Universidade de São Paulo
Instituto de Matemática e Estatística

MAC 5701 Tópicos em Ciências da Computação

Programação Orientada a Aspectos Dinâmica

Flávia Rainone
fla@ime.usp.br

Orientador: Francisco Reverbel
reverbel@ime.usp.br

São Paulo, 07 de junho de 2005

1.Introdução

A programação orientada a aspectos (POA) tem se popularizado nos últimos anos como uma solução para o encapsulamento de funcionalidades ortogonais. O uso de tecnologias que fornecem suporte a esse novo paradigma já é uma realidade tanto no ambiente acadêmico como na indústria.

A POA é vista por muitos profissionais como uma evolução do paradigma de orientação a objetos. Nesse último, encapsulamos funcionalidades em objetos. Cada objeto tem um objetivo simples e claro, resultante da funcionalidade que ele encapsula. Quando precisamos implementar uma funcionalidade ortogonal utilizando esse paradigma, somos obrigados a espalhar código relacionado a essa funcionalidade por vários objetos. Mesmo que tal funcionalidade ortogonal seja encapsulada em um objeto, ela deve ser aplicada em diversos instantes do sistema. Isso implica que os vários objetos responsáveis pela execução desses instantes terão que ser alterados para invocar uma funcionalidade ortogonal. Conseqüentemente, esses objetos passam a ser sobrecarregados em sua função e ficam responsáveis também por essa nova funcionalidade.

Com a programação orientada a aspectos, podemos encapsular uma funcionalidade ortogonal em uma nova unidade: um aspecto. Esse encapsulamento é feito de tal forma que garantimos que a funcionalidade será aplicada nos instantes corretos da execução do sistema sem que os outros objetos que executam tais instantes tenham que ser alterados.

Um dos desafios de programação orientada a aspectos é o desenvolvimento de ferramentas que habilitem a programação orientada a aspectos dinâmica em sistemas escritos em Java. Com esse recurso, é possível aplicar e remover aspectos a um sistema em tempo de execução, sem a necessidade de reiniciar tal sistema.

2.Programação Orientada a Aspectos

A programação orientada a aspectos introduz novos conceitos à área da computação. Dentre eles, o principal é o aspecto, que, como já foi dito, é a unidade de encapsulamento de uma funcionalidade ortogonal.

Mas como uma funcionalidade ortogonal é encapsulada em um aspecto? Sabemos que funcionalidades são implementadas em blocos de código, que manipulam dados. Em orientação a objetos, encapsulamos uma funcionalidade em um objeto, que contém métodos e campos. Os métodos contêm o código que implementa a funcionalidade, e os campos, os dados manipulados pelos primeiros. Em orientação a aspectos, também temos que encapsular uma funcionalidade. Ainda que ortogonal, essa também pode ser implementada na forma de blocos de código. Tais blocos são conhecidos como *advices* e estão contidos em um aspecto.

Além disso, é preciso identificar em quais pontos do sistema os *advices* deverão ser executados. Tais pontos são identificados por outro componente do aspecto, um *pointcut*, que é uma expressão cuja sintaxe depende da ferramenta de POA utilizada. Tal expressão se refere a instantes ou momentos da execução do sistema, denominados *joinpoints*. Esses podem ou não correponder a um ponto do código. Quando a expressão do *pointcut* é da forma: “sempre que o método `b()` da

classe `A` for executado”, podemos facilmente discernir os *joinpoints* identificados por tal expressão como sendo o resultado da execução de um ponto exato do sistema : o método `b()` da classe `A`. Mas se o *pointcut* for mais restrito, como: “sempre que o método `b()` da classe `A` for executado e que a variável booleana `x` tiver valor verdadeiro”, não podemos dizer os *joinpoints* identificados por tal expressão equivalem à execução de um ponto do código. Similarmente, uma linha de código que invoca um método corresponde a vários *joinpoints* de chamada de método, um para cada vez que a linha é executada. Dizemos que um *joinpoint* é um momento da execução do sistema.

A estrutura principal de um aspecto se resume a *advices* e *pointcuts*, onde os últimos identificam quando os primeiros deverão ser aplicados.

Porém, *advices* e *pointcuts* não são os únicos conceitos associados à programação orientada a aspectos. Uma funcionalidade ortogonal pode ser implementada com mais facilidade se for possível adicionar alguns métodos auxiliares a objetos existentes do sistema. Esses métodos auxiliares não têm relação alguma com as funcionalidades representadas pelos objetos afetados e sim com a funcionalidade ortogonal. Mas a adição de um ou mais métodos a um objeto existente nos leva a uma pergunta: quem seria encarregado de responder por tais invocações? Certamente, se o objeto alterado não possuía tais métodos originalmente, ele não têm condições de responder por essas invocações. O aspecto que encapsula a funcionalidade ortogonal em questão é responsável por fornecer um objeto que implemente os métodos adicionados e que responda por tais invocações. Esse objeto é denominado *mixin*. Além disso, um conjunto de métodos é, conceitualmente, uma interface. Desse modo, a adição de um conjunto de métodos a objetos do sistema é denominada *introdução de interfaces*.

Finalmente, outro conceito da programação orientada a aspectos é a *introdução de meta-dados*. Meta-dados são quaisquer informações relacionadas a um ponto do código e podem ser introduzidos no sistema para auxiliar a execução dos aspectos. Os meta-dados associados ao ponto em que o aspecto está executando podem ser utilizados pelo próprio aspecto para tomada de decisões.

A identificação dos pontos do sistema aos quais devem ser aplicados introduções de meta-dados, introduções de interfaces e *mixins* são também identificados por *pointcuts* contidos em aspectos.

Conseqüentemente, podemos concluir que um aspecto encapsula uma funcionalidade ortogonal, que é implementada através dos *pointcuts*, *advices*, *mixins*, introduções de interfaces e de meta-dados contidos no aspecto.

3.Programação Orientada a Aspectos em Java

A programação orientada a aspectos em Java é disponibilizada por várias ferramentas, como o AspectJ[AJProj], JBoss AOP[JBAOP], AspectWerkz[AWZ] e PROSE [PROSE].

A mais conhecida e utilizada é o AspectJ. Tal ferramenta incrementa a linguagem Java com novas palavras chaves. No AspectJ, um aspecto é nada mais do que uma classe, exceto que é declarado com a palavra chave `aspect`, ao invés de `class`. O *advices* são métodos com uma sintaxe especial, que os associa ao *pointcut* que deverá ser utilizado para identificar os *joinpoints* de aplicação dos

mesmos. O AspectJ fornece uma extensa gama de opções para definição de *pointcuts*.
Entretanto, o AspectJ não fornece suporte a programação orientada a aspectos dinâmica.
Muitas ferramentas suportam a programação orientada a aspectos em Java sem alterar linguagem. Para permitir que o usuário declare aspectos, *advices*, *pointcuts* e outros sem o uso de novas palavras chaves, outros meios são utilizados, como arquivos de xml, anotações[JAnnot] e *xdoclets* [XDCLT].
Independente de como os aspectos e *advices* são definidos, a grande maioria dessas ferramentas atinge o seu objetivo através da manipulação de *bytecodes*[1].

4.Ferramentas de POA Dinâmica

As ferramentas de POA Dinâmica estão adquirindo uma importância crescente na computação. A possibilidade da adição e remoção de aspectos em tempo de execução adiciona novos horizontes para a flexibilidade dos sistemas escritos em Java.
Veremos algumas dessas ferramentas e como elas implementam os mecanismos de POA dinâmica nas próximas sessões.

4.1.JBoss AOP

O JBoss AOP é a ferramenta de programação orientada a aspectos fornecida e utilizada pelo JBoss [JBOSS].
Com o JBoss AOP, um aspecto é uma classe comum Java. Para que a classe passe a ser um aspecto basta declará-la como tal em um arquivo denominado `jboss-aop.xml` ou anotá-la com `org.jboss.aop.Aspect`.
Advices são métodos de um aspecto e recebem como argumento uma instância de `org.jboss.aop.joinpoint.Invocation` ou de uma subclasse.
Para configurar quais pontos do código deverão ser interceptados por quais *advices*, basta declarar no arquivo de configuração um *bind*:

```
<bind pointcut=”execution(POJO->new())”>
    <advice name=”constructorAdvice” class=”MyAspect”/>
</bind>
```

O *bind* é utilizado para associar uma expressão *pointcut* a um ou mais *advices* que deverão ser invocados em todos os pontos do sistema identificados pela expressão *pointcut*. O exemplo anterior declara que a classe `MyAspect` é um aspecto e contém um *advice* denominado `constructorAdvice`, a ser invocado sempre que o construtor da classe POJO for executado.
Outra forma de associar um *advice* a um *pointcut* é através de anotações:

```
@Aspect
public class MyAspect
{
    @Bind (pointcut=”execution(POJO->new())”)
```

1 *Bytecodes* é o conteúdo dos arquivos gerados por um compilador Java. Os *bytecodes* contêm instruções de código portável, que podem ser processadas por qualquer máquina virtual Java.

```
    public Object constructorAdvice(ConstructorInvocation invocation) throws
Throwable
    {
        ...
    }
}
```

No código acima, vemos o aspecto `MyAspect`, e seu *advice* `constructorAdvice`. A anotação `Aspect` indica que `MyAspect` é um aspecto. A anotação `Bind`, que `constructorAdvice` é um *advice* que deve ser invocado sempre que o construtor de `POJO` for executado.

