2
Conceitos Básicos

Este capítulo tem por objetivo definir os conceitos básicos a serem utilizados ao longo desta dissertação, sendo eles: conceitos essenciais do tratamento de exceção de mecanismos e da programação orientada a aspectos (POA), bem como as características da linguagem JBoss AOP. Esta seção também busca detalhar o mecanismo de tratamento de exceção na linguagem JBoss AOP que é implementado. Além disto, apresentamos uma visão geral sobre a plataforma JEE (Java Enterprise Edition), o servidor de aplicações JBoss AS, os conceitos associados ao framework EJB e, por fim, fazemos uma breve introdução sobre a ferramenta SAFE, que é a base para esta dissertação.

2.1. JBoss AOP

No desenvolvimento de aplicações, frequentemente existem interesses que podem estar espalhados em vários componentes da uma mesma aplicação, tais como: (i) segurança, (ii) persistência e (iii) distribuição. Esse tipo de interesse é chamado de interesse transversal e implementá-lo, em linguagens tradicionais, leva ao espalhamento do código relativo ao interesse pelos componentes afetados por este (TIRELO, et alii, 2004).

A Programação Orientada a Aspectos (POA) (KICZALES, 1996; KICZALES et alii, 1997) é um paradigma que foi proposto com o objetivo de melhorar a separação de responsabilidades na codificação de interesses transversais que propõe: (i) uma nova abstração, chamada Aspecto, para codificar tais interesses que não podem ser facilmente expressos pelos elementos de abordagens de decomposição tradicionais, (e.g., classes, funções); e (ii) um processo para combinar o Aspecto com o código da aplicação, chamado weaving.

Esse paradigma tem sido cada vez mais utilizado com o intuito de modularizar interesses transversais, tais como a persistência (RASHID e CHITCHYAN, 2003; SOARES et alii, 2006), a distribuição (SOARES et alii, 2006), e padrões de design (HANNEMANN e KICZALES, 2002; GARCIA et

alii, 2005). Assim, foi observada, empiricamente, que a decomposição dos interesses transversais em aspectos promove a modularidade (GARCIA et alii, 2005) e a estabilidade do design (GREENWOOD et alii, 2007), além de se observar também que aspectos podem ser usados para modularizar o tratamento de exceções em algumas situações (FILHO et alii, 2007).

O weaving, nos frameworks orientados a aspectos, pode ocorrer em tempo de compilação, tempo de carregamento ou tempo de execução. As principais vantagens do weaving em tempo de compilação são as de que ele evita sobrecarga desnecessária para modificar a aplicação em tempo de execução e que, além disso, são realizadas checagens estáticas durante este processo, o que pode expor muitos erros de composição. Por outro lado, esta forma de weaving exige que todos os códigos afetados pelo aspecto (e o código referenciado por este) estejam presentes no momento da compilação.

Quando o weaving é realizado em tempo de carregamento da aplicação, a implementação (i.e., framework ou linguagem) POA lê um arquivo de configuração que especifica quais são os aspectos a serem combinados com o código original da aplicação quando a mesma é carregada. Por fim, quando o weaving ocorre em tempo de execução é permitida a adição e remoção de aspectos nesse tempo e, normalmente, isto é feito através da definição de proxies (GOF, 1995) ou instrumentação de código em cada elemento que será afetado por um aspecto.

Apesar de haver, atualmente, algumas linguagens e frameworks que implementam este paradigma (como, por exemplo, AspectJ, JBoss AOP, Spring AOP, JAC e CaesarJ), esta dissertação, por conveniência, só abordará o JBoss AOP. Essa linguagem, nas suas primeiras versões, foi um framework OA que vinha embarcado no servidor de aplicações da JBoss, o chamado JBoss AS. No entanto, o framework JBoss AOP utiliza a linguagem Java e arquivos XML de forma a permitir a construção de estruturas de programas que utilizem classes e aspectos para implementar interesses transversais.

A abstração de aspecto no JBoss AOP é composta por pointcuts, advices, binds, inner-types, métodos e atributos internos. Join points são localizações bem definidas na estrutura de um programa (i.e. chamadas de métodos, declarações de métodos, dentre outras.); pointcuts são expressões que identificam um ou mais joinpoints no código da aplicação; advices são métodos que contém o

comportamento adicionado ao joinpoint quando a expressão do pointcut o identifica; bind indica quais serão os advices invocados quando um pointcut identifica um ou mais joinpoints; e, por fim, inner-types são métodos, atributos e classes adicionados ao código original da aplicação para que os advices sejam invocados.

