemos que a comparação se inicia na posição $(\mathbf{n}, \mathbf{j})$ de T e na posição $(\mathbf{skp}, \mathbf{i})$ de $searchKey$.
Ainda, definimos $searchKey$ da seguinte forma:

$$searchKey = skp_0 \cdot \texttt{`*'} \cdot skp_1 \cdot \texttt{`*'} \ldots skp_{p-1} \cdot \texttt{`*'} \cdot skp \cdot \texttt{`*'} \cdot skp_{p+1} \ldots \cdot \texttt{`*'} \cdot skp_{m-1} \quad \text{(B.2.1)}$$

Onde m é o número de subchaves que compõem $searchKey$, skp_l é a l-ésima subchave de $searchKey$ e skp, é a p-ésima chave de $searchKey$.

Observe que o estado inicial da comparação existe para os quatro tipos de algoritmos de busca que vimos anteriormente, pois é o estado que precede a execução da iteração de comparação de caracteres, presente em todos eles. É interessante também notar que nos algoritmos de busca simples e de prefixo, a posição j no nó n é sempre 0. Quando se trata da busca de padrão e de busca de sufixo, por outro lado, o valor de inicial de j é dado pelo resultado da busca da subchave anterior. Nas chamadas recursivas de uma busca de padrão ou de sufixo, j poderá ser 0, quando o algoritmo segue para o início de um novo nó na árvore. Quanto ao valor do índice i na subchave de busca skp, a comparação de skp sempre se inicia no primeiro caractere da subchave. Portanto, quando começamos a comparação de uma subchave, i é 0. Já nas chamadas recursivas do algoritmo que busca por skp, o valor de i será definido pela instância do algoritmo que fez a chamada recursiva, de modo a dar continuidade à comparação da cadeia de caracteres skp.

Desse ponto em diante, denotaremos a relação entre uma folha l na árvore $T(N, r, E, K, V, \delta)$, e a chave que ela representa $key \in K$, por $key \mapsto l$:

$$key \mapsto l \Leftrightarrow path_T(r, l) = key$$

Dado um estado inicial de comparação, $\mathcal{S}(T, n, j, searchKey, skp, j)$, chamamos de sufixos as cadeias de caracteres da árvore que deverão ser comparadas desse estado em diante, até chegarmos ao final de uma folha com sucesso na busca.

Definição Seja $\mathcal{S}(T, n, j, searchKey, skp, i)$ o estado inicial de comparação num determinado instante da busca de $searchKey$ em T. Denotamos a cadeia de caracteres que forma a subchave do nó n por $S[0..|n| - 1]$, onde $|n|$ é seu comprimento. De forma similar, $skp = V[0..|skp| - 1]$.

Dado $key \in K$ onde $key \mapsto l$ sendo l uma folha na subárvore $T_n = (N_n, n, E_n, K_n, V_n, \delta_n)$, definimos:

***Sufixo de** key **em** (n, j)* : é a cadeia dos caracteres que ainda precisam ser comparados a partir da posição (n, j) até chegarmos ao final da folha l. O sufixo de key em (n, j) é representado por $\gamma_{key}(n, j)$, conforme a equação abaixo[2]:

$$\gamma_{key}(n, j) = S[0..j - 1]^{-1} \cdot path_T(n, l) \tag{B.2.2}$$

Também definimos:

***Conjunto dos sufixos em** (n, j)* : é o conjunto de todos os sufixos possíveis em (n, j):

$$\Gamma(n, j) = \{\gamma_{key}(n, j) \mid \forall\, key \in K, key \mapsto l \in N_n\} \tag{B.2.3}$$

***Sufixo de** $searchKey$ **em** (skp, i)* : é a cadeia de caracteres de $searchKey$ que ainda precisa ser comparada com caracteres da árvore até que obtenhamos sucesso na busca. O sufixo de uma

[2] A definição dada utiliza a operação inversa à concatenação, dada por $S[0..j - 1]^{-1} \cdot path_T(n, l)$. Essa operação representa uma extração de um prefixo. Para ilustrar essa operação, considere a expressão $\texttt{"asp"}^{-1} \cdot \texttt{"aspecto"}$, onde o resultado dessa operação é a cadeia de caracteres `"ecto"`.

chave de busca é denotado por $\gamma_{searchKey}(skp, i)$ e é definido por:

$$\gamma_{searchKey}(skp, i) = V[i..|skp| - 1] \cdot skp_{p+1} \cdot skp_{p+2} \cdot ... \cdot skp_{m-1} \tag{B.2.4}$$

O foco do controle de comprimento que iremos descrever são os comprimentos desses sufixos. Denotamos o comprimento do sufixo $\gamma_{key}(n, j)$ por $\phi_{key}(n, j)$ e, da mesma forma, o comprimento do sufixo $\gamma_{searchKey}(skp, i)$ é dado por $\phi_{searchKey}(skp, i)$. Em relação ao comprimento de todos os sufixos na posição $(n,\ j)$ da árvore, dados pelo conjunto $\Gamma(n, j)$, definimos:

Definição O conjunto dos comprimentos dos sufixos em $(n,\ j)$ é dado por:

$$\Phi(n, j) = \{\phi_{key}(n, j) \mid \forall\ key \in K,\ k \mapsto l \in N_n\} \tag{B.2.5}$$

O valor mínimo do conjunto $\Phi(n, j)$ será representado por $\Phi_{min}(n, j)$; o máximo, por $\Phi_{max}(n, j)$.

Definimos, também, os sufixos e seus comprimentos em n, como se segue:

Definição Seja $\mathcal{S}(T, n, j, searchKey, skp, i)$ o estado inicial de comparação num determinado instante da busca de $searchKey$ em T. Denotamos a cadeia de caracteres que forma a subchave do nó n por $S[0..|n| - 1]$, onde $|n|$ é seu comprimento. Definimos:

***Sufixo de** key **em** n* : dado $key \in K$ onde $key \mapsto l$ sendo l uma folha na subárvore $T_n = (N_n, n, E_n, K_n, V_n, \delta_n)$, definimos o sufixo de key em n:

$$\gamma_{key}(n) = \gamma_{key}(n, 0) = path_T(n, l) \tag{B.2.6}$$

Denotamos o comprimento de $\gamma_{key}(n)$ por $\phi_{key}(n)$.

***Conjunto dos sufixos em** n* : é o conjunto de todos os sufixos possíveis em n:

$$\Gamma(n) = \Gamma(n, 0) = \{\gamma_{key}(n) \mid \forall\ key \in K,\ k \mapsto l \in N_n\} \tag{B.2.7}$$

O conjunto dos comprimentos dos sufixos em n é dado por $\Phi(n)$; o valor mínimo desse conjunto será representado por $\Phi_{min}(n)$ e o máximo, por $\Phi_{max}(n)$.

O lema a seguir mostra uma relação importante entre $\phi_{key}(n)$ e $\phi_{key}(n, j)$:

Lema B.2.1 *Seja $\mathcal{S}$(T, n, j, searchKey, skp, i) o estado inicial de comparação num determinado instante da busca de searchKey em T. Podemos afirmar que:*

$$\phi_{key}(n, j) = \phi_{key}(n) - j \tag{B.2.8}$$

Prova Por (B.2.2) e (B.2.6), temos:
$\gamma_{key}(n, j) = S[0..j - 1]^{-1} \cdot path_T(n, l) \implies \gamma_{key}(n, j) = S[0..j - 1]^{-1} \cdot \gamma_{key}(n)$

Disso, podemos concluir que a relação entre o comprimento de $\gamma_{key}(n, j)$ e o comprimento de $\gamma_{key}(n)$ é:

$\phi_{key}(n, j) = -j + \phi_{key}(n) \implies \phi_{key}(n, j) = \phi_{key}(n) - j$

■

O teorema a seguir descreve a propriedade que utilizamos no controle de comprimento na busca de sufixo:

Teorema B.2.2 *Seja $\mathcal{S}(T,\ n,\ j,\ searchKey,\ skp,\ i)$ o estado inicial de comparação num determinado instante da busca de $searchKey$ em T. Se skp é a subchave sufixa da chave de busca $searchKey$, os caracteres de n contidos no intervalo*

$[0, min(|n|, \Phi_{min}(n) - |skp|)[$

são irrelevantes para a busca de $searchKey$.

Prova Seja r o índice de um caractere c no nó n, onde $r \in [0, |n|[$ sendo c é relevante para a busca de $searchKey$. Queremos provar que $r \notin [0, min(|n|, \Phi_{min}(n) - |skp|)[$.