É possível também utilizar *xdoclets*, no caso de o usuário querer utilizar o recurso de anotações com uma versão Java anterior ao Java 5:

```
/**
 * @@Aspect
  */
public class MyAspect
{
    /**
     *  @@Bind (pointcut=”execution(POJO->new())”)
     */
    public Object constructorAdvice(ConstructorInvocation invocation) throws
Throwable
    {
        ...
    }
}
```

Note que a diferença entre esse exemplo e o anterior é apenas o fato de as anotações estarem contidas em comentários *javadocs* e de serem utilizados dois caracteres `'@'` ao invés de um. Naturalmente, o uso de *xdoclets* ao invés de anotações possui algumas desvantagens, como a ausência de checagem das anotações feita por compiladores. Além disso, essa opção exige que o usuário processe as suas classes por um compilador de anotações provido pelo JBoss AOP. Isso deve ser feito antes de tudo, inclusive antes do processamento das classes para a instrumentação.

O JBoss AOP fornece ainda um tipo adicional de aspecto, o interceptador. Um interceptador é um aspecto que possui um único *advice*, o método `invoke`, cuja assinatura é definida pela interface:

```
public interface Interceptor
{
    public String getName();
    public Object invoke(Invocation invocation) throws Throwable;
}
```

Interceptadores são muito utilizados pelo servidor de aplicação do JBoss, o *JBoss Application Server* [JBAS].

Toda classe é processada pelo JBoss AOP antes de ser carregada na memória por um *class loader*[2]. O processamento de uma classe pode resultar em uma transformação de seus *bytecodes* se o JBoss AOP considerar necessário. Para tanto, todos os pontos do código da classe são analisados (execução de construtores, invocações de métodos, leituras de campos e etc). São instrumentados os pontos cuja execução puder resultar em um ou mais *joinpoints* identificados por uma expressão

---

2 Instância da classe `java.lang.ClassLoader`.

pointcut. Dizemos que tais pontos são identificados (*matched*) por expressões *pointcuts*. Quando um ponto é instrumentado, o código que o executa é substituído por um bloco de código que executa uma pilha de interceptadores e, ao final da pilha, executa o *joinpoint* propriamente dito. Desse modo, quando um código instrumentado pelo JBoss AOP for executado, os interceptadores adequados serão aplicados aos pontos identificados por expressões *pointcuts*.

Além de *pointcuts*, aspectos e *advices*, com o JBoss AOP é possível lançar mão de introduções de interfaces, *mixins* e meta-dados para a implementação de uma funcionalidade ortogonal. Ademais, ele inclui outras funcionalidades, como a "introdução de anotações", através da qual podemos adicionar uma anotação a uma classe, a um método, a um campo ou a um construtor. Esse recurso pode ser interpretado como uma forma de "introdução de meta-dados", no sentido em que pode prover ao aspecto meta-dados sobre um determinado *joinpoint*, permitindo a tomada de decisões baseada nas informações obtidas. Porém, as anotações são um recurso da linguagem Java e, como tal, podem ser utilizadas por outros arcabouços e ferramentas. Com isso, é possível utilizar o JBoss AOP para introduzir anotações que serão processadas por outras ferramentas, como uma nova forma de comunicação de dados entre arcabouços.

No JBoss AOP, todos esses elementos são representado por objetos contidos em uma instância *singleton* (vide padrão Singleton (127) [GOF]) de `org.jboss.aop.AspectManager`. Essa é a principal classe do JBoss AOP e é também a fachada do sistema.

É através dela que as principais operações de POA dinâmica podem ser realizadas:

- `AspectManager.addBinding(org.jboss.aop.advice.AdviceBinding)`
- `AspectManager.removeBinding(String)`
- `AspectManager.removeBindings(ArrayList)`

Essas operações permitem a adição e remoção de objetos do tipo `org.jboss.aop.advice.AdviceBinding` ao sistema. Essa classe é utilizada para representar os *binds* declarados nos arquivos `jboss-aop.xml`. Com essas operações, o usuário pode adicionar novos interceptadores e aspectos em tempo de execução. Basta criar uma instância dessa classe, contendo um grupo de interceptadores e a expressão *pointcut* que deve ser satisfeita para que eles sejam invocados. Para que o novo *bind* seja aplicado, é preciso adicionar o objeto criado ao `AspectManager` através do método `addBinding`. O resultado dessa operação é a aplicação dos dos novos interceptadores nos pontos identificados pela expressão *pointcut*. É possível também utilizar o método `addBinding` para redefinir *binds* previamente registrados.

Além dessas operações, um ou mais interceptadores podem ser aplicados a um 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. Então, se o objeto alvo da adição de interceptadores é da classe:

```
class A
{
    private int myNumber;
    public String myName;
    public method1() {...}
    public method2(String text) {...}
```

```
}
```

Os interceptadores serão invocados sempre que:

- os métodos `method1` e `method2` forem executados
- os valores dos campos `myNumber` e `myName` forem lidos ou definidos por algum método ou construtor de qualquer objeto do sistema;

Para tomar conhecimento de como adicionar interceptadores a um objeto, é preciso entender as alterações aplicadas às classes instrumentadas pelo JBoss AOP. Cada classe instrumentada passa a implementar a interface `org.jboss.aop.Advised`:

```
public interface InstanceAdvised
{
    public InstanceAdvisor _getInstanceAdvisor();
    public void _setInstanceAdvisor(InstanceAdvisor advisor);
}
public interface Advised extends InstanceAdvised
{
    public Advisor _getAdvisor();
}
```

Através dos métodos definidos pela interface `org.jboss.aop.InstanceAdvised`, herdados por `org.jboss.aop.Advised`, podemos obter o `InstanceAdvisor` associado a uma uma instância de uma classe instrumentada. Com ele, é possível adicionar e remover interceptadores a um objeto. Veja os métodos disponibilizados por `InstanceAdvisor`:

- `public void insertInterceptor(Interceptor interceptor);`
- `public void removeInterceptor(String name);`
- `public void appendInterceptor(Interceptor interceptor);`
- `public void insertInterceptorStack(String stackName);`
- `public void removeInterceptorStack(String stackName);`
- `public void appendInterceptorStack(String stackName);`

A pilha de interceptadores a ser aplicada a um *joinpoint* é, na verdade, constituída de três pilhas: aquela que contém os interceptadores inseridos na instância associada ao *joinpoint* (`insertInterceptor`); a que contém os interceptadores associados ao *joinpoint* através de *binds*; e, finalmente, a que contém os interceptadores adicionados ao final da pilha (`appendInterceptor`). É através das operações de `InstanceAdvisor` e de `AspectManager` que a POA dinâmica pode ser utilizada no JBoss AOP.

Mas como tudo isso funciona? O JBoss AOP implementa a POA dinâmica instrumentando totalmente os *joinpoints* identificados por uma ou mais expressões de *pointcuts*. São somente esses que podem ser afetados em decorrência de uma operação de POA dinâmica, realizada em tempo de execução. É possível fazer com que *joinpoints* que não são instrumentados inicialmente (ou seja, não são identificados por *pointcuts*) possam ser interceptados através da POA dinâmica. Para isso, basta declarar uma ou mais expressões `prepare` que identifiquem tais joinpoints:

```
<prepare expr="all(*POJO)" />
```

O elemento acima, se adicionado ao arquivo `jboss-aop.xml`, fará com que todos os *joinpoints* associados à classe POJO sejam passíveis de interceptação em tempo de execução. Com isso, mesmo aqueles que não são interceptados inicialmente por nenhum aspecto, poderão passar a sê-lo

em tempo de execução, através de uma operação de POA dinâmica.

Outro modo de fazer isso é anotar a classe POJO como se segue:

```
import org.jboss.aop.Prepare;
@Prepare
class POJO {
 ....
}
```

Uma expressão `prepare` é registrada internamente pelo `AspectManager` como uma expressão *pointcut* comum. O resultado disso é que o sistema não faz distinção entre os *pointcuts* associados a *binds* e as expressões `prepare`.

Como foi detalhado há pouco, o JBoss AOP instrumenta os pontos identificados por qualquer expressão *pointcut* contida no `AspectManager`. Assim sendo, pontos identificados por expressões `prepare` também serão instrumentados, mesmo que não existam interceptadores a serem aplicados em tais pontos, isto é, mesmo que a pilha de interceptadores seja vazia. Com esse mecanismo, quando um método de POA dinâmica é invocado, basta adicionar os interceptadores adequados às pilhas adequadas, e eles serão automaticamente invocados em tempo de execução.