A seguir serão apresentados mais detalhadamente os componentes de um aspecto em JBoss AOP, bem como a linguagem dos pointcuts e como funciona o processo de composição entre o código de uma aplicação e os aspectos.

2.1.1. Aspecto

No JBoss AOP a forma com que os advices interceptam o código da aplicação pode ser definida tanto através de anotações numa classe Java, quanto através de arquivos XML, separando, desta forma, a definição dos pointcuts da dos códigos dos advices. Na Figura 1 o aspecto é definido através de anotações no código Java. Neste exemplo, o aspecto intercepta todas as chamadas para métodos públicos que sejam `void`.

```
1. @Aspect (scope = Scope.PER_VM)
2. public class PerformanceAspect {
3.   @PointcutDef ("call(public void * -> * (..))")
4.   public static Pointcut publicVoidMethods;
5.   @Bind (pointcut="PerformanceAspect.publicVoidMethods"
   type=AdviceType.AROUND)
6.   public Object performance(Invocation invocation)
7.   {
8.     long start = System.currentTimeMilis();
9.     try {
10.        Object ret= invocation.invokeNext();
11.      } finally {
12.        long end= System.currentTimeMilis();
13.        System.out.println("executed in:" +(end-start))
14.      }
15.    }
16.  }
```

Figura 1 – Exemplo de código de um aspecto utilizando JBoss AOP.

Todavia, pode-se definir o mesmo aspecto utilizando XML como pode ser visto na Figura 2 abaixo.

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
     <aspect class="PerformanceAspect"/>
     <bind pointcut="call(public void * -> * (..))">
        <around aspect=" PerformanceAspect" name="performance"/>
     </bind>
</aop>
```

Figura 2 – Exemplo de definição de aspectos usando XML.

Em algumas situações é necessário definir se um aspecto interceptará ou não um método utilizando outras fontes de informação que vão além dos arquivos de configuração próprios do JBoss AOP. Um exemplo disso é o gerenciamento de transações do framework EJB no servidor de aplicação JBoss AS (ver Seção 2.3.1). O gerenciamento de transações define políticas que determinam o início e fim de cada transação, sendo que, para cada uma dessas políticas, há um aspecto específico de implementação e a definição de qual política será utilizada encontra-se no descritor de implantação (deployment descriptor) do EJB (ver Seção 2.3.1).

Para permitir o acesso a essas informações, o JBoss AOP possui, por sua vez, o conceito de fábrica de aspectos, permitindo, assim, que o aspecto interceptador do método seja definido através de uma fábrica, ou seja, da utilização do padrão Factory Method (GOF, 1995) para indicar qual aspecto interceptará o joinpoint quando este for executado. O exemplo abaixo, ilustra a implementação de uma fábrica de aspectos que implementa a interface `org.jboss.aop.advice.AspectFactory`, e que define o método da fábrica chamado `createPerJoinpoint`.

```
1. public class ExampleFactory implements
   org.jboss.aop.advice.AspectFactory
2. {
3.    public void initialize()
4.    {
5.    }
6.    public Object createPerJoinpoint(Advisor advisor,
   Joinpoint jp)
7.    {
8.      return new ExampleInterceptor(advisor,jp);
9.    }
10.     }
```

Figura 3 – Implementação de uma fábrica de aspectos.

Outra peculiaridade do JBoss AOP, no que tange a definição de aspectos, é a presença de domínios (Domain). Um domínio tem por finalidade definir um conjunto de aspectos e pointcuts. A definição deste conjunto é útil no momento do weaving, pois o desenvolvedor pode optar que apenas aspectos definidos em um dado domínio sejam combinados com uma ou mais classes da aplicação. Outra propriedade do domínio é a extensão, ou seja, um domínio pode herdar os aspectos e pointcuts de outro domínio, como pode ser visto no exemplo abaixo.