Se o caractere c é relevante para a busca de $searchKey$, ele é relevante para a busca do sufixo skp, pois essa busca é quem retorna o resultado final da busca de $searchKey$. Para que c seja relevante à busca do sufixo, ele deve pertencer aos $|skp|$ últimos caracteres de uma chave key onde $key \mapsto l$ sendo l uma folha da subárvore T_n. Desta forma:

$\phi_{key}(n, r) \leq |skp| \implies \phi_{key}(n) - r \leq |skp| \implies \phi_{key}(n) \leq |skp| + r \implies$

$\implies \Phi_{min}(n) \leq |skp| + r \implies r \geq \Phi_{min}(n) - |skp|$

Dado que $r \in [0, |n|[$, temos:

$$\Phi_{min}(n) - |skp| \leq r < |n| \tag{B.2.9}$$

Quando $|n| < \Phi_{min}(n) - |skp|$, por (B.2.9) concluímos que $\nexists r \in [0, |n|[$ onde r é o índice de um caractere relevante para a busca da chave $searchKey$ e portanto todo caractere no intervalo enunciado pelo teorema

$$[0, min(|n|, \Phi_{min}(n) - |skp|)[\tag{B.2.10}$$

é irrelevante para a busca.

Por outro lado, se $|n| \geq \Phi_{min}(n) - |skp|$, a Equação (B.2.9) afirma que o caractere no índice r é relevante se $r \in [\Phi_{min}(n), |n|[$ e portanto todos os caracteres que estão no intervalo (B.2.10) são irrelevantes para a busca de $searchKey$.

■

O controle de comprimento realizado no algoritmo de busca de sufixo consiste em aplicar o Teorema B.2.2 antes de iniciar a comparação dos caracteres de um nó da árvore com os caracteres do sufixo, como pode ser visto nas linhas 5-9 do Algoritmo 6 na página 93. Quando j não é um índice relevante

para a busca do sufixo, o valor de $relevantIndex$ é assinalado a j, de modo que j passa a ser igual ao menor índice relevante. Caso essa atualização resulte em um índice $j > |keyPart|$, a comparação de caracteres das linhas 10-13 não ocorrerá e o algoritmo executará recursivamente em todos os nós filhos de *node*.

O próximo teorema valida o aborto da busca em (n, j) quando o sufixo $\gamma_{searchKey}(skp, i)$ não cabe mais em nenhum dos sufixos do conjunto $\Gamma(n, j)$. Esse teorema é aplicável não só à busca de sufixo como também aos demais tipos de busca.

Antes de apresentarmos o teorema, faremos uma última definição.

Definição Seja key uma chave na árvore T=(N, r, E, K, V, δ) onde o conjunto de valores $\{value \in V \mid (key, value) \in \delta\}$ fará parte do resultado da busca de $searchKey$ em T. Dizemos que key casa com $searchKey$ e denotamos essa relação por $key \sqsubset searchKey$.

Considere $\mathcal{S}(T, n, j, searchKey, skp, i)$ o estado inicial de comparação num determinado instante da busca de $searchKey$ em T. Se $key \sqsubset searchKey$, onde $key \mapsto l$ e $l \in N_n$, o algoritmo irá percorrer o sufixo de key em (n, j), comparando-o com o sufixo de $searchKey$ em (skp, i), e irá concluir a busca com sucesso. Nesse caso, dizemos que o sufixo $\gamma_{key}(n, j)$ casa com o sufixo $\gamma_{searchKey}(skp, i)$, como especificado na equação a seguir:

$$key \sqsubset searchKey \Rightarrow \gamma_{key}(n, j) \sqsubset \gamma_{searchKey}(skp, i) \tag{B.2.11}$$

O teorema a seguir define o controle de comprimento que aplicamos a todos os algoritmos descritos anteriormente.

Teorema B.2.3 *Seja $\mathcal{S}(T, n, j, searchKey, skp, i)$ o estado inicial de comparação num determinado instante da busca de $searchKey$ em T. Se $\Phi_{max}(n, j) < \phi(skp, i)$, a busca por $searchKey$ na subárvore T_n irá falhar.*

Prova A chave de busca $searchKey$ é formada por m subchaves, sendo que skp é a p-ésima subchave. A forma de $searchKey$ foi definida pela Equação (B.2.1) como:

$searchKey = skp_0 \cdot \text{`*'} \cdot skp_1 \cdot \text{`*'} \ldots skp_{p-1} \cdot \text{`*'} \cdot skp \cdot \text{`*'} \cdot skp_{p+1} \ldots \cdot \text{`*'} \cdot skp_m$

Por definição, um curinga casa com zero ou mais caracteres de uma chave, enquanto que cada subchave de busca casa apenas com uma cadeia de caracteres igual a ela. Assim, podemos afirmar que toda a chave key encontrada na busca pela chave $searchKey$ deve ter comprimento igual ou maior à soma dos comprimentos das suas subchaves de busca:

$\exists\, key \in K \mid key \sqsubset searchKey \implies |key| \geq \sum_{l=0}^{m} |skp_l|$

Se $key \mapsto l$ sendo l é uma folha da subárvore T_n, a Equação (B.2.11) nos permite concluir que $\gamma_{key}(n, j) \sqsubset \gamma_{searchKey}(skp, i)$. Mas, para isso, é preciso que cada caractere de $\gamma_{searchKey}(skp, i)$ seja comparado com um caractere de $\gamma_{key}(n, j)$, resultando em sucesso na busca. Portanto:

$\exists\, key \in K \mid \gamma_{key}(n, j) \sqsubset \gamma_{searchKey}(skp, i) \implies \phi_{key}(n, j) \geq \phi_{searchKey}(skp, i) \implies$

$\implies \Phi_{max}(n, j) \geq \phi_{searchKey}(skp, i)$

Disso, concluímos que:

$\Phi_{max}(n, j) < \phi_{searchKey}(skp, i) \implies \nexists\, key \in K \mid \gamma_{key}(n, j) \sqsubset \gamma_{searchKey}(skp, i)$

■

Na nossa implementação, o Teorema B.2.3 foi aplicado em todos os algoritmos de busca. Antes de começarmos uma comparação de caracteres, verificamos se o teorema se aplica e, em caso afirmativo, abortamos a busca na subárvore T_n.

Além disso, fazemos uma verificação adicional em relação ao comprimento quando se trata do algoritmo de busca simples. Essa verificação aplica o Teorema B.2.4.

Teorema B.2.4 *Seja $\mathcal{S}(T,\ n,\ j,\ searchKey,\ skp,\ i)$ o estado inicial de comparação num determinado instante da busca de $searchKey$ em T. Se skp é uma subchave de busca simples e $\Phi_{min}(n,j) > \phi(skp,i)$, a busca por $searchKey$ na subárvore T_n irá falhar.*

Prova Se skp é uma subchave simples, $searchKey$ não possui curingas e portanto:

$\exists\, key \in K \ \mid\ key \sqsubset searchKey \implies |key| = |searchKey|$

Também sabemos que skp é a única subchave de $searchKey$, por se tratar de uma subchave simples:

$$\exists\, key \in K \ \mid\ key \sqsubset searchKey \implies |key| = |skp| \tag{B.2.12}$$

Pela Equação (B.2.11):

$\exists\, key \in K \mid\ key \sqsubset searchKey \implies \exists\, key \in K \mid \gamma_{key}(n,j) \sqsubset \gamma_{searchKey}(skp,i)$

No estado $\mathcal{S}$, i caracteres de skp já foram comparados. Assim, se $key \sqsubset searchKey$, por (B.2.12) sabemos que precisamos comparar o número restante de caracteres:

$\exists\, key \in K \mid \gamma_{key}(n,j) \sqsubset \gamma_{searchKey}(skp,i) \implies \phi_{key}(n,j) = \phi_{searchKey}(skp,i) \implies$

$\implies \Phi_{min}(n,j) \leq \phi_{searchKey}(skp,i)$

Isso é equivalente a afirmar que:

$\Phi_{min}(n,j) > \phi_{searchKey}(skp,i) \implies \nexists\, key \in K \mid \gamma_{key}(n,j) \sqsubset \gamma_{searchKey}(skp,i)$

■

Para que seja possível a aplicação de todos os teoremas que vimos, os valores de $\Phi_{min}(n)$ e $\Phi_{max}(n)$ são armazenados em cada nó da árvore. De posse desses números, podemos chegar ao valor de $\Phi_{min}(n,j)$ para qualquer índice j. A definição feita na Equação (B.2.5) e o Lema B.2.1 na página 147 nos permitem calcular o valor de $\Phi_{min}(n,j)$ a partir de $\Phi_{min}(n)$:

$\Phi(n,j) = \{\phi_{key}(n,j) \mid\ \forall\ key \in K,\ k \mapsto l \in N_n\} \implies$

$\implies \Phi(n,j) = \{\phi_{key}(n) - j \mid\ \forall\ key \in K,\ k \mapsto l \in N_n\} \implies$

$\implies \Phi_{min}(n,j) = \Phi_{min}(n) - j$

Seguindo o mesmo raciocínio, temos que $\Phi_{max}(n,j) = \Phi_{max}(n) - j$.