Com isso, fica mais claro porque pontos do código que não são identificados por expressões *pointcut* não podem ser interceptados como resultado de operações de POA dinâmica. Considerando que eles não foram instrumentados antes de serem carregados na máquina virtual, eles não invocam pilha de interceptadores alguma.

Chamamos o modo de instrumentação de classes utilizado pelo JBoss AOP como modo estático de instrumentação, já que as classes são instrumentadas uma única vez, antes de serem executadas.

4.2.AspectWerkz

No AspectWerkz, um aspecto também é representado por uma classe Java. Os *advices* equivalem a métodos de um aspecto que recebem um único argumento, do tipo `org.codehaus.aspectwerkz.joinpoint.JoinPoint`.

Do mesmo modo que com o JBoss AOP, é preciso indicar para o o AspectWerkz quais classes são aspectos. Para isso, basta fazer uma declaração em um arquivo xml:

```
<aspectwerkz>
    <system id="MySystem">
         <aspect class="MyAspect"/>
    </system>
</aspectwerkz>
```

O elemento `aspect` acima declara que a classe `MyAspect` é um aspecto.

Para associar os *advices* de `MyAspect` a uma expressão *pointcut*, é preciso adicionar os elementos `pointcut` e `advice` ao arquivo xml:

```
<aspect class="MyAspect">
    <pointcut name="somePointcut" expression="execution(POJO.(..))"/>
    <advice name="someAdvice" type="before" bind-to="somePointcut"/>
</aspect>
```

No trecho acima, o *pointcut* `somePointcut` é declarado como membro de `MyAspect`. Em seguida,

o método `someAdvice` da classe `MyAspect` é declarado como um *advice* e tem a sua execução associada ao *pointcut* `somePointcut`. Note o atributo `type` dentro de `advice`. Ele indica o tipo de `advice`. Esse recurso é fornecido tanto pelo AspectWerkz quanto pelo AspectJ e permite que o usuário indique quando o *advice* deverá ser aplicado: "antes" (`before`), "depois" (`after`) ou "durante" (`around`) a execução dos *joinpoints* identificados pela expressão *pointcut*. No modo "durante", o *advice* é invocado antes da execução do pointcut e é responsável por invocar a execução do ponto interceptado. Uma vez executado o ponto interceptado, o *advice* pode acessar o que esse ponto retornou (somente se o ponto retornar um valor; isso ocorre em pontos que são invocações de métodos com valor de retorno, de construtores, leituras de valores de campos, etc).

Já no JBoss AOP, não existem tipos de *advices*, e todos os *advices* e interceptadores são executados de forma equivalente ao modo "durante". Todo *advice* é responsável por executar o próximo elemento da pilha de interceptadores ou o próprio ponto interceptado através do método `Invocation.invokeNext()`.

Outra forma de configurar um aspecto no AspectWerkz é através de anotações:

```
@Aspect
public class MyAspect
{
    @Before("execution(POJO.(..))")
    public void someAdvice(JoinPoint joinPoint)
    {
        ...
     }
}
```

No código acima, a mesma configuração vista no arquivo xml é feita através de anotações. O AspectWerkz permite ainda a utilização de javadoc para configuração. Nesse modo, as anotações são colocadas dentro de comentários javadoc, e podem ser processadas em ambientes que utilizam Java versão 1.4 ou anterior. Para que as configurações contidas em comentários javadocs sejam visíveis, é necessário um pré-processamento das classes por um compilador, fornecido pelo AspectWerkz. Esse mecanismo é bem similar ao uso de *XDoclets* pelo JBoss AOP.

As operações de POA dinâmica são providas por uma série de métodos estáticos, através da classe `org.codehaus.aspectwerkz.transform.inlining.deployer.Deployer`:

- `DeploymentHandle deploy(Class aspect)`
- `DeploymentHandle deploy(Class aspect, ClassLoader deployLoader)`
- `DeploymentHandle deploy(Class aspect, DeploymentScope scope, ClassLoader deployLoader)`
- `DeploymentHandle deploy(Class aspect, String xmlDef, ClassLoader deployLoader)`
- `DeploymentHandle deploy(Class aspect, String xmlDef, DeploymentScope scope)`
- `DeploymentHandle deploy(Class aspect, String xmlDef, DeploymentScope scope, ClassLoader deployLoader)`
- `DeploymentHandle undeploy(Class aspect)`
- `DeploymentHandle undeploy(Class aspect, ClassLoader loader)`
- `DeploymentHandle undeploy(DeploymentHandle deploymentHandle)`

Esses métodos realizam a adição (*hot deployment*) e remoção (*hot undeployment*) de aspectos em tempo de execução. O primeiro método simplesmente adiciona o aspecto `aspect` ao sistema. Os componentes do aspecto adicionado têm que ser configurados pela própria classe `aspect`, através de anotações. O alvo desse método são as classes carregadas pelo mesmo *class loader* de `aspect`. O segundo método permite que seja especificado o *class loader* cujas classes serão afetadas pela operação. Já o terceiro, permite a utilização de um escopo de implantação (`DeploymentScope`) para restringir a operação de POA dinâmica a certos pontos do código. Um escopo de implantação é uma expressão do tipo *pointcut* e deve ser declarado como no exemplo abaixo:

```
<deployment-scope name="toString" expression="execution(String *.toString())"/>
```

Sempre que o escopo `"toString"` for utilizado, os efeitos de uma operação de POA dinâmica ficarão restritos a somente os métodos `toString()` das classes alvo. Um escopo pode ser declarado também através de anotações. Para utilizar um escopo, ele tem que ser obtido na forma de instância de `DeploymentScope`:

```
SystemDefinition.getDefinitionFor(loader, systemId).getDeploymentScope("toString");
```

A primeira parte desse comando obtém a definição de um sistema, responsável por conter as definições feitas pelo usuário no AspectWerkz (observe o sistema declarado no exemplo de configuração através de arquivos xml). A segunda parte executa uma requisição ao sistema para obter o escopo de implantação através de seu nome, `"toString"`. Uma vez obtido o escopo, ele pode ser passado como argumento para uma operação de *hot deployment*.

Os métodos `deploy` que recebem `"String xmlDef"` como argumento utilizam um arquivo de xml para o *hot deployment* ao invés das anotações contidas na classe do aspecto.

Finalmente, os métodos `undeploy` removem as interceptações resultantes da adição de um aspecto. Quando não é definido o *class loader* ao qual essa operação deverá ser aplicada, o *class loader* associado à classe do aspecto é utilizado.

É possível também desfazer as alterações realizadas por uma execução de um método `deploy`, utlizando-se o último método `undeploy` da lista. Ele deve receber como argumento o `DeploymentHandler` retornado pelo `deploy` que se deseja cancelar.

O AspectWerkz funciona em dois modos de instrumentação: estático (classes são instrumentadas antes de serem carregadas por um *class loader*)e dinâmico.

No entanto as operações de POA dinâmica só estão disponíveis no modo dinâmico. Tais operações são disponibilizadas através da troca de bytecodes em tempo de execução, uma prática conhecida como *hot swap*. É a aplicação dessa técnica que caracteriza a instrumentação como dinâmica, pois isso permite que classes sejam instrumentadas em tempo de execução.

De forma similar ao JBoss AOP, somente os pontos identificados por expressões *pointcuts* configuradas na inicialização do sistema são alvos da POA dinâmica. Assim, todo *joinpoint* ao qual se deseja aplicar aspectos através da POA dinâmica precisa ser identificado por uma expressão *pointcut* registrada no AspectWerkz previamente à execução do sistema (através de anotações ou de um arquivo xml).

Cada ponto identificado por um ou mais *pointcuts* gera informações que são armazenadas no

AspectWerkz. Entre essas informações, encontra-se a lista dos *advices* que devem ser aplicados ao ponto (note que a lista pode ser vazia no caso de o ponto ser identificado por *pointcuts* que não são utilizados por nenhum *advice*). Quando uma operação de POA dinâmica é invocada, apenas esses pontos são analisados em busca de alterações na lista dos *advices*. Após essa análise, os *bytecodes* das classes são alterados, utilizando-se o *hot swap* da JVMTI. Essa alteração é feita de modo a ativar a interceptação dos pontos que passaram a ser afetados por *advices* após a execução da operação de POA dinâmica. Veremos mais detalhes sobre a JVTI no item 5.

4.3.Prose

A ferramenta *PROgramable Service Extensions* (PROSE) utiliza apenas classes Java para habilitar a programação orientada a aspectos. Não é preciso nenhum arquivo xml de configuração nem o uso de anotações.