```
<aop>
<interceptor factory="InteceptorExample1" />
<interceptor factory="InteceptorExample2" />
<interceptor factory="InteceptorExample3" />

<domain name="domain1">
      <bind pointcut="execution(public * *->*(..))">
         <interceptor-ref name="InteceptorExample1"/>
      </bind>
   </domain>
 <domain name="domain2">
      <bind pointcut="execution(public * *->*(..))">
         <interceptor-ref name="InteceptorExample2"/>
      </bind>
   </domain>
<domain name="ExtendedDomain" extends=”domain1”>
      <bind pointcut="execution(public * *->*(..))">
         <interceptor-ref name="InteceptorExample3"/>
      </bind>
   </domain>
</aop>
```

Figura 4 – Exemplo da configuração de domínios no JBoss AOP.

Na Figura 4 acima são definidos três domínios, sendo eles (i) `domain1`, (ii) `domain2` e (iii) `ExtendedDomain`. O primeiro define que todos os métodos públicos serão interceptados pelo `InteceptorExample1`; o segundo, que todos os métodos públicos serão interceptados pelo `InteceptorExample2`; e o terceiro, que é herdado de `domain1`, define que todos os métodos públicos serão interceptados pelos `InteceptorExample1` e `InteceptorExample3`.

Posto isto, se efetuarmos o processo de weaving usando `ExtendedDomain` como domínio da classe `A` da Figura 5 abaixo, teremos que o método `m` de `A` será interceptado pelos advices `InteceptorExample1` e `InteceptorExample3`. Em contrapartida, se aplicamos o mesmo processo usando o `domain2` na classe `B`, então o método `n` da classe `B` será interceptado pelo advice `InteceptorExample2`, permitindo, desta forma, a separação das definições de aspectos para a classe `A` e `B`.