O controle de comprimento nos permite evitar o que seriam os dois piores casos dos algoritmos que descrevemos. Um deles é quando uma busca segue até o final de uma folha l comparando os caracteres do caminho percorrido com caracteres de subchaves de busca, para, ao fim, descobrir que chegou ao final de l mas não ao final da chave de busca $searchKey$. O outro pior caso seria o inverso desse, onde

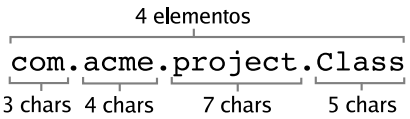


Figura B.1.: Uma chave é dividida em elementos, que são compostos por caracteres.

a busca percorre a árvore até um ponto (n, j) que não é o final de uma folha, mas termina de comparar todos os caracteres da chave de busca $searchKey$. Ambos os cenários terminam em uma falha na busca, após uma série de comparações desnecessárias. O controle de comprimento impede esses cenários de ocorrerem, evitando um número grande de comparações de caracteres que poderiam ser executadas nos cenários descritos.

Na próxima seção veremos uma proposta para aumentar a granularidade dos dados de comprimento, a fim de aumentar eficiência do controle de comprimento.

B.3. Subchaves, elementos e caracteres

Até o momento, descrevemos algoritmos de busca que comparam caracteres da árvore com caracteres da chave de busca. Nesta seção, propomos uma alteração desses algoritmos, onde dividiremos as subchaves em elementos e faremos a comparação de elementos ao invés de caracteres. A comparação de cada elemento da árvore com um elemento da chave será realizada também por algoritmos de busca, dessa vez comparando caracteres com caracteres. Iremos descrever essa solução em detalhes, que resulta em melhor eficácia no controle de comprimento que vimos na seção anterior.

Uma subchave é uma parte de uma chave válida (definida na Equação (6.2.1)) e pode conter um ou mais caracteres `'.'`. Assim, é possível quebrar cada subchave em elementos, utilizando o caractere `'.'` como **caractere separador**. Nesse caso, medimos o comprimento da subchave em elementos e, cada elemento, por sua vez, é composto por caracteres e tem o seu comprimento medido em caracteres. A divisão de uma chave em elementos e caracteres é ilustrada na Figura B.1. Essa divisão também se aplica à chave de busca que, assim como a chave de inserção, possui caracteres `'.'`, conforme a Equação (6.2.2). A divisão de uma chave de busca é ilustrada na Figura B.2-A.

Assim, os algoritmos de busca simples, de prefixo, de padrão e de sufixo agora passam a comparar elementos na árvore com elementos na chave de busca. Uma série de novos fatores resultam a partir dessa mudança. Antes, para casar uma subchave com uma subchave de busca, caracteres eram comparados e o controle de comprimento era realizado utilizando comprimentos medidos em caracteres. Com a quebra de subchaves em elementos a subchave de busca passa a ter o comprimento de seu sufixo também medido em elementos. Isso aumenta a quantidade de informações que temos sobre o sufixo, aumentando a eficácia do controle de comprimento.

Outro fator que resulta dessa mudança é a complexidade maior no algoritmo de busca. A comparação de elementos tem as suas peculiaridades, como mostra a Figura B.2-B. Para uma chave de busca que contém um ou mais curingas, o último elemento de um prefixo e o primeiro elemento de um padrão

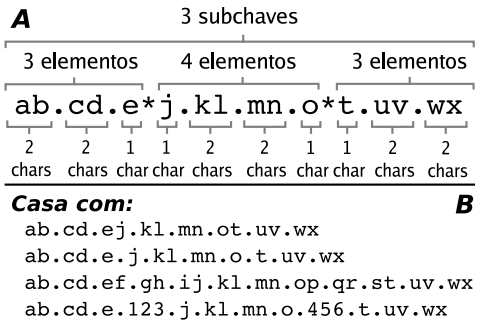


Figura B.2.: A divisão de uma chave de busca em elementos. **A-** Uma chave de busca é composta por subchaves, que estão divididas em elementos, que, por sua vez, são compostos por caracteres. **B-** O último elemento de uma subchave prefixa e o primeiro elemento de uma subchave padrão podem casar com um único elemento na árvore, assim como podem casar com elementos distintos.

podem casar com o mesmo elemento de uma chave na árvore.

Apesar da complexidade adicional que obtemos ao dividir subchaves em elementos, criando uma estrutura de duas camadas (elementos e caracteres), os mesmos algoritmos detalhados na seção anterior podem ser utilizados para fazer buscas nessa árvore. A diferença principal é que eles serão utilizados para comparar tanto elementos quanto caracteres, dependendo da camada na qual estão executando.

Esse mecanismo é ilustrado na Figura B.3 na próxima página. Na primeira linha da figura temos uma chave de busca sem curingas, `"simple.search.expression"`. Essa chave de busca é composta por uma única subchave que, por sua vez, possui 3 elementos. Utilizamos o algoritmo de busca simples para comparar elemento com elemento. Observe que essa comparação é feita de modo transparente, o algoritmo de busca simples de elementos compara elementos sem estar ciente de como a comparação elemento a elemento é realizada. Os elementos dessa chave de busca são comparados também com o algoritmo de busca simples. Na camada de elementos, ele compara caracteres com caracteres, de modo que o elemento `expression`, por exemplo, só casa positivamente com um elemento igual a ele.

A próxima chave de busca que vemos na Figura B.3 possui dois curingas e é portanto dividida em 3 subchaves: `"pr.ef.ix"`, `"pa.tt.ern"` e `"su.ff.ix"`. Usamos o algoritmo de busca de prefixo para buscar pela primeira subchave na árvore. Assim como no exemplo anterior, comparamos cada um dos três elementos que compõe o prefixo, `"pr"`, `"ef"` e `"ix"`, com elementos na árvore. Novamente, a forma como esses elementos são comparados entre si é transparente para o algoritmo de busca de prefixo, cujo objetivo é apenas comparar os elementos com os primeiros elementos da árvore. Vemos na figura também que classificamos os dois primeiros elementos, `"pr"` e `"ef"`, como elementos simples. O motivo dessa classificação é que utilizamos o algoritmo de busca simples para comparar esses elementos com elementos da árvore, pois queremos encontrar elementos iguais a `"pr"` e `"ef"` na árvore. Quanto ao último elemento do prefixo, `"ix"`, usamos o algoritmo de busca de prefixo para comparar os seus caracteres com os primeiros caracteres de um elemento na árvore.

Seguindo o mesmo raciocínio, temos que o padrão `"pa.tt.ern"` é comparado com elementos da

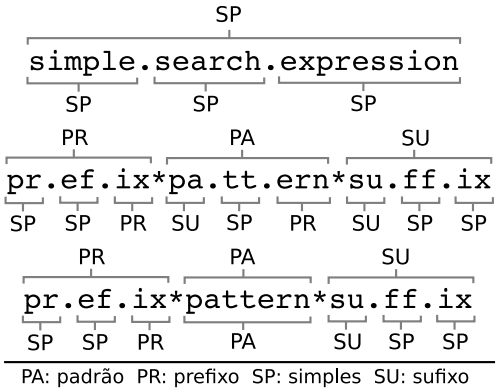


Figura B.3.: Exemplos do uso dos algoritmos de busca simples, de prefixo, de padrão e de sufixo em chaves de busca. Acima de cada chave vemos o algoritmo utilizado para comparar seus elementos com elementos da árvore. Abaixo de cada chave, vemos o algoritmo que compara os caracteres de um elemento de busca com caracteres de um elemento na árvore.