No PROSE, um aspecto é qualquer subclasse de `ch.ethz.prose.Aspect`. Os únicos componentes de um aspecto são instâncias de `ch.ethz.prose.crosscut.Crosscut`, que chamaremos de *crosscuts*. É nesses objetos que ficam contidos os *advices*, métodos cujo nome é definido de acordo com o tipo de *crosscut* que está sendo utilizado. A assinatura do *advice* pode adicionar restrições aos *joinpoints* que deverão ser interceptados. Por exemplo, é possível especificar os tipos dos argumentos do método cuja execução será interceptada, no caso de um c*rosscut* do tipo `ch.ethz.prose.crosscut.MethodCut`. Além disso, todo *crosscut* precisa implementar o método abstrato:

```
protected abstract PointCutter pointCutter();
```

Esse método deve retornar uma instância de `ch.ethz.prose.filter.PointCutter`, que representam um *poincut*.

Dessa forma, um *advice* é associado a um *pointcut* através de *crosscuts* contidos no aspecto. Veja o exemplo abaixo:

```
public class MyAspect extends Aspect {
    public Crosscut c1 = new MethodCut()
    {
        // advice
        public void METHOD_ARGS(Foo x, ANY y)
        {
            ...
        }
        // pointcut
        protected PointCutter pointCutter()
        { return  (Executions.before() . AND   (Within.method("*")) );}
    };

    // retorna os crosscuts contidos no aspecto.
    protected Crosscut[] crosscuts()
    {
        return new Crosscut[]{c1};
    }
}
```

A classe `MyAspect` é um aspecto e contém apenas um *crosscut*. Esse é do tipo `MethodCut` e é utilizado para interceptar execução de métodos. O *advice* tem o nome definido pelo tipo do *crosscut* que o contém: `METHOD_ARGS`. A lista de argumentos que recebe adiciona a restrição de que o método a ser interceptado deve ser da classe `Foo` e pode possuir quaisquer argumentos (`ANY`). A expressão pointcut retornada por `pointCutter` indica que o *advice* deve ser executado antes de qualquer método. Somando a esse *pointcut* a assinatura de `METHOD_ARGS`, conclui-se que o *advice* será invocado antes da execução de qualquer método de `Foo`.

O PROSE permite que *advices* sejam executados antes e depois de um joinpoint, mas não durante (equivalente ao `around` do AspectWerkz e ao comportamento padrão do JBoss AOP).

A API de *crosscuts* e de pointcuts do PROSE é facilmente estensível para adição de novos tipos de joinpoints.

Essa ferramenta é totalmente voltada para a POA dinâmica e utiliza duas abordagens de interceptação.

A primeira é o uso da JVMDI (*Java Virtual Machine Debugger Interface*[3] [JVMDI]). Os *advices* que têm que ser aplicados a cada ponto são armazenados na memória do PROSE, em uma estrutura de mapeamento de ponto do código por *advice*. Para cada ponto do código alvo de *advices,* um *breakpoint* na JVMDI é registrado para que o PROSE seja notificado da execução do mesmo. Quando o controle é passado para o PROSE, ele encontra os *advices* que têm que ser aplicados, executa cada um deles, e devolve o controle da execução para a máquina virtual.

A JVMDI não é acessada diretamente pelo PROSE. Para a interceptação, uma camada de indireção entre a JVMDI e o PROSE foi criada: o *Java Multiplexing Debugger* (JMD). O JMD é um módulo provido pelo PROSE que permite a depuração simultânea da mesma máquina virtual por vários clientes. Isso permite que o usuário utilize um depurador ao mesmo que tempo que utiliza o PROSE. Além disso, o próprio PROSE possui mais de um cliente para a JVMDI: um cliente para execução de métodos, outro para a instrumentação da leitura de campos, e assim por diante.

Outra alternativa utiliza a API do compilador Jikes [JkIBM], fornecido pela IBM. Essa API fornece a funcionalidade de *hot swap*. Com isso, o PROSE é utiliza *hot swap* para trocar os bytecodes dos pontos que passam a ser interceptados em tempo de execução como resultado das operações de POA dinâmica. Para a instrumentação, o PROSE utiliza BCEL[BCEL].

As operações de POA dinâmica são disponibilizadas pela classe `ch.ethz.prose.AspectManager`:

- `public void insert(Aspect ext)`
- `public void insert(Aspect ext, Object txId)`
- `public void withdraw(Aspect ext)`
- `public void withdraw(Aspect ext, Object txId)`
- `public void commit(Object txId)`
- `public void abort(Object txId)`

Os métodos `insert` adicionam um aspecto ao sistema enquanto que os métodos `withdraw`

3 Com a JVMDI, é possível executar uma máquina virtual Java em modo de depuração. A JVMDI é muito utilizada por depuradores e ferramentas que avaliam performance de sistemas escritos em Java.

removem. Se um identificador de transação é fornecido como parâmetro `txId`, a operação não será realizada imediatamente. Isso permite que toda uma seqüência de operações de POA dinâmica seja aplicada de uma única vez, o que é mais eficiente. Para isso, é preciso comitar a transação através do método `commit`. Quando ele é executado, todas as operações realizadas utilizando `txId` como identificador de transação são aplicadas. O argumento `txId` deve ser um objeto Java que possa ser utilizado como chave de um mapa (ou seja, que implementa corretamente os métodos `equals` e `hashCode`). Em contrapartida, o método `abort` cancela todas as operações associadas à transação `txId`, o que implica que elas nunca serão aplicadas.

Além desses métodos, aspectos podem ser removidos e adicionados em tempo de execução através de uma interface de linha de comando e de uma interface GUI.

O PROSE não fornece outros recursos além de *advices* e *pontcuts*, como "introdução de interfaces" e *mixins*.

5.POA Dinâmica no JBoss AOP: Uma Nova Abordagem

Como pudemos ver na seção anterior, a implementação de programação orientada a aspectos dinâmica no JBoss AOP apresenta algumas limitações.

Para habilitar a POA dinâmica, a instrumentação de código é aplicada a todos os pontos preparados de uma classe antes de ela ser carregada por um *class loader*. Essa instrumentação resulta na invocação dos interceptadores contidos em uma pilha, mesmo que essa esteja vazia durante a inicialização. Assim, quando o usuário adiciona um *bind* ou um interceptador a uma instância durante a execução, a adição dos interceptadores corretos às pilhas adequadas é tudo o que é necessário para o funcionamento da POA dinâmica. Mas existe um porém. E se o usuário quiser preparar o seu sistema inteiro para POA dinâmica? Talvez ele queira ser capaz de ativar o *log* em todas as classes em tempo de execução se um erro ocorrer no seu sistema. É uma boa idéia mudar os bytecodes, fazendo invocações a cadeias de interceptadores vazias em cada ponto de cada classe? Não se a performance for um fator de sucesso para o cliente.

A solução para isso seria manter o fluxo de execução do código intacto e trocar os *bytecodes* das classes necessárias se uma operação de POA dinâmica resultar na interceptação de pontos previamente não afetados. Com já vimos, a troca de *bytecodes* em tempo de execução é denominada *hot swap*. O *hot swap* foi incorporado à linguagem Java na versão 5, através da *Java Virtual Machine Tool Interface* (JVMTI) [JVMTI].

Para utilizar a JVMTI é necessário escrever um agente. Um agente é uma classe Java que deve implementar o método:

```
public static void premain(String agentArgs, java.lang.instrument.Instrumentation
inst);
```

Através desse método, o agente é inicializado e tem acesso às funcionalidades da JVMTI, fornecidas pela classe `java.lang.instrument.Instrumentation`. Atualmente o JBoss AOP possui um agente, que utiliza a JVMTI para a instrumentação das classes antes de serem carregadas por um *class loader*.

Apesar de possuir um agente, o JBoss AOP não utiliza a funcionalidade de *hot swap* fornecida pela JVMTI.

Como veremos, o *hot swap* em Java apresenta algumas restrições e, por isso, é preciso fazer alguma instrumentação em pontos preparados antes que as classes sejam carregadas. Todavia, essa instrumentação não altera o fluxo de controle nem a performance do sistema.

5.1. Preparação e Empacotamento

Atualmente, não é possível fazer mudanças arbitrárias nos *bytecodes* de uma classe com *hot swap*, pois há uma regra a ser cumprida: a assinatura de uma classe não pode ser alterada. Entretanto, a instrumentação de um ponto de um código acarreta na adição de métodos e de campos auxiliares a uma classe. Com isso, apesar de utilizarmos *hot swap* para a interceptação de um ponto do código em tempo de execução, é preciso que a assinatura de tal classe seja alterada antes de ela ser carregada.

Assim, temos que separar a transformação de classes em duas etapas:

- adição de métodos e campos necessários para a instrumentação de um ponto;
- substituição do ponto do código por um bloco que invoca uma pilha de interceptadores.