```
1. public class A{
2.   public void m(){
3.     B b= new B();
4.     b.n();
5.   }
6. public static void main(String[] args){
7.   A a = new A();
8.   a.m();
9. }
10. }
11. public class B{
12.   public void n(){
13.   }
14. }
```

Figura 5 – Exemplos de uma aplicação afetada por dois domínios diferentes.

Assim, o conceito de domínio apresentado acima é utilizado na implementação do framework EJB no servidor de aplicação JBoss AS para definir os aspectos que interceptarão cada tipo de bean (ver Seção 2.3.1).

Além dos conceitos citados acima, o JBoss AOP permite que informações (objetos) sejam armazenadas numa área específica, denominada contexto de excecução. Esta área é criada a cada execução de aspectos e é utilizada para passar parâmetros para eles.

2.1.2.Pointcut

Pointcut é uma expressão da linguagem do JBoss AOP que pode identificar um ou mais joinpoints, sendo possível, assim, criar expressões que delimitem o subconjunto da aplicação que se deseja interceptar (JBOSS AOP, 2009). Segundo o pointcut, a linguagem é flexível e contém expressões para composição, funções e coringas e os operadores para composição de expressões são os apresentados abaixo:

- !<expressão> indica negação da expressão

• <expressão> AND <expressão> indica que o valor final da expressão será aplicado o operador AND ao resultado das avaliações das expressões

• <expressão> OR <expressão> indica que o valor final da expressão será aplicado o operador OR ao resultado das avaliações das expressões

Na Tabela 1 serão apresentadas funções da linguagem, em que <método> indica uma expressão que identifica um método; <construtor> indica uma expressão que identifica um construtor; e <atributo>, uma que identifica um atributo.

Tabela 1 – Resumo Funcões da Linguagem.

Expressão	Descrição
*	Indica que para haver o casamento da expressão com a frase, a frase pode ter 0 ou mais caracteres.
..	Indica que o método pode ter 0 ou mais parâmetros.
@	Indica que a expressão seguinte é uma anotação.
$instanceof{}	Pode ser usado para substituir o nome da classe. Desta forma é possível realizar o casamento para subtipos de classes.
$typedef{id}	Utiliza expressões complexas que foram previamente definidas.
public, static, private	Indica modificadores de classes/métodos.
new()	O casamento ocorre na chamada do construtor.
execution(<método> ou <construtor>)	O casamento ocorre dentro do método que esta sendo interceptado.
construction(<construtor>)	O casamento ocorre dentro do construtor do objeto
get (<atributo>)	O casamento ocorre quando o atributo

	identificado pela expressão entre parenteses é acessado para leitura.
set(<atributo>)	O casamento ocorre quando o atributo identificado pela expressão entre parenteses é acessado para escrita.
field(<atributo>)	O casamento ocorre quando o atributo identificado pela expressão entre parênteses é acessado para leitura ou escrita.
all(<método> ou <construtor> ou <atributo>)	O casamento ocorre para qualquer tipo, seja atributo, método ou construtor.
call(<método> ou <construtor>)	O casamento ocorre na chamada do método ou construtor. Esta expressão é utilizada quando não há possibilidade de interceptar o método, por exemplo, a classe `System`.
within(<método> ou <construtor>)	O casamento ocorre dentro de qualquer código que tenha um tipo em particular.
withincode(<método> ou <construtor>)	O casamento ocorre dentro de um método ou construtor.
has(method or constructor)	Adiciona mais uma restrição ao casamento. A classe deve ter o método ou construtor que case com a expressão que foi definida.
hasfield(<atributo>)	Adiciona mais uma restrição ao casamento. A classe deve ter o atributo que case com a expressão que foi definida.

2.1.3.Advices

Advices são trechos de código que podem ser vinculados a joinpoints e, no caso do JBoss AOP, são métodos de uma classe, havendo seis tipos deles listados e detalhados a seguir: (i) before é executado antes do joinpoint; (ii) after é executado depois da execução do joinpoint, se não houver exceção; (iii) around

encapsula a execução do joinpoint; (iv) after-throwing é executado se na execução do joinpoint ocorrer o lançamento de uma exceção; (v) finally é executado após a execução do joinpoint independentemente deste lançar ou não uma exceção; (vi) interceptor é semelhante ao advice around e encapsula a chamada ao joinpoint. Como um advice é representado por um método de uma classe, então uma classe pode ter vários advices. No entanto, há uma restrição no framework que limita ao ponto de que cada classe possa ter apenas um advice do tipo interceptor, pois para que um advice deste tipo possa ser definido, a classe deve implementar a interface `org.jboss.aop.advice.Interceptor` e definir o método `invoke` que representará o adivce do tipo interceptor.

O advice pode receber os parâmetros da execução do joinpoint e, para tanto, é necessário que os parâmetros do método sejam anotados de tal forma a indicar quais os parâmetros da execução serão disponibilizados em quais parâmetros do advice (ver Tabela 2).

Abaixo segue uma lista de anotações:

Tabela 2 – Anotações utilizadas para associar valores da execução a parâmetros no método do advice.

Anotação	Função
`@JoinPoint`	Indica qual parâmetro irá receber o objeto que representa o joinpoint;
`@Target`	Indica qual parâmetro irá receber o objeto no qual está localizado o joinpoint ;
`@Caller`	Indica qual parâmetro irá receber o objeto que realizou a chamada para o joinpoint ;
`@Thrown`	Indica que parâmetro irá receber a exceção, quando for o caso;
`@Return`	Indica que parâmetro irá receber o retorno da invocação;
`@Arg`	Indica qual parâmetro irá receber que parâmetro da invocação;
`@Args`	Indica qual parâmetro irá receber todos os parâmetros da invocação.

No exemplo abaixo (Figura 6), foi definido um aspecto que intercepta todas as chamadas de método que retornem um objeto do tipo `java.lang.Object` e, ao final de sua execução, o advice imprime o objeto retornado ou a mensagem da exceção ocorrida dentro do método interceptado ou algum método chamado pelo mesmo. Como pode ser observado na Figura 6, o advice `finallyCall` recebe dois parâmetros sendo eles (i) o objeto retornado pela invocação do método interceptado, representado pelo parâmetro `returnedValue` e anotado como `@Return`; e (ii) a exceção ocorrida, caso alguma exceção seja levantada pela execução do método interceptado, representado pelo parâmetro `thrownEx` e tendo como anotação `@Thrown`. Desta forma, o advice detecta se houve ou não um erro na execução do método e imprime a mensagem adequada.

```
1. @Aspect (scope = Scope.PER_VM)
2. public class FinallyAspect {
3.   @PointcutDef ("call(public Object * -> * (..))")
4.   public static Pointcut publicObjectMethods;
5.   @Bind (pointcut="FinallyAspect.publicObjectMethods"
   type=AdviceType.FINALLY)
6.   public void finallyCall(@Return Object returnedValue,
   @Thrown Throwable thrownEx){
7.     if (thrownEx !=null){
8.      System.out.println("Error:"+ thrownEx.getMessage();
9.     } else if (returnedValue != null){
10.    System.out.println("Returned Value"+ returnedValue);
11.   }
12.   }}
```

Figura 6 – Exemplo de advice recebendo o contexto de execução.

2.1.4.Binding

O binding tem por função indicar quais advices estão associados a quais pointcuts. Além disso, pode especificar que (i) os advices só irão ser executados caso ocorra um fluxo específico, e esse é definido através de designador cflow; e que (ii) a decisão de executar ou não os advices pode ser feita em tempo de execução, através de um método definido pelo usuário.

Quando o binding é feito usando XML, naturalmente se define a precedência dos advices pela ordem de chamada dos mesmos. Em contrapartida, se a definição dos aspectos for feita através das anotações, faz-se necessária a formalização da precedência do Aspecto, realizada a partir da anotação `@Precedence`.

2.1.5.Introductions / Mixing

Outro tipo de construção que normalmente é implementado pelos frameworks OA é a modificação da estrutura estática das classes interceptadas, isto é, hierarquia de classes e anotações. No JBoss AOP ele permite adicionar interfaces ou anotações a classes, esta operação é chamada de Introduction; e uma vez que a interface foi adicionada é possível definir sua implementação, onde, esta operação é chamada de Mixing.

Na Figura 7 abaixo foi criado um aspecto que adiciona a interface `java.io.Externalizable` a todas as classes que possuírem um método anotado com `@externalize`, além disso, define que a implementação da interface será a classe `Externalizer`.