árvore usando o algoritmo de busca de padrão. Novamente, esse algoritmo procurará na árvore pelo padrão composto pelos elementos `"pa"`, `"tt"` e `"ern"`, sem saber como esses elementos são comparados internamente. O elemento `"pa"` casa com um elemento da árvore se esse elemento terminar com os caracteres `"pa"`. Portanto, utilizamos o algoritmo de busca de sufixo para comparar os seus caracteres. Quanto aos outros dois elementos, `"tt"` e `"ern"`, utilizamos os algoritmos de busca simples e de busca de prefixo, respectivamente, para comparar os seus caracteres com os caracteres de um elemento na árvore.

A última chave de busca da Figura B.3 é similar à anterior, exceto pela subchave de padrão, `"pattern"`, composta por um único elemento. Apesar dessa diferença, tratamos essa subchave da mesma forma que no caso anterior, comparando os seus elementos com elementos da árvore de forma transparente. Internamente, porém, esse elemento único que compõe o padrão será comparado com um elemento da árvore utilizando-se o algoritmo de busca de padrão, verificando se os caracteres do elemento `"pattern"` ocorrem no elemento da árvore.

Os algoritmos de busca continuam comparando o comprimento dos sufixos, como descrito anteriormente. A grande diferença é que a maior granularidade da estrutura das subchaves se reflete na granularidade dessas medidas, aumentando a eficácia do controle de comprimento. Assim, os comprimentos máximo e mínimo dos sufixos em um nó n, que denotamos por $\Phi_{min}(n)$ e $\Phi_{max}(n)$, são medidos tanto em elementos como em caracteres. Essas medidas podem ser vistas na Figura B.4, que apresenta uma visão mais completa da árvore da Figura 6.2 na página 84.

Também na mesma figura, vemos que a utilização dos caracteres `'.'` como separadores de elementos não nos permite distinguir se o último elemento de um nó está completo, ou se ele é apenas um prefixo do primeiro elemento de cada um dos nós filhos. Para lidar com isso, adicionamos um valor booleano a cada um dos nós, que deve ser verdadeiro somente nos nós cujo último elemento está completo. Veja como exemplo a raiz da árvore, que contém um único elemento, `"com"`. O valor que indica se o elemento

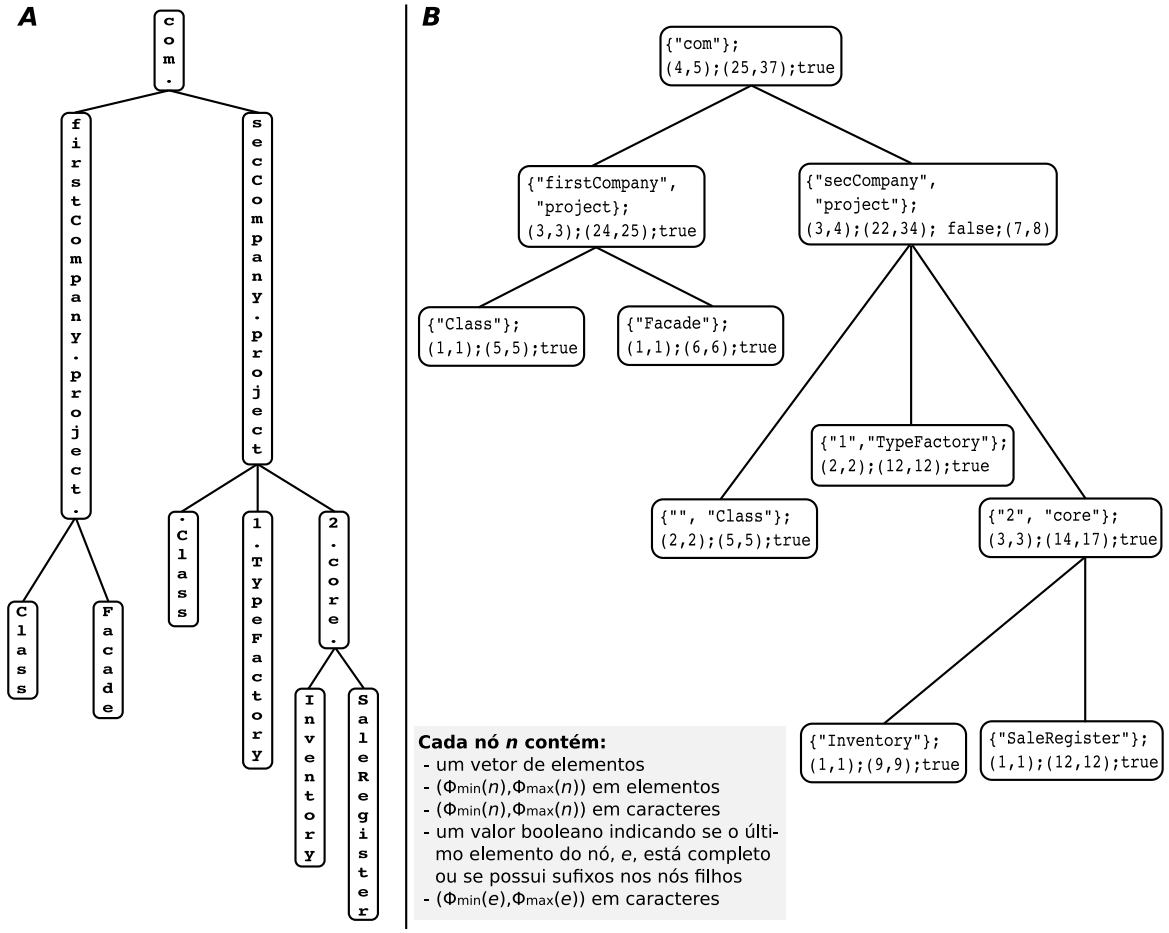


Figura B.4.: Revendo o exemplo da Figura 6.2 na página 84. **A-** Estrutura da árvore vista anteriormente. **B-** Essa imagem mostra em detalhes as informações necessárias para realizar o controle de comprimento em buscas.

está completo nos permite saber que `"firstCompany"` e `"secCompany"`, os primeiros elementos dos nós filhos, não são uma continuação do elemento `"com"`. Caso esse valor fosse `"false"`, isso significaria que as chaves da árvore iniciam com um dos elementos `"comfirstCompany."` e `"comsecCompany."`, elementos esses que teriam sido quebrados ao meio durante a extração do prefixo comum a ambos para a inserção. Esse é o caso do nó que contém `"secCompany.project"`. O valor `false` indica que o seu último elemento não está completo, e ele tem três possíveis complementos, que são o primeiro elemento de cada um dos nós filhos.

Quando o algoritmo de busca estiver operando em um elemento incompleto e, comparando os seus caracteres com os caracteres de um elemento de busca, utilizamos medidas correspondentes a $\Phi_{min}(n)$ e $\Phi_{max}(n)$ para deduzir a distância ao final do elemento e. Assim, denotamos o comprimento máximo e mínimo dos elementos possíveis que se iniciam em e por $\Phi_{min}(e)$ e $\Phi_{max}(e)$. Na Figura B.4-B, o

único nó cujo último elemento está incompleto é o nó que contém `secCompany.project`. O seu último elemento, `"project"`, é complementado nos nós filhos por `""`, `"1"` e `"2"`, indicando que ele é o prefixo de três elementos de chaves na árvore, a saber `"project"`, `"project1"` e `"project2"`. O valor de $\Phi_{min}(e)$ é o menor comprimento desses elementos, 7 e $\Phi_{max}(e)$ é maior comprimento deles, 8.

Observe como o aumento da granularidade desses comprimentos nos permite maior eficácia em casos de falha. Antes, quando medíamos comprimentos em caracteres, chaves como `"com.package.Class"` teriam o mesmo comprimento que `"comPackage.Class1"`. Agora, sabemos que a primeira chave é composta por 3 elementos, enquanto que a segunda, por 2. Além disso, mantemos a medida original em caracteres. Isto nos garante a eficácia do controle de comprimento nos cenários onde esse controle em caracteres é decisivo para abortar uma busca.

B.4. Curingas de Caracteres Separadores

Na tabela 6.1 na página 72, apresentamos a expressão *`package`*, que pode ser utilizada para especificar um tipo em *pointcuts*. Até esse momento, não tratamos da busca por expressões desse tipo, tratamos apenas de expressões de busca do tipo definido na Equação (6.2.2). Apesar de tal omissão, a expressão *`package`* é suportada pela árvore de sombras como chave de busca.

Essa expressão possui um tipo diferente de curinga, que é a seqüência de dois caracteres separadores `".."`. Tal seqüência é utilizada na expressão `package` para casar com o nome de uma classe. Vimos na Seção 6.1.1.1 que a diferença do curinga `".."` para o curinga `'*'` é que o primeiro casa apenas com o nome de uma classe, excluindo classes de um subpacote. Assim, `"my.package.."` casa com `"my.package.AnyClass"`, mas não casa com `"my.package.subpackage.AnyClass"`. A expressão `"my.package.*"`, por outro lado, casa com ambas as classes.