Por essa razão, a implementação da transformação no JBoss AOP teve que ser refatorada de modo a ser dividida em duas partes. Chamaremos a primeira parte, que altera a assinatura de uma classe, de "preparação"; e a segunda, de "empacotamento" (ou *wrapping*), pois o ponto será envolto por um bloco de código que invoca uma pilha de interceptadores e depois executa o código contido no ponto original.

Ao utilizar essa abordagem durante a transformação, sempre que um ponto do código precisar ser preparado, ele sofrerá a "preparação". E se existirem um ou mais interceptadores a serem aplicados a tal ponto, ele será "preparado" e "empacotado".

Isso resulta em mudanças nas classes responsáveis pela transformação de um ponto do código:

- `org.jboss.aop.instrument.FieldAccessTransformer`
- `org.jboss.aop.instrument.ConstructorExecutionTransformer`
- `org.jboss.aop.instrument.MethodExecutionTransformer`

Além dessas classes, existe a `org.jboss.aop.instrument.CallerTransformer`. Ela é responsável pela transformação de pontos de chamadas: invocação de um método feita por um outro método, de um método por um construtor, de um construtor por um método e de um construtor por um construtor. Esse tipo de transformação adiciona um campo a cada chamada interceptada e a preparação de pontos de chamada de uma classe resultaria na adição de um campo a cada invocação feita por métodos e construtores. Isso não é desejável, pois o número de campos adicionados se torna muito alto. Por esse motivo, os pontos de chamadas não são alvos da POA dinâmica no JBoss. Sendo assim, essa classe não teve que ser refatorada para a implementação da POA dinâmica através da JVMTI.

Naturalmente, a divisão do código de transformação em duas etapas não é o suficiente. É preciso distinguir pontos que serão interceptados dos que não o serão. Atualmente, o JBoss AOP

instrumenta ("prepara" e "empacota") todo ponto identificado por uma expressão *pointcut*. O algoritmo utilizado para determinar se um ponto deve ser instrumentado ou não é dado pelo seguinte pseudo-código:

```
for each pointcut
do
  if pointcut->matches(joinpoint)
    return WRAPPED
end
return NOT_INSTRUMENTED
```

É um algoritmo simples, que apenas verifica se um ponto é identificado por algum *pointcut* registrado no sistema. Se sim, o ponto é classificado como WRAPPED, que indica que ele tem que ser "empacotado".[4] Caso contrário, o joinpoint não será instrumentado.

Utilizando a nova abordagem de POA dinâmica, é preciso discernir os pontos do código que deverão ser preparados (chamaremos essa classificação de `PREPARED`) dos que deverão ser empacotados (`WRAPPED`). Para tanto, é preciso saber se o *pointcut* que identifica o ponto a ser instrumentado está associado a um *bind* ou não. Se ele estiver, então podemos concluir que ponto precisa ser empacotado. Se o ponto é apenas identificado por expressões *pointcuts* que não estão associadas a um *bind* (como é o caso das expressões "`prepare`"), sabemos que o ponto não deve ser "empacotado" e sim somente "preparado".

Todos os pontos que foram preparados antes de serem carregados por um *class loader* poderão ser empacotados em tempo de execução através da JVMTI, caso uma operação de POA dinâmica resulte na interceptação de tais pontos por um ou mais interceptadores.

Assim, os transformadores precisam de um novo algoritmo de classificação, que faça distinção entre "preparação" e "empacotamento":

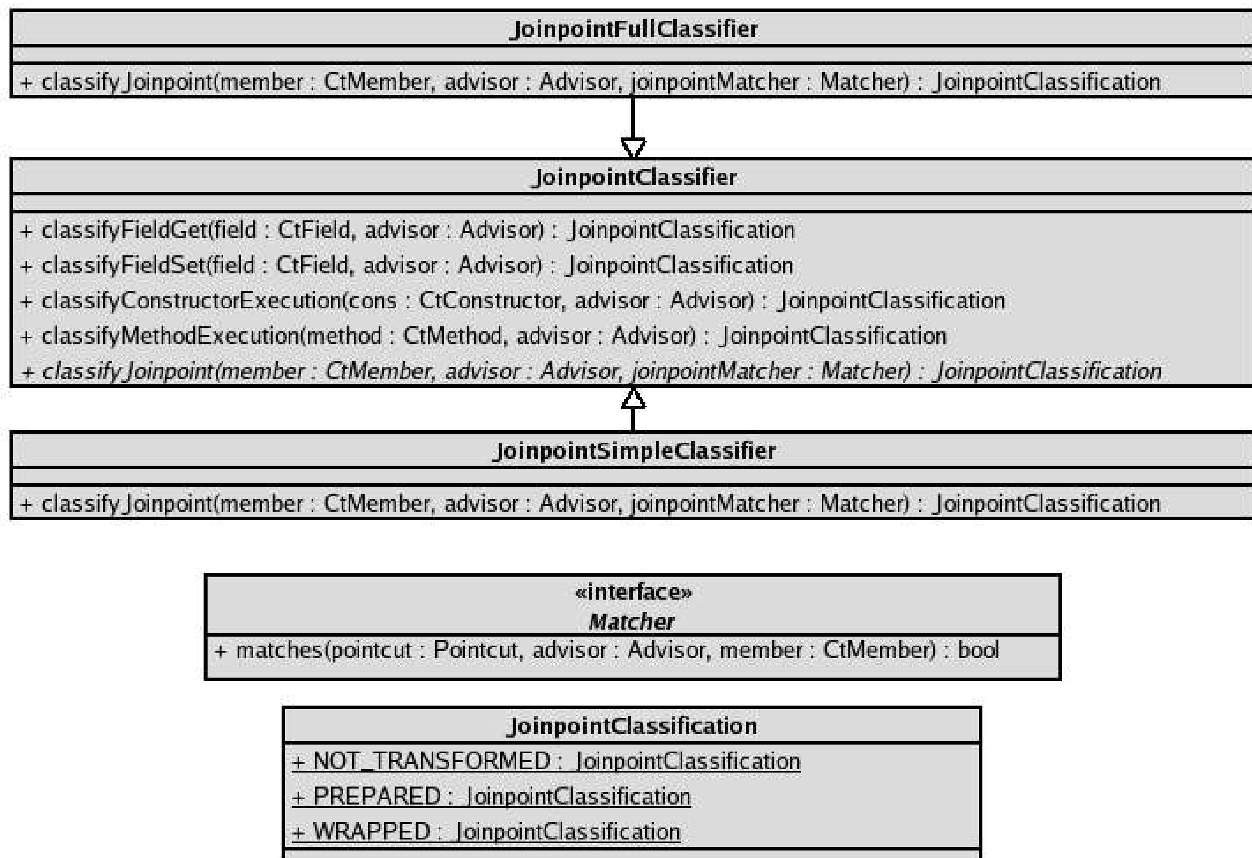
```
classification = NOT_INSTRUMENTED
for each pointcut
do
  // avoid useless matching
  if classification == PREPARED AND pointcut.bind == NULL
    continue;
  if pointcut->matches(joinpoint)
  do
    // no bind associated with pointcut: prepare the joinpoint only
    if pointcut.bind == NULL
      classification = PREPARE
    // one bind associated with pointcut means one or more interceptors
    // to be applied at this joinpoint: wrap the joinpoint
    else
    do
      classification = WRAPPED
      break;
    end
  end
end
return classification
```

4 Observe que para que um ponto seja "empacotado" é preciso que ele tenha sido previamente "preparado". Assim, fica subentendido que a classificação `WRAPPED` indica que um ponto terá que ser "preparado" e "empacotado".

A primeira verificação contida na iteração evita o custo desnecessário de verificar se o *pointcut* identifica o ponto quando esse já está classificado como "preparado" (`PREPARED`) e não existe um *bind* associado ao *pointcut*. Nesse caso, se o *pointcut* identificar o ponto, a classificação resultante seria `PREPARED`, o que torna a execução de `matches` inútil, porém custosa. O próximo comando "`if`" verifica se o *pointcut* identifica o ponto. Se sim, é preciso determinar se isso acarretará na execução de um ou mais interceptadores nesse ponto. Para isso, basta verificar se existe um *bind* associado ao *pointcut*. Se sim, a classificação é `WRAPPED` e a iteração termina. Caso contrário, a classificação é temporariamente `PREPARED`, até que um pointcut associado a um *bind* que identifique esse ponto seja encontrado (caso em que a classificação final será `WRAPPED`) ou que todos os *pointcuts* tenham sido verificados.

O algoritmo acima deve ser utilizado pelos transformadores a fim de determinar se um certo ponto deve ser somente "preparado" ou "preparado e empacotado".