```
1. <aop>
2.  <introduction expr="has(* *->@externalize(..))">
3.   <mixin>
4.     <interfaces>java.io.Externalizable</interfaces>
5.     <class>Externalizer</class>
6.     <construction>new Externalizer(this)</construction>
7.   </mixin>
8.  </introduction>
9. </aop>
```

Figura 7 - Exemplo de adição de interfaces a classes interceptadas.

2.1.6.Weaving

Weaving é a denominação atribuída a um mecanismo que realiza a composição entre o código original da aplicação e os códigos relativos aos aspectos. Uma característica desse mecanismo do JBoss AOP é permitir que os aspetos sejam adicionados à aplicação tanto em tempo de carregamento quanto em tempo de compilação da aplicação. No entanto, apesar de o processo de adição de aspectos em tempo de carregamento ser semelhante ao processo de compilação dos aspectos, há uma diferença entre eles que diz respeito à realização. Isso se deve ao fato de que no processo de carregamento da aplicação, a introdução dos códigos nas classes é realizada na memória ao invés de modificar os arquivos binários.

Tanto em tempo de compilação quanto em tempo de carregamento, o JBoss AOP funciona de forma semelhante ao padrão Observer (GOF, 1995), no qual um ou mais observadores (Observer) observam um assunto (Subject). Uma vez estabelecido esse paralelo, o assunto em questão relatado no padrão Observer é o joinpoint definido pelo pointcut e os observadores são os advices. Assim, o weaver do JBoss AOP modifica o código do joinpoint (em tempo de compilação ou carregamento) acrescentando somente códigos relativos à chamada dos observadores e adicionando advices como observadores do joinpoint (e isto corre somente durante o processo de carregamento da aplicação); permitindo, desta forma, que advices sejam adicionados e removidos em tempo de execução.

Com a finalidade de instrumentar as classes interceptadas, o JBoss possui basicamente duas estratégias de instrumentação que são (i) a Classic Weaving e (ii) a Generated Advisor Weaving, discutidas a seguir.

2.1.6.1.Classic Weaving

Esta estratégia de instrumentação foi a primeira a ser utilizada e pode ser descrita a partir dos modos otimizado e não-otimizado. Imaginemos, por exemplo, o fluxo `m1` → `ad1` → `m2` , em que `m1` e `m2` são métodos de classes e `ad1` é um advice que intercepta o método `m2`. No modo não-otimizado, a invocação do joinpoint, que foi identificada pelo pointcut, é realizada utilizando reflexão, ou seja, a chamada de `ad1` para m2 é feita através de reflexão. Entretanto, em contrapartida, o modo otimizado não utiliza reflexão, mas gera código adicional para realizar a mesma atividade. Se por um lado, a performance é melhor, pois se utiliza do modo otimizado, por outro, mais códigos são adicionados à classe instrumentada.

Para exemplificar o processo de transformação que ocorre quando uma classe é interceptada por um aspecto, as listagens abaixo mostram um exemplo de classe antes de sofrer tal interceptação (ver Figura 8), o advice (ver Figura 9) e a configuração do aspecto (ver Figura 10), bem como o resultado da instrumentação feita pelo JBoss AOP na classe – esse obtido através da decompilação do código resultante utilizando a ferramenta Jad (ver Anexo 8.5) (JAD, 2000).

```
1. public class Pojo {
2.    public void m() {
3.          System.out.println("method m invoked");
4.    }
5.    public static void main(String[] args) {
6.          (new Pojo ()).m();
7.    }
8. }
```

Figura 8 - Exemplo de uma classe simples.

A Figura 9 abaixo apresenta um exemplo de advice para gravar informações quando o método interceptado for executado com sucesso.