Falando em termos de elementos e caracteres, podemos afirmar que `".."` casa um único elemento qualquer, mas não casa com mais de um elemento. O exemplo anterior mostra isso: `".."` casa com `"AnyClass"`, mas não com `"subpackage.AnyClass"`. Por esse motivo, no nível de elementos traduzimos o curinga `".."` como um elemento qualquer. Porém, ao comparar o elemento correspondente a `".."` com um elemento da árvore, queremos utilizar um algoritmo de busca que nos permita obter sucesso sempre, independente do elemento com o qual estamos comparando. Para isso, traduzimos `".."` em um elemento sufixo vazio `""`, que percorre qualquer elemento até o seu final e considera o resultado da busca como positivo.

Assim, a chave de busca `"my.package.."` é dividida em três elementos: `"my"`, `"package"` e `""`, como ilustra a Figura B.5. Comparamos cada um desses elementos com um elemento da árvore, de modo que queremos encontrar uma chave que também tenha três elementos, sendo que cada elemento dessa chave casa com um elemento da chave de busca. Internamente, a comparação desses elementos é feita de maneiras diferentes. Os dois primeiros elementos devem casar somente com elementos idênticos a eles e, portanto, são elementos simples. O último elemento é um elemento sufixo vazio, que casa com qualquer elemento.

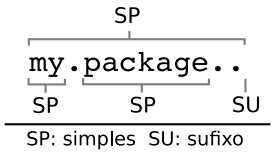


Figura B.5.: Exemplo do uso dos algoritmos de busca simples e de sufixo em chaves de busca com curingas de caracteres separadores.

B.5. Arquitetura

O diagrama de classes da Figura B.6 mostra uma visão simplificada da arquitetura da implementação da árvore de sombras, que iremos descrever nesta seção.

A árvore de sombras é representada pela classe `Tree`. Ela contém apenas um nó, a raiz `root`. Esse nó pode ser tanto um nó interno ou uma folha. Todos nós contêm uma subchave, do tipo $KeyPart$. Além da subchave, um nó pode conter outras informações, conforme o seu tipo. Se o nó é um nó interno, ele contém uma lista dos nós internos; se ele é uma folha, ele contém uma lista dos valores associados à chave que essa folha representa.

A classe `Tree` possui operações de inserção e busca. Para realizar essas operações, `Tree` transforma chaves de inserção e de busca, cujo formato original é `String`, em instâncias das classes `InsertionKey` e `SearchKey`, respectivamente. Esses objetos são responsáveis por percorrer os nós da árvore a fim de executar a inserção ou a busca conforme os algoritmos que descrevemos anteriormente.

Naturalmente, a busca é muito complexa para ser realizada por um único objeto. A classe `SearchKey` se utiliza de várias outras classes para executar a sua tarefa. A classe `Matcher` é a classe principal nesse processo. Essa classe executa um dos algoritmos de busca que vimos anteriormente, comparando uma cadeia de objetos `Comparable` da chave de busca com uma subchave ou mais subchaves da árvore, o que chamamos de alvo da busca. Os objetos `Comparable` são responsáveis por realizar essa comparação, devolvendo um dos resultados: falha na comparação; chegou-se ao final do alvo da comparação; chegou-se ao final dos componentes contidos no objeto `Comparable`; ou chegou-se ao final de `Comparable` e do alvo da comparação. O objeto `Matcher` utiliza o resultado da comparação para definir o próximo passo na busca, que depende do tipo de algoritmo que `Matcher` está executando. Esse algoritmo é especificado através do perfil de `Matcher`, a classe do tipo enum `MatcherProfile`. Note que há um perfil para cada um dos algoritmos que vimos: busca simples, busca de prefixo, busca de padrão e busca de sufixo.

Como a classe `Matcher` delega a comparação de cadeias para objetos `Comparable`, o tipo de elementos que são comparados por um objeto `Comparable` e o modo como eles são comparados é transparente para `Matcher`. Há duas implementações da interface `Comparable`: `ComparableKeyPart` e `ComparableElement`. Essa última representa um elemento de busca e é composta por caracteres. `ComparableKeyPart` é composta por uma lista de objetos `Matcher` que, por sua vez, é responsável por comparar elementos. Essa lista, denominada *element matchers*, delega a comparação para objetos `ComparableElement`. Finalmente, a classe `ComparableKeyPart` é utilizada para comparar elementos através dos objetos `Matcher` contidos na classe `SearchKeyPart`. Essa lista de objetos `Matcher` é denominada *key part matchers*,

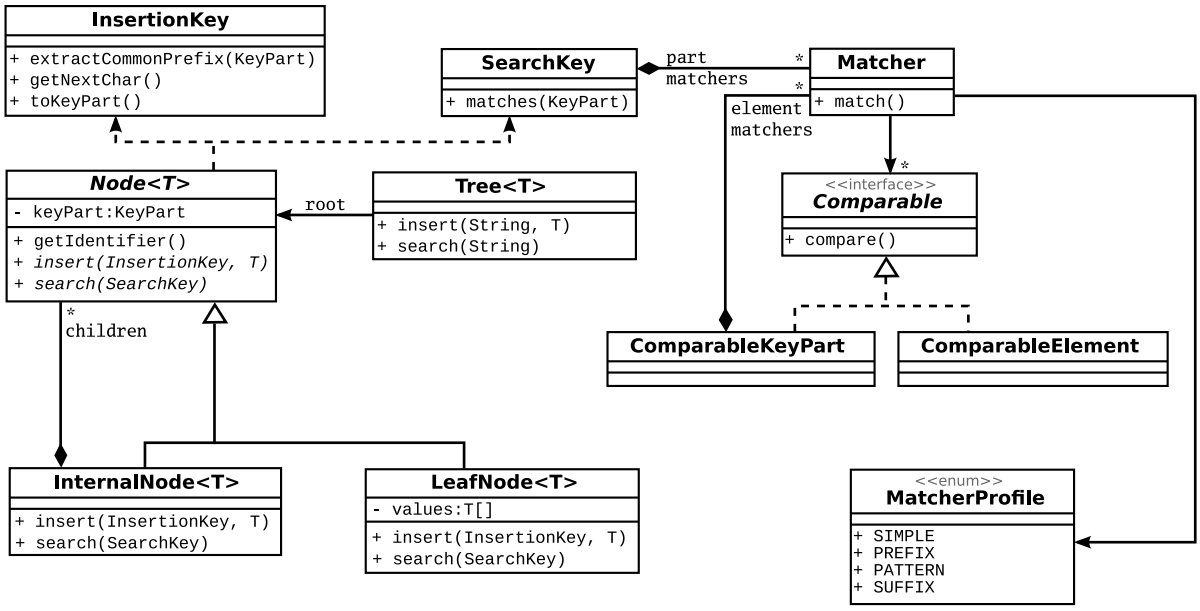


Figura B.6.: Diagrama de classe da árvore de sombra. Esse diagrama mostra a estrutura dessa árvore e dos seus nós. As principais classes auxiliares são também ilustradas no diagrama. Todas as classes desse diagrama pertencem ao pacote `org.jboss.aop.joinpoint.graph.tree`.

uma vez que ela compara subchaves de busca com subchaves na árvore.

B.6. Árvores de Métodos e Construtores

Vimos na Seção 6.3.3 na página 96 que as chaves identificadoras de métodos e construtores devem ser compostas pelas expressões da linguagem *pointcut* a seguir:

return-type method\(*arg-list*\)

constructor\(*param-list*\)

Aparentemente, a chave identificadora de `BehaviorNode` pode seguir o formato acima sem alterações. Porém, não será possível fazer buscas corretas utilizando esse formato. Veja a definição de *arg-list* na tabela 6.1:

arg-list = (*arg*(, *arg*)*)?

Essa expressão declara uma lista de argumentos, que são especificados pela expressão a seguir:

arg = *class*|*primitive*|*|\.\.

Há dois tipos de curingas na lista de argumentos: '*' e "..". Diferentemente do seu significado em uma expressão *type*, esses curingas casam com um argumento qualquer ('*') e com zero ou mais argumentos ("..."), respectivamente.