Porém, o *hot swap* é uma funcionalidade que pode não estar disponível. Por exemplo, os usuários que utilizam uma máquina virtual de versão anterior ao 1.5 não podem utilizar a JVMTI. Sendo assim, a aplicação de *hot swap* para operações de POA dinâmica é limitada. Quando ela não estiver disponível, executar o algoritmo acima seria desnecessário, inclusive se considerarmos que ele é mais lento do que o algoritmo atualmente utilizado para a classificação de *joinpoints*. Sendo assim, queremos que o algoritmo de classificação seja dependente da abordagem para POA dinâmica que



está sendo utilizada. O algoritmo de classificação de pontos foi implementado como uma aplicação do padrão Strategy (315) [GOF] por esta razão.
A estratégia a ser utilizada durante a transformação é recebida pelo construtor de `org.jboss.aop.instrument.Instrumentor`, a fachada para o pacote de instrumentação. Essa classe é responsável por fornecer a funcionalidade da instrumentação para o JBoss AOP e delega tal tarefa aos transformadores, que estão contidos no pacote `org.jboss.aop.instrument`.
Uma conseqüência dessa solução é que os transformadores não precisam saber que só devem "preparar" um ponto do código se não houverem interceptadores a serem aplicados e se o *hot swap* estiver disponível. Eles utilizam a estratégia fornecida por `Instrumentor` para classificar cada ponto. Essa estratégia fará a classificação e dirá para os transformadores se o ponto deve ser "preparado" ou "empacotado". Portanto, os transformadores ficam responsáveis apenas por instrumentar pontos do código conforme sua classificação.

A arquitetura dessa solução é detalhada no diagrama anterior. A classe `org.jboss.aop.instrument.JoinpointClassifier` é a super classe da estratégia. 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. A classe `org.jboss.aop.instrument.JoinpointFullClassifier` implementa tal método com segundo algoritmo que vimos, que verifica se o *pointcut* tem um *bind* associado a ele. Já a classe `org.jboss.aop.instrument.JoinpoinSimpleClassifier`, implementa primeiro algoritmo que vimos, atualmente utilizado pelo JBoss AOP. Para possibilitar a implementação de um único algoritmo para todos os tipos de pontos (leituras e escritas de campos, execução de construtores e de métodos), a interface `org.jboss.aop.instrument.JoinpointClassifier.Matcher` é utilizada. A identificação de um ponto feita por uma expressão *pointcut* (processo de *matching*) é executada por um objeto do tipo `Matcher`. Esse é apenas uma referência para um método *matches* e 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 verifica pontos de leitura de campo. Essa instância é passada para o método `classifyJoinpoints`, que contém o algoritmo de classificação implementado pela subclasse e é independente do tipo de ponto a ser classificado.
Com essa estratégia, foi obtida uma forma efetiva de classificar pontos do código e utilizar o resultado de tal classificação para a execução da instrumentação de uma classe antes de ela ser carregada por um *class loader*. Além disso, o algoritmo a ser utilizado pode ser facilmente configurado.

5.2.POA Dinâmica e as Pilhas de Interceptadores

No item anterior, vimos como tratar pontos do código que não serão interceptados inicialmente, mas

que são identificados por uma expressão *pointcut*. Esses pontos são "preparados" pelos transformadores e podem passar a ser interceptados durante a execução do sistema através de um operação de POA dinâmica. Para que os interceptadores adicionados sejam chamados durante a execução do ponto a ser interceptado, não é o suficiente adicioná-los na pilha de interceptadores. É preciso que os *bytecodes* do ponto que passou a ser interceptado sejam trocados (*hot swapped*) por novos bytecodes, que "empacotam" a execução do ponto em um bloco de código. Esse bloco de código é responsável por executar a pilha de interceptadores e o ponto interceptado.

Desse modo, é necessário que as pilhas de interceptadores sejam observadas após a execução de cada operação de POA dinâmica. Todas as pilhas que deixarem de ser vazias resultarão no "empacotamento" do ponto do código ao qual devem ser aplicadas. Por outro lado, pilhas que passam a ser vazias também têm que ser observadas, para que o ponto que deixou de ser interceptado seja "desempacotado" (isto é, o bloco de código que antes executava a pilha de interceptadores e que envolvia o ponto interceptado tem que ser trocado por um bloco de código que apenas executa o ponto de código envolvido).

As pilhas de interceptadores a serem aplicadas em um ponto de código estão contidas em uma instância de `org.jboss.aop.Advisor` associada a uma classe instrumentada, que denominaremos *advisor*. Como já vimos, toda classe que é instrumentada passa a implementar a interface `org.jboss.aop.Advised`, através da qual é possível acessar o *advisor* e uma instância de `org.jboss.aop.InstanceAdvisor`, que chamaremos de *instance advisor*.

O *advisor* contém as pilhas de interceptadores que devem ser aplicadas a cada ponto de qualquer instância da classe instrumentada. Isso significa que temos um *advisor* para cada classe instrumentada. Por outro lado, o *instance advisor* contém os interceptadores que têm que ser invocados somente nos *joinpoints* da instância à qual ele está associado. Essa pilha de interceptadores pode ser manipulada através dos métodos de POA dinâmica fornecidos por *instance*

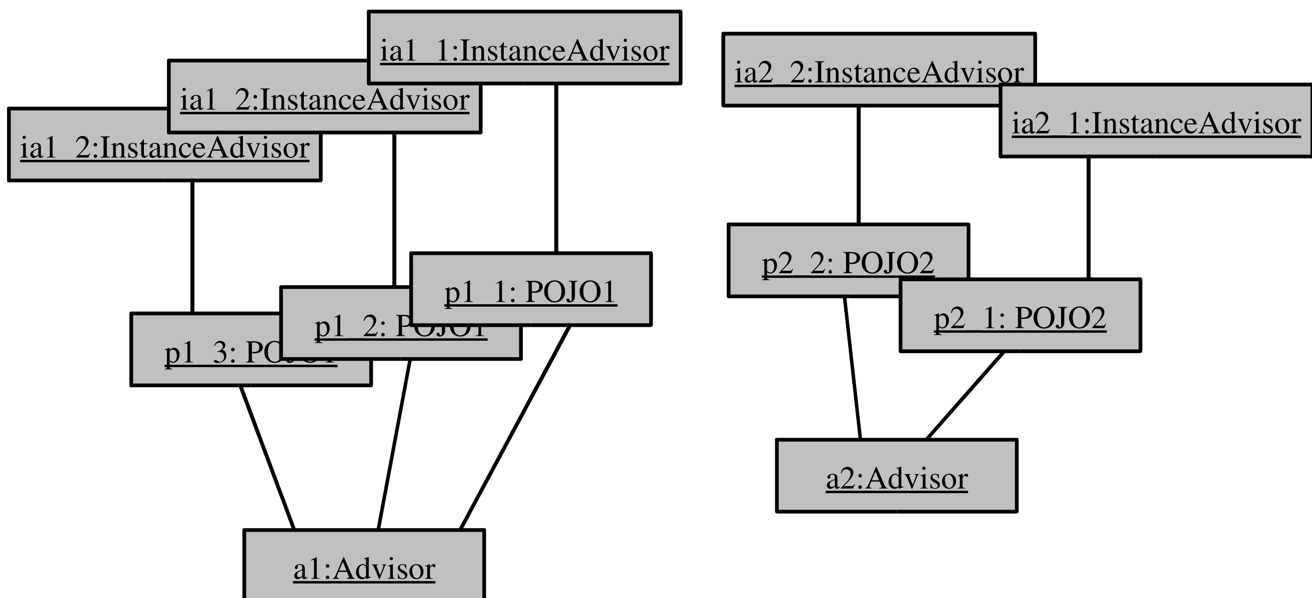


Fig 1- Diagrama de objetos detalhando como os advisors se relacionam com instâncias de POJO1 e de POJO2.

adivsor, listados anteriormente. O diagrama de objetos da Figura 1 mostra os *advisors* e *instance advisors* associados a três instâncias da classe `POJO1`, e duas de `POJO2`. Todas as instâncias de `POJO1` compartilham a mesmo *advisor*, porém possuem *instance advisors* distintos. O mesmo ocorre com as instâncias de `POJO2`. O *advisor* utilizado pelas instâncias de `POJO1` difere do associado às instâncias de `POJO2`.

As pilhas de interceptadores contidas no *advisor* e nos *instance advisors* de uma classe devem ser observadas durante a execução do sistema, para que os pontos sejam "empacotados" e "desempacotados" conforme a alteração no tamanho de suas pilhas de interceptadores. Para isso, foi adicionada a interface `org.jboss.aop.InterceptorChainObserver`, que implementa o padrão de arquitetura Observer(293) [GOF]. Um `InterceptorChainObserver` é responsável por observar as pilhas de uma classe, contidas no *advisor*. Além disso, ele é notificado de mudanças nas pilhas de cada instância daquela classe, contidas em *instance advisors*. Com esse mecanismo. é possível determinar quando o *hot swap* deve ser aplicado para o "empacotamento" e "desempacotamento" de pontos do sistema.

Na figura a seguir, é possível ver o diagrama da interface `InterceptorChainObserver`. Os primeiro método deve ser invocado pelo a*dvisor* para notificar qual o estado inicial das pilhas em cada ponto da classe. O método `interceptorChainsUpdated` deve ser invocado pelo advisor sempre que as pilhas forem alteradas. O métodos seguintes devem ser chamados pelos *instance advisors* sempre que uma de suas operações de POA dinâmica for invocada.

Fig 2- métodos disponiblizados pela interface org.jboss.aop.InterceptorChainObserver.

5.3.Hot Swap e a JVMTI

Com a arquitetura de observadores de cadeias de interceptadores implementada e a disponibilidade das operações de "preparação", "empacotamento" e "desempacotamento" através dos transformadores, é preciso mover para o próximo passo.

Os observadores devem notificar os transformadores sobre os pontos que passaram a ser interceptados, para que eles sejam "empacotados", e sobre os que deixaram de sê-lo, para serem "desempacotados". Para simplificar essa comunicação, um método foi adicionado à fachada `org.jboss.aop.instrument.Instrumentor`, `interceptorChainsUpdated`. Esse tem a função de receber a notificação dos observadores, na qual devem constar quais pontos tiveram o

estado da sua interceptação alterada. De posse dessa informação, a classe `org.jboss.aop.instrument.Instrumentor` delega o “empacotamento” e “desempacotamento” dos pontos necessários para os transformadores. Uma vez alterados os *bytecodes* desses pontos, é preciso executar o *hot swap* desses bytecodes para que as alterações sejam aplicadas na máquina virtual. É nesse instante que a JVMTI deve ser utilizada.

Entretanto, o código do JBoss AOP é dividido em duas partes: uma que é compatível com o Java 5, e outra que deve ser compatível com o Java 1.4. A classe `org.jboss.aop.instrument.Instrumentor` está contida no segundo grupo e, portanto, não pode acessar uma interface disponibilizada somente pelo Java 5. Uma solução possível para esse problema seria duplicar essa classe nos dois grupos do JBoss AOP. O `Instrumentor` que fosse adicionado à parte do Java 5 utilizaria a JVMTI. O outro, por sua vez, não precisaria disponibilizar a operação `interceptorChainsUpdated` para a notificação de mudanças de estado dos *joinpoints*, pois a implementação que estamos detalhando visa a utilizar a JVMTI para a execução do *hot swap* e, conseqüentemente, nunca poderá ser aplicada com o Java versã 1.4 ou anterior.

Mas existem fatores importantes a serem considerados. O primeiro deles é o quão indesejável é o código duplicado, por ser difícil de manter. O segundo, a impossibilidade de aplicar essa solução futuramente utilizando outros mecanismos de *hot swap*, compatíveis com o Java 1.4.

Por isso, uma alternativa à duplicação de `org.jboss.aop.instrument.Instrumentor` foi aplicada: a criação de uma interface para a execução do *hot swap*, que disponibiliza o serviço independente da sua implementação. Dessa forma, após a alteração dos *bytecodes* realizada pelos transformadores, a interface `org.jboss.aop.instrument.HotSwapper` é utilizada para o *hot swap* dos novos *bytecodes*. Uma implementação dessa interface deve ser provida sempre que o *hot swap* for utilizado. Assim, um adaptador da JVMTI a essa interface (aplicação do padrão Adapter (139) [GOF]), `org.jboss.aop.standalone.InstrumentationAdapter`, foi implementado e adicionado à parte do código fonte compatível com o Java 5. Caberá ao agente JVMTI do JBOSS AOP, que tem acesso às funcionalidades da JVMTI, fornecer uma implementação dessa interface.

O algoritmo utilizado pelo método `interceptorChainsUpdated` de `Instrumentor` é detalhado a seguir:

```
for each joinpointStatusUpdate
do
  fieldTransformer->wrap(joinpointStatusUpdate.newlyAdvisedJoinpoints.fieldReads)
  fieldTransformer->unwrap
     (joinpointStatusUpdate.newlyUnadvisedJoinpoints.fieldReads)
  fieldTransformer->wrap(joinpointStatusUpdate.newlyAdvisedJoinpoints.fieldWrites)
  fieldTransformer->unwrap
     (joinpointStatusUpdate.newlyUnadvisedJoinpoints.fieldWrites)
  constructorTransformer->wrap
     (joinpointStatusUpdate.newlyAdvisedJoinpoints.constructorExecutions)
  constructorTransformer->unwrap
     (joinpointStatusUpdate.newlyUnadvisedJoinpoints.constructorExecutions)
  methodTransformer->wrap
     (joinpointStatusUpdate.newlyAdvisedJoinpoints.methodExecutions)
  methodTransformer->unwrap
```

```
      (joinpointStatusUpdate.newlyUnadvisedJoinpoints.methodExecutions)
  if NOT joinpointStatusUpdate->isEmpty
      classes->add(joinpointStatusUpdate.clazz)
end
for each class in classes
do
  hotSwapper->registerChange(class, class.byteCodes)
end
hotSwapper->hotSwap
```

O algoritmo é simples: executa o "empacotamento" e o "desempacotamento" dos pontos necessários, e depois faz o *hot swap* dos *bytecodes* resultantes da transformação.

A arquitetura descrita nessa seção pode ser vista no diagrama abaixo:

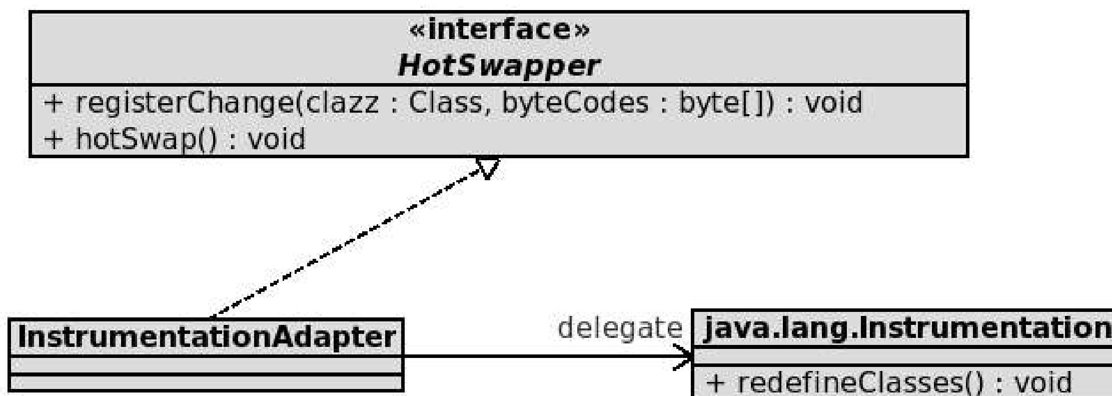


Fig 3 - Arquitetura utilizada para a execução de hot swap no JBoss AOP.

5.4.POA Dinâmica no JBoss AOP: Uma Estratégia

Todos os elementos necessários para o funcionamento da POA Dinâmica no JBoss AOP com *hot swap* foram expostos nos tópicos anteriores. Sabemos que precisamos: utilizar um novo algoritmo de classificação; de um observador de pilhas de interceptadores; de uma forma de notificar os transformadores quais pontos precisam ser "empacotados" e "desempacotados"; e de uma forma de fazer o *hot swap* após as alterações feitas pelos transformadores.

Porém, quando o *hot swap* estiver indisponível, não queremos sobrecarregar o sistema com um observador de pilhas de interceptadores, que faz comparação de pilha por pilha para descobrir quais passaram ou deixaram de ser vazias. Já vimos que também devemos utilizar o algoritimo de classificação mais simples quando não houver *hot swap*. Além disso, é preciso fornecer uma implementação da interface `org.jboss.aop.instrument.HotSwapper` se o HotSwap for utilizado.

Pode-se concluir a necessidade de comportamentos diferentes em vários pontos do sistema, relacionados entre si pela abordagem que está sendo utilizada para a implementação da POA dinâmica. Some a isso a necessidade de que a funcionalidade de *hot swap* seja altamente plugável, para que possa ser ativada e desativada facilmente.

Por esses motivos, foi utilizado o padrão Strategy(315)[GOF] para definir a estratégia para a implementação de POA dinâmica no JBoss AOP. A estratégia é representada pela interface `org.jboss.aop.DynamicAOPStrategy` e é responsável por prover:

- o algoritmo de classificação de *joinpoints* que deverá ser utilizado durante a transformação;
- uma instância de `org.jboss.aop.InterceptorChainObserver`, que poderá observar as alterações das pilhas de interceptadores ao longo da execução;
- e um método para ser notificada da finalização da execução de uma operação de POA dinâmica.