Se identificamos os métodos por uma chave no formato: *return-type method*\(*arg-list*\), não

será possível obter compatibilidade com o curinga `'*'` na lista de argumentos. Vamos provar isso com um exemplo. Considere o método `void execute(int, long)` de uma classe qualquer do sistema base, que iremos identificar pela chave `"void execute(int, long)"`, conforme o formato sugerido. Dado o *pointcut* abaixo:

```
"execution(void *->execute(*))"
```

Extraímos a chave de busca de métodos `"void execute(*)"`. Ao utilizar essa chave para fazer uma busca na árvore de nós `BehaviorNode`, a árvore irá devolver o nó que representa o método `execute(int, long)`, pois a árvore enxerga o curinga `'*'` como algo que pode casar com zero ou mais caracteres. Porém, a expressão acima não casa com o método `void execute(int, long)`, já que ele possui mais de um argumento.

Portanto, é preciso que a árvore de sombras possa enxergar que o curinga `'*'` está limitado a um único argumento. A árvore, no entanto, só consegue distinguir elementos, separados por um caractere separador. É preciso, então, que ela enxergue os argumentos como um desses componentes para que ela possa tratar o curinga `'*'` de forma adequada. Para isso, iremos transformar os argumentos de um método em elementos. Redefinimos o caractere separador utilizado pela árvore de nós `BehaviorNode` como sendo `'|'`. A chave do método `"void execute(int, long)"` terá um formato diferente, usando esse novo caractere separador:

```
"void| execute#| int| long  "
```

Como é possível ver, além de utilizarmos o caractere `'|'` como separador entre argumentos, o utilizamos também para separar o valor de retorno do nome do método. Isso nos permite tirar vantagem do controle de comprimento, fazendo com que a árvore enxergue `"void"` e `"execute"` como elementos, comparando os seus comprimentos com os comprimentos do nome de um método e do seu tipo de retorno em uma chave de busca.

Usando esse novo formato, extraímos uma chave de busca a partir do *pointcut* `"execution(void *->execute(*))"`:

```
"void| execute#||"
```

Essa expressão faz uso do curinga de caracteres separadores que vimos na Seção B.4, de modo que a árvore só irá casar a expressão acima com métodos que tenham um único argumento.

Quanto ao curinga `".."`, que casa com um ou mais argumentos, ele é traduzido em curingas comuns. Para exemplificar, considere o *pointcut* `"execution(void *->execute(..))"`. Extraímos dele a chave de busca de métodos:

```
"void| execute#|*"
```

Observe que o formato de chaves que estamos utilizando inclui o caractere `'#'` separando o nome do método dos seus argumentos. Esta é mais uma característica necessária para garantir a corretude durante uma busca. O caractere `'#'` marca o início da lista de argumentos. Caso não utilizássemos esse caractere, correríamos o risco de casar o nome do método com um argumento de uma chave de busca. Dado o *pointcut* abaixo:

```
"execution(void *->*(long))"
```

Se não utilizássemos o caractere '#', teríamos uma chave de busca `"void| *| long "`, que casa com a suposta chave identificadora do método `execute`: `void| execute| int| long "`, claramente um erro. O mesmo não ocorre se usamos um caractere adicional após o nome do método: `"void| *#| long "`. Essa expressão de busca não casa com a chave `"void| execute#| int| long "`, como era esperado.

As chaves de métodos são, assim, geradas no formato que descrevemos. As chaves de construtores são geradas usando um formato similar. Por exemplo, a chave de um construtor de uma classe `Pojo`, cuja assinatura é `Pojo(int, long)`, é da forma:

```
"new#| int| long  "
```

Esse formato também é aplicado para chaves identificadoras de chamadas. Como exemplo, uma chamada ao construtor `Pojo(int)`, da classe pojo, será identificada pela chave `"Pojo| new# int "`. Caso o construtor e a classe possuam anotações, geramos mais chaves para identificar a mesma chamada, como, por exemplo: `@jboss.aop.Aspect| new# int "`, `Pojo| @java.lang.Deprecated#| int "` e `"@jboss.aop.Aspect| @java.lang.Deprecated#| int "`.

C. Pointcuts Experimentais

Neste apêndice incluímos a lista completa dos *pointcuts* utilizados nos experimentos da Seção 8.2 (página 115).

```
"execution(* *->*(..))"

"execution(*->new(*))"

"field(* *->*)"

"execution(void *->set*(*))"

"execution(* *->get*())"

"execution(public void *->lostOwnership( \\
    java.awt.datatransfer.Clipboard, \\
    java.awt.datatransfer.Transferable))"

"field(private java.awt.Image *->image)"

"execution(* org.jhotdraw.gui.JSheet->*(..))"

"execution(org.jhotdraw.geom.Dimension2DDouble->new(*))"

"field(* org.jhotdraw.geom.Insets2D->*)"

"execution(void org.jhotdraw.geom.Insets2D->set*(*))"

"execution(* org.jhotdraw.geom.Insets2D->get*())"

"execution(public void
   org.jhotdraw.gui.datatransfer.CompositeTransferable-> \\
   lostOwnership(java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable))"
```

```
"field(private * org.jhotdraw.samples.svg.figures.SVGImageFigure-> \\
   imageData)"

"execution(* *.gui.JSheet->*(..))"

"execution(*imension2DDouble->new(*))"

"field(* *otdraw.geom.Insets2D->*)"

"execution(void *.Insets2D->set*(*))"

"execution(* *.geom.Insets2D->get*())"

"execution(public void *transfer.CompositeTransferable-> \\
   lostOwnership(java.awt.datatransfer.Clipboard,
   java.awt.datatransfer.Transferable))"

"field(private * *samples.svg.figures.SVGImageFigure->imageData)"

"execution(* *jhotdraw.*->*(..))"

"execution(*Dimension2*->new(*))"

"field(* *geom.I*->*)"

"execution(void *geom.*->set*(*))"

"execution(* *org.jhotdraw.geom.*->get*())"

"execution(public void *datatransfer.C*->lostOwnership( \\
   java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable))"

"field(private * *org.jhotdraw.samples.svg.figures.SVGImageFigure*-> \\
   imageData)"

"execution(* org.*.gui*JSheet->*(..))"

"execution(*jhotdra*imension2D*->new(*))",
```

```
"field(* *.jhotd*m*2*->*)"

"execution(void o*w.*eo*.I*s*t*D->set*(*))"

"execution(* *o*w.*eo*.I*s*t*D*->get*())"

"execution(public void *jhotdraw*.datatransfer.*->lostOwnership( \\
   java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable))"

"field(private * *jhotdraw*sam*s.*.figures.*Image*->imageData)"

"execution(* org.jhotdraw.gui..->*(..))"

"execution(org.jhotdraw.geom..->new(*))"

"field(* org.jhotdraw.geom..->*)"

"execution(void org.jhotdraw.geom..->set*(*))"

"execution(* org.jhotdraw.geom..->get*())"

"execution(public void org.jhotdraw.gui.datatransfer..-> \\
   lostOwnership(java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable))"

"field(private * org.jhotdraw.samples.svg.figures..->imageData)"

"execution(* $instanceof{java.awt.datatransfer.Transferable}->*(..))"

"execution($instanceof{java.awt.datatransfer.Transferable}->new(*))"

"field(* $instanceof{java.awt.datatransfer.Transferable}->*)"

"execution(void $instanceof{java.awt.datatransfer.Transferable}-> \\
   set*(*))"

"execution(* $instanceof{java.awt.datatransfer.Transferable}->get*())"

"execution(public void $instanceof{
```

```
   java.awt.datatransfer.Transferable}->lostOwnership( \\
   java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable))"

"field(private * $instanceof{ \\
   org.jhotdraw.samples.svg.figures.SVGFigure}->imageData)"

"execution(* $instanceof{java.awt.datatransfer.*}->*(..))"

"execution($instanceof{java.awt.datatransfer.*}->new(*))"

"field(* $instanceof{java.awt.datatransfer.*}->*)"

"execution(void $instanceof{java.awt.datatransfer.*}-> set*(*))"

"execution(* $instanceof{java.awt.datatransfer.*}->get*())"

"execution(public void $instanceof{java.awt.datatransfer.*}-> \\
   lostOwnership(java.awt.datatransfer.Clipboard,
   java.awt.datatransfer.Transferable))"

"field(private * $instanceof{org.jhotdraw.samples.svg.figures.*}-> \\
   imageData)"

"execution(* $instanceof{*}->*(..))"

"execution($instanceof{*}->new(*))"

"field(* $instanceof{*}->*)"

"execution(void $instanceof{*}->set*(*))"

"execution(* $instanceof{*}->get*())"

"execution(public void $instanceof{*}->lostOwnership( \\
   java.awt.datatransfer.Clipboard,
   java.awt.datatransfer.Transferable))"

"field(private * $instanceof{*}->imageData)"
```

```
"execution(* $instanceof{java.lang.Object}->*(..))"

"execution($instanceof{java.lang.Object}->new(*))"

"field(* $instanceof{java.lang.Object}->*)"

"execution(void $instanceof{java.lang.Object}->set*(*))"

"execution(* $instanceof{java.lang.Object}->get*())"

"execution(public void $instanceof{java.lang.Object}->lostOwnership( \\
   java.awt.datatransfer.Clipboard,
   java.awt.datatransfer.Transferable))"

"field(private * $instanceof{java.lang.Object}->imageData)"

"call(* *->*(..))"

"call(*->new(*))"

"call(void *->set*(*))"

"call(* *->get*())"

"call(public java.lang.String *->getUndoPresentationName())"

"call(* org.jhotdraw.gui.JSheet->*(..))"

"call(org.jhotdraw.geom.Dimension2DDouble->new(*))"

"call(void org.jhotdraw.geom.Insets2D->set*(*))"

"call(* org.jhotdraw.geom.Insets2D->get*())"

"call(public void \\
   org.jhotdraw.gui.datatransfer.CompositeTransferable-> \\
   lostOwnership(java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable))"

"call(* org.jhotdraw.gui..->*(..))"
```

```
"call(org.jhotdraw.geom..->new(*))"

"call(void org.jhotdraw.geom..->set*(*))"

"call(* org.jhotdraw.geom..->get*())"

"call(java.lang.String javax.swing.undo..->getUndoPresentationName())"

"call(* $instanceof{java.awt.datatransfer.Transferable}->*(..))"

"call($instanceof{java.awt.datatransfer.Transferable}->new(*))"

"call(void $instanceof{java.awt.datatransfer.Transferable}->set*(*))"

"call(* $instanceof{java.awt.datatransfer.Transferable}->get*())"

"call(public java.lang.String $instanceof{ \\
   javax.swing.undo.UndoableEdit}->getUndoPresentationName())"

"call(* $instanceof{java.awt.datatransfer.*}->*(..))"

"call($instanceof{java.awt.datatransfer.*}->new(*))"

"call(void $instanceof{java.awt.datatransfer.*}->set*(*))"

"call(* $instanceof{java.awt.datatransfer.*}->get*())"

"call(public java.lang.String $instanceof{javax.swing.undo.*}-> \\
   getUndoPresentationName())",

"call(* $instanceof{*}->*(..))"

"call($instanceof{*}->new(*))"

"call(void $instanceof{*}->set*(*))"

"call(* $instanceof{*}->get*())"

"call(public java.lang.String $instanceof{*}-> \\
```

```
   getUndoPresentationName())"

"call(* $instanceof{java.lang.Object}->*(..))"

"call($instanceof{java.lang.Object}->new(*))"

"call(void $instanceof{java.lang.Object}->set*(*))"

"call(* $instanceof{java.lang.Object}->get*())"

"call(public java.lang.String $instanceof{java.lang.Object}-> \\
   getUndoPresentationName())"

"call(* *->*(..)) AND within(org.jhotdraw.geom.*)"

"call(*->new(*)) AND \\
   withincode(void org.jhotdraw.samples.svg.Main->main(..))"

"call(void *->set*(*)) AND within(net.n3.nanoxml.XMLElement)"

"call(* *->get*()) AND within(net.n3.nanoxml.XMLElement)"

"call(public java.lang.String *->getUndoPresentationName()) \\
   AND within(org.jhotdraw.*)",

"call(* org.jhotdraw.gui.JSheet->*(..)) AND \\
   within(org.jhotdraw.app.action.ExportAction)"

"call(org.jhotdraw.geom.Dimension2DDouble->new(*)) AND withincode( \\
   Dimension2DDouble org.jhotdraw.draw.AbstractCompositeFigure \\
   ->getPreferredSize())"

"call(void org.jhotdraw.geom.Insets2D->set*(*)) AND \\
   within(org.jhotdraw.geom.Insets2D)"

"call(* org.jhotdraw.geom.Insets2D->get*()) AND \\
   within(org.jhotdraw.geom.Insets2D)"

"call( \\
   public void org.jhotdraw.gui.datatransfer.CompositeTransferable-> \\
```

```
   lostOwnership(java.awt.datatransfer.Clipboard, \\
   java.awt.datatransfer.Transferable)) AND \\
   withincode(org.jhotdraw.gui.*->new(..))"

"call(* org.jhotdraw.gui..->*(..)) AND \\
   withincode(org.jhotdraw.gui..->new(..))"

"call(org.jhotdraw.geom..->new(*)) AND \\
   withincode(* org.jhotdraw.geom..->*(*))"

"call(void org.jhotdraw.geom..->set*(*)) AND \\
   within(org.jhotdraw.geom..)"

"call(* org.jhotdraw.geom..->get*()) AND within(org.jhotdraw.geom..)"

"call(java.lang.String javax.swing.undo..-> \\
   getUndoPresentationName()) AND withincode( \\
   void org.jhotdraw.undo.UndoRedoManager->updateActions())"

"call(* $instanceof{java.awt.datatransfer.Transferable}->*(..)) \\
   AND within($instanceof{java.awt.datatransfer.Transferable})"

"call($instanceof{java.awt.datatransfer.Transferable}->new(*)) AND \\
   within($instanceof{java.awt.datatransfer.Transferable})"

"call(void $instanceof{java.awt.datatransfer.Transferable}->set*(*)) \\
   AND within($instanceof{java.awt.datatransfer.Transferable})"

"call(* $instanceof{java.awt.datatransfer.Transferable}->get*()) AND \\
   within($instanceof{java.awt.datatransfer.Transferable})"

"call(public java.lang.String $instanceof{ \\
   javax.swing.undo.UndoableEdit}->getUndoPresentationName()) AND \\
   withincode(private void $instanceof{ \\
   javax.swing.event.UndoableEditListener}->updateActions())"

"call(* $instanceof{java.awt.datatransfer.*}->*(..)) AND \\
   within($instanceof{java.awt.datatransfer.*})"

"call($instanceof{java.awt.datatransfer.*}->new(*)) AND \\
```

```
   within($instanceof{java.awt.datatransfer.*})"

"call(void $instanceof{java.awt.datatransfer.*}->set*(*)) AND \\
   within($instanceof{java.awt.datatransfer.*})"

"call(* $instanceof{java.awt.datatransfer.*}->get*()) AND \\
   within($instanceof{java.awt.datatransfer.*})"

"call(public java.lang.String $instanceof{javax.swing.undo.*}-> \\
   getUndoPresentationName())  AND withincode( \\
   private void $instanceof{javax.swing.event.*}->updateActions())"

"call(* $instanceof{*}->*(..)) AND \\
   withincode($instanceof{*}->new(..))"

"call($instanceof{*}->new(*)) AND withincode(* $instanceof{*}->*(..))"

"call(void $instanceof{*}->set*(*)) AND \\
   withincode($instanceof{*}->new(..))"

"call(* $instanceof{*}->get*()) AND \\
   withincode(* $instanceof{*}->*(..))"

"call(public java.lang.String $instanceof{*}-> \\
   getUndoPresentationName())  AND withincode( \\
   private void $instanceof{*}->updateActions())"

"call(* $instanceof{java.lang.Object}->*(..)) AND \\
   withincode($instanceof{java.lang.Object}->new(..))"

"call($instanceof{java.lang.Object}->new(*)) AND \\
   withincode(* $instanceof{java.lang.Object}->*(..))"

"call(void $instanceof{java.lang.Object}->set*(*)) AND \\
   withincode($instanceof{java.lang.Object}->new(..))"

"call(* $instanceof{java.lang.Object}->get*()) AND \\
   withincode(* $instanceof{java.lang.Object}->*(..))"

"call(public java.lang.String $instanceof{java.lang.Object}-> \\
```

```
getUndoPresentationName()) AND withincode( \\
private void $instanceof{java.lang.Object}->updateActions())"
```

Índice Remissivo

Referências Bibliográficas

[1] AJHotDraw homepage.
http://sourceforge.net/projects/ajhotdraw.

[2] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[3] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters. In *ECOOP*, pages 386–407, 1994.

[4] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Core Design Series. Prentice Hall PTR, 2nd edition, 2003.

[5] Asm homepage.

[6] AspectJ homepage.
http:www.aspectj.org/.