Esse último item merece um pouco de detalhamento. Naturalmente, sabemos que o observador de cadeias de interceptadores é notificado das mudanças ocorridas após uma operação de POA dinâmica. Então, de onde surge a necessidade de notificar a estratégia se o próprio observador já tem conhecimento de que uma operação de POA dinâmica foi executada? A questão é que existe um observador por classe. Eles não têm uma visão geral do sistema. Se fôssemos nos basear nos observadores para concluir que uma operação de POA dinâmica foi executada, cada observador que notou alteração nas pilhas dos interceptadores faria uma notificação a `org.jboss.aop.instrument.Instrumentor`. Isso resultaria em, possivelmente, mais de uma notificação por operação de POA dinâmica. Isso é desnecessariamente custoso. Sendo assim, somente quando a estratégia é notificada da ocorrência de uma operação de POA dinâmica os dados de todos os observadores são reunidos e enviados ao método `interceptorChainsUpdated` de `org.jboss.aop.instrument.Instrumentor`.

Existem duas implementações dessa estratégia. A primeira é a `HotSwapStrategy`. Ela apenas implementa o comportamento visto nos tópicos anteriores:

- fornece o algoritmo `org.jboss.aop.instrument.JoinpointFullClassifier` para ser utilizado pelos transformadores;
- fornece observadores que armazenam as informações de quais pontos do código passaram ou deixaram de ser interceptados 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 `interceptorChainsUpdated` de `org.jboss.aop.instrument.Instrumentor`. Juntamente com esses dados, essa estratégia envia para esse método uma implementação de `org.boss.aop.instrument.HotSwapper`.

A segunda implementação é a classe `org.jboss.aop.LoadInterceptedClassesStrategy`. Essa classe apenas implementa a abordagem já existente à POA dinâmica: classes são totalmente instrumentadas antes de serem carregadas, e a simples atualização das pilhas dos interceptadores é suficiente para que as operações de POA dinâmica sejam efetivadas. Essa estratégia:

- fornece `org.jboss.aop.instrument.JoinpointSimpleClassifier` como algoritmo de classificação de joinpoints;
- fornece um observador nulo para as cadeias de interceptadores, uma vez que não é preciso observar pilhas de interceptadores nessa abordagem;
- ignora as notificações de ocorrências de operações de POA dinâmica, pois não é preciso tomar nenhuma ação quando elas ocorrem.

A arquitetura completa dessa solução pode ser vista a seguir:

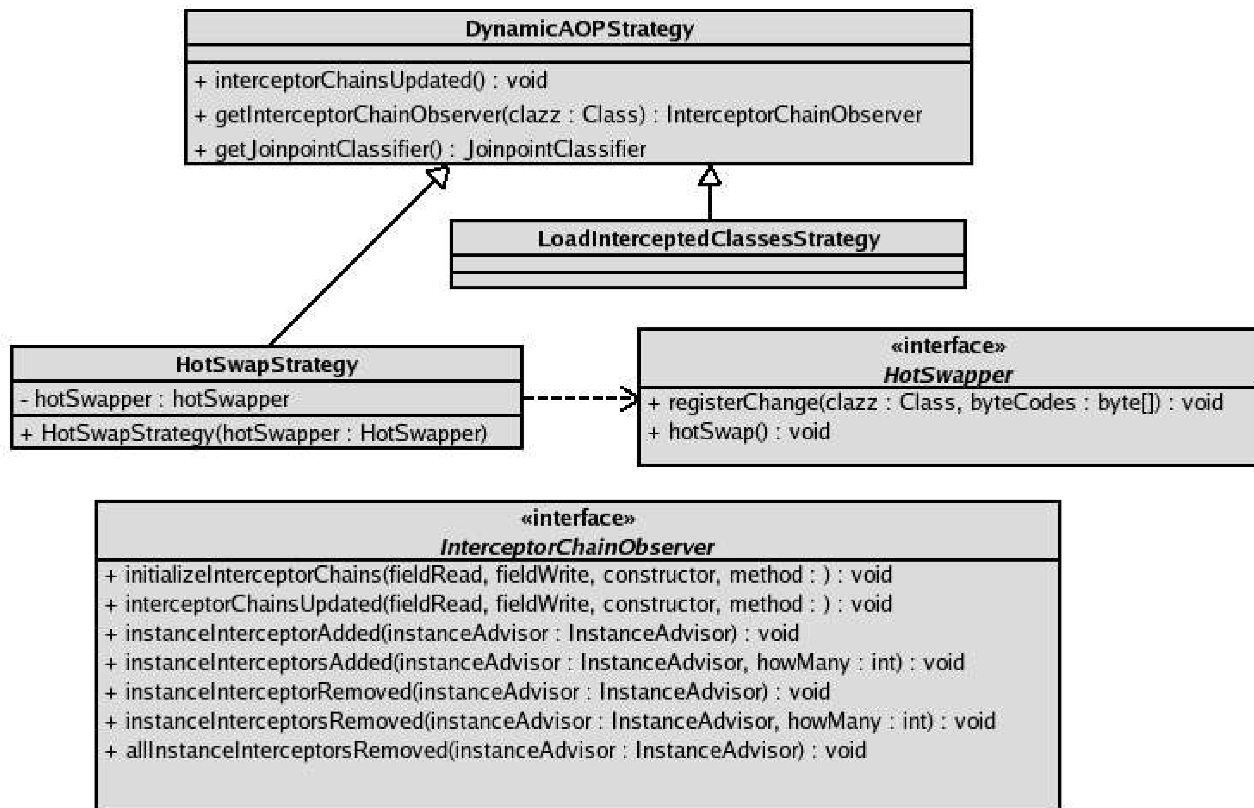


Fig 4- Estratégia de POA Dinâmica

5.5.Conseqüências

Uma nova abordagem a POA dinâmica no JBoss AOP foi desenvolvida. Ela foi implementada de forma a ser uma funcionalidade plugável no sistema, o que garante um mecanismo simples e eficaz para habilitar ou desabilitar seu uso.

Além disso, para utilizar a solução apresentada com um novo método de *hot swap* que não a JVMTI, basta implementar a interface `org.jboss.aop.instrument.HotSwapper`. Disso, concluímos que essa solução pode ser facilmente estendida para o uso com outras técnicas de *hot swap*.

6.Conclusão do Estudo

Foram vistas diferentes abordagens à POA dinâmica através dos arcabouços JBoss AOP, AspectWerkz e PROSE. Apesar das divergências, todas essas ferramentas lançam mão da alteração do fluxo de execução das classes do usuário para a aplicação de aspectos. Enquanto que o JBoss AOP aplica tais alterações de forma estática, antes das classes serem carregadas, as outras duas ferramentas fornecem a possibilidade da instrumentação dinâmica através do uso de *hot swap*. A instrumentação estática apresenta a vantagem do baixo custo de operações de POA dinâmica, sob a

penalidade de afetar o fluxo de controle do sistema mesmo na ausência de aspectos.
Em contrapartida, o AspectWerkz fornece os modos de instrumentação dinâmico e estático. Porém, a programação orientada a aspectos dinâmica é disponibilizada somente no modo dinâmico, sendo menos flexível que o JBoss AOP. Apesar disso, o AspectWerkz fornece a vantagem do *hot swap*, garantindo uma melhor eficiência que o JBoss AOP quando não existem aspectos a serem aplicados.
Por último, a única ferramenta totalmente dinâmica vista é o PROSE. Essa é também a única ferramenta que fornece uma alternativa à instrumentação de bytecodes, a adição de *breakpoints* em uma máquina virtual no modo de depuração. Todavia, esse modo é muito lento em comparação com os demais.
Após esse estudo, foi implementada uma nova solução de POA dinâmica no JBoss AOP utilizando-se a JVMTI para a troca de bytecodes em tempo de execução. O *hot swap* foi adicionado como uma funcionalidade plugável no sistema e também apresenta vantagens e desvantagens se comparado com a instrumentação estática. O uso de *hot swap* mantém o fluxo de controle do sistema inalterado antes da adição de aspectos com operações de POA dinâmica. Por outro lado, tais operações se tornam mais custosas porque exigem maior processamento, decorrente da análise das alterações ocorridas nas pilhas de interceptadores e do custo adicional do *hot swap*.

7. Referências

[AJProj]: AspectJ Project, http://www.eclipse.org/aspectj/
[JBAOP]: JBoss AOP Project, http://www.jboss.org/products/aop
[AWZ]: AspectWerkz Project, http://aspectwerkz.codehaus.org
[PROSE]: PROSE Project, prose.ethz.ch
[JAnnot]: Java Annotations Guide,
http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html
[XDCLT]: XDoclet Project, http://xdoclet.sourceforge.net
[JBOSS]: JBoss, http://www.jboss.org
[JBAS]: JBoss AS Project, http://www.jboss.com/products/jbossas
[GOF]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, 1994
[JVMDI]: Java Virtual Machine Debug Interface Reference,
http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html
[JkIBM]: Jikes Project, http://jikes.sourceforge.net
[BCEL]: BCEL Project, http://jakarta.apache.org/bcel
[JVMTI]: Java Virtual Machine Tool Interface, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/