[7] AspectWerkz homepage.
http://aspectwerkz.codehaus.org/.

[8] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An Extensible AspectJ Compiler. In *AOSD - Aspect Oriented Software Development*, pages 87–98, 2005.

[9] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity first: A case for mixing aop and attribute grammars. In *AOSD.08: Technical Track Proceedings*, pages 25–35, 2008.

[10] The Byte Code Engineering Library (BCEL) manual.

[11] Dave Binkley, Mariano Ceccato, Mark Harman, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32:2006.

[12] Dave Binkley, Mariano Ceccato, Mark Harman, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *In Proceedings International Conference on Software Maintenance (ICSM 2005). IEEE Computer Society, Los Alamitos*, pages 27–36. IEEE Computer Society, 2005.

[13] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over Dynamic Properties of Program Execution. In *DAW - Dynamic Aspects Workshop*, pages 71–75, 2005.

[14] Jonas Bonér. AspectWerkz 2: An Extensible Aspect Container. *TheServerSide.COM*, nov 2004.

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, Inc., 1st edition, 1996.

[16] CAESARJ homepage. http://caesarj.org.

[17] S. Chiba, Y. Sato, and M. Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. In *1st Workshop on Advancing the State-of-the-Art in Run-Time Inspection (ECOOP 2003)*, 2003.

[18] Maria Agustina Cibran and Bart Verheecke. Dynamic Aspects for Web Service Management. In *DAW - Dynamic Aspects Workshop*, pages 146–152, 2004.

[19] Maria Agustina Cibran and Bart Verheecke. Dynamic Business Rules for Web Service Composition. In *DAW - Dynamic Aspects Workshop*, pages 13–18, 2005.

[20] Christian Dalager, Simon Jorsal, and Eske Sort. Aspect Oriented Programming in JBoss 4. Master's thesis, IT University of Copenhagen, feb 2004.

[21] Simon Denier and Pierre Cointe. Understanding design patterns density with aspects a case study in jhotdraw using aspectj.

[22] Arie Van Deursen. Ajhotdraw: A showcase for refactoring to aspects. In *In Linking Aspect Technology and Evolution (AOSD-2005*, 2005.

[23] Arie Van Deursen, Marius Marin, and Leon Moonen. A systematic aspectoriented refactoring and testing strategy, and its application to jhotdraw. technical report sen-r0507, centrum voor wiskunde en informatica. Technical report, 2005.

[24] Peter Ebraert and Tom Tourwe. A reflective approach to dynamic software evolution. In Walter Cazolla, editor, *In the proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'04) in conjunction with the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, June 2004.

[25] Martin Fowler. *Patterns of Enterprise Application Architecture*. The Addison-Wesley Signature Series. Addison-Wesley Professional, 1st edition, 2001.

[26] Thomas Fritz, Marc Ségura, Mario Südholt, Egon Wuchner, and Jean-Marc Menaud. An Application of Dynamic AOP to Medical Image Generation. In *DAW - Dynamic Aspects Workshop*, pages 5–12, 2005.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1st edition, 1995.

[28] Celina Gibbs and Yvonne Coady. Garbage Collection in Jikes: Could Dynamic Aspects add Value? In *DAW - Dynamic Aspects Workshop*, pages 56–63, 2004.

[29] Wasif Gilani and Olaf Spinczyk. A Family of Aspect Dynamic Weavers. In *DAW: Dynamic Aspects Workshop*, pages 64–75, 2004.

[30] Philip Greenwood and Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. In *DAW - Dynamic Aspects Workshop*, pages 76–88, 2004.

[31] Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 4th edition, 2003.

[32] Patrick Hill, Simon Holland, and Robin C. Laney. Using Dynamic Aspects in Music Composition Systems. In *DAW - Dynamic Aspects Workshop*, pages 89–97, 2004.

[33] Christian Hofmann, Robert Hirschfeld, and Jeff Eastman. Flexible Call-by-call Settlement — An Opportunity for Dynamic AOP. In *DAW - Dynamic Aspects Workshop*, pages 19–25, 2005.

[34] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel. pages 24–35.

[35] JAC project home page.
http://jac.aopsys.com.

[36] JARP.
http://jarp.sourceforge.net/.

[37] Javassist.
http://www.csg.is.titech.ac.jp/ chiba/javassist.

[38] JBoss AOP homepage.
http:labs.jboss.com/portal/jbossaop/.

[39] Java Compiler Compiler.
https://javacc.dev.java.net/.

[40] JHotDraw homepage.
http://www.jhotdraw.org/.

[41] The Jikes Research Virtual Machine User's Guide.

[42] Java Object Oriented Neural Engine.
http://sourceforge.net/projects/joone.

[43] JStock - KLSE Real-Time Monitor.
http://sourceforge.net/projects/jstock.

[44] Java Virtual Matchine Debug Interface.
http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html.

[45] Java Virtual Machine Interface Reference.
http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

[46] Olivier Kaczor, Yann gaël Guéhéneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR*, pages 22–24, 2006.

[47] Peter Kenens, Sam Michiels, Frank Matthijs, Bert Robben, Eddy Truyen, Bart Vanhaute, Wouter Joosen, and Pierre Verbaeten. An AOP Case with Static and Dynamic Aspects. In *ECOOP Workshops*, pages 428–430, 1998.

[48] Mik Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA*, pages 340–352, 1999.

[49] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[50] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[51] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1997.

[52] John Lamping, Gregor Kiczales, Luis Rodriguez, and Erik Ruf. An Architecture for an Open Compiler.

[53] Craig Larman. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, 2nd edition, 2002.

[54] K. Lieberherr, J. Palm, and R. Sundaram. Expressiveness and complexity of crosscut languages. In *FOAL*, 2005.

[55] Cristina Videira Lopes and Walter L. Hürsch. Separation of Concerns, 1995.

[56] M Marin. Refactoring jhotdraw's undo concern to aspectj. In *In Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE2004*, 2004.

[57] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pierre Verbaeten. Aspects should not die.

[58] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Porgramming Languages*. PhD thesis, Vom Fachbereich Informatik der Technischen Universität Darmstadt, Darmstadt, Alemanha, 2005.

[59] Isabel Michiels, Theo D'Hondt, Kris De Schutter, and Ghislain Hoffman. Using Dynamic Aspects to Distill Business Rules from Legacy Code. In *DAW - Dynamic Aspects Workshop*, pages 98–102, 2004.

[60] Luis Daniel Benavides Navarro, Mario Sudholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvee. Explicitly Distributed AOP Using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM Press.

[61] Angela Nicoara and Gustavo Alonso. Dynamic aop with prose. In *ASMEA*, 2005.

[62] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *EuroSys*, 2008.

[63] Klaus Ostermann and Mira Mezini. Conquering Aspects With Caesar. In *AOSD - Aspect Oriented Software Development*, pages 90–99, 2003.

[64] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *International Conference on Reflection*, pages 1–24, 2001.

[65] A. Popovici, T. Gross, and G. Alonso. Dynamic Homogenous AOP with PROSE. Technical report, ETH Zurich, Department of Computer Science, March 2001.

[66] Andrei Popovici, Thomas R. Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD*, pages 141–147, 2002.

[67] Andrei Popovici, Thomas R. Gross, and Gustavo Alonso. Just in time aspects: Efficient dynamic weaving for java. In *AOSD*, pages 100–109, 2003.

[68] Prose homepage.
http:prose.ethz.ch/.

[69] Yoshiki Sato and Shigeru Chiba. Negligent Class Loaders for Software Evolution. In *ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE)*, pages 53–60, 2004.

[70] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, Inc., 2000.

[71] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley Longman Inc., 4th edition, 1999.

[72] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Professional, 3 edition, 1998.

[73] Davy Suvée and Wim Vanderperren. JAsCo: An Aspect-Oriented approach tailored for Component Based Software Development. In *AOSD - Aspect Oriented Software Development*, pages 21–29, 2003.

[74] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, pages 107–119, 1999.

[75] Wim Vanderperren and Davy Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *DAW: Dynamic Aspects Workshop*, pages 120–134, 2004.

[76] Alexandre Vasseur. Dynamic AOP and Runtime Weaving for Java—How does AspectWerkz Address It? In *DAW: Dynamic Aspects Workshop*, pages 135–145, 2004.

[77] Honda Y. and Tokoro M. Soft Real-Time Programming through Reflection. In *International Workshop on Reflection and Meta-Level Architecture*, pages 12–23, 1992.

[78] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, New York, NY, 1992. ACM Press.