```
1. public class AfterAspect {
2.    public void logAdvice(){
3.      System.out.println("log after...:");
4.     }
5. }
```

Figura 9 - Exemplo de um advice.

A configuração da Figura 10 define que o aspecto intercepta todas as chamadas não estáticas de classes que iniciem com a palavra “Pojo”, adicionando, assim, o adivice do tipo After chamado `logAdvisor` da classe `AfterAspect`(ver Figura 9).

```
1. <aop>
2.   <aspect class=" AfterAspect"/>
3.   <bind pointcut="call(!static * *.Pojo*-> * (..))">
4.      <after aspect="AfterAspect" name="logAdvice"/>
5.   </bind>
6. </aop>
```

Figura 10 - Exemplo da configuração do JBoss AOP para que o aspecto da Figura 7 intercepte os métodos da classe da Figura 6.

A instrumentação de uma classe interceptada ocorre de acordo com o descrito a seguir: (i) a classe interceptada passa a implementar a interface

`org.jboss.aop.Advised` bem como todos os métodos que compõem a mesma (linhas 1 e 39 a 54 do Anexo 8.5); (ii) são adicionados um atributo estático `ClassAdvisor aop$classAdvisor$aop`, e outro atributo de instância `ClassInstanceAdvisor _instanceAdvisor` que são o gerenciador de advices da classe e da instância, respectivamente (linhas 56 e 57 do Anexo 8.5); (iii) é adicionada uma `java.lag.ref.WeakReference` para cada joinpoint interceptado (linha 58 do Anexo 8.5) onde são guardadas as referências para informações a respeito do joinpoint ; (iv) é adicionado um bloco estático que tem por objetivo inicializar os aspectos da classe quando esta for carregada pela máquina virtual Java (linhas 59 a 65 do Anexo 8.5); (v) quando a instrumentação é otimizada, é definida uma classe Invocation Bean para cada joinpoint identificado por um pointcut (linhas 3 a 18 do Anexo 8.5); e, por fim, (vi) são substituídos os joinpoints identificados por pointcuts a partir de chamadas ao método `invokeNext` dos respectivos Invocation Beans, não havendo, então, qualquer referência ao tipo de advice a ser executado.

Invocation Beans são classes que representam a invocação do joinpoint, a partir delas é que os advices que interceptam o joinpoint são invocados. Inovation beans implementam a interface `Invocation` e para cada tipo de joinpoint existe sua respectiva classe que o representa (ver Anexo 8.5).

Desta forma, o fluxo de execução de um aspecto é iniciado quando a classe interceptada é carregada pela máquina virtual e, assim, são configurados os advices que a interceptam (linhas 59 a 65 do Anexo 8.5). Uma vez que a configuração dos aspectos esteja carregada, ocorre o seguinte fluxo quando o joinpoint interceptado é executado: primeiramente, o código que substitui o joinpoint verifica se há advices configurados para o mesmo (linha 28 do Anexo 8.5). Caso haja, o invocation bean do joinpoint é instanciado com as informações dos advices que interceptam o joinpoint e é atualizado com as informações de contexto[2] e, em seguida, o método `invokeNext`, do respectivo invocation bean, é executado (linhas 30 a 33 do Anexo 8.5). Porém, se não houver, o joinpoint será executado normalmente (linhas 35 e 26 do Anexo 8.5). A partir do array de advices, que são representados pela classe `Interceptor`, cada elemento do

[2] Informações de contexto são informações a respeito do contexto o qual o joinpoint é executado, por exemplo: a instância do objeto o qual o joinpoint pertence, e/ou os parâmetros que o método recebe, se este for um joinpoint de uma chamada de método.

mesmo é invocado de forma semelhante ao padrão de projeto Chain of Responsability (GOF, 1995), em que cada Interceptor invoca o próximo elemento da cadeia de interceptadores até não restar mais nenhum e, assim, o joinpoint interceptado ser invocado.

A interface `Invocation` representa a invocação de um joinpoint (linha 3 do Anexo 8.5) e sua hierarquia é composta por diversas classes que são descritas a seguir. Primeiramente, temos a `InvocationBase` como sendo a superclasse de todos os tipos de joinpoint e contém o código genérico. Já `FieldInvocation` é a classe que representa a leitura (`FieldReadInvocation`) ou escrita (`FieldWriteInvocation`) de um atributo. Por sua vez, `ConstructionInvocation` pode ser definida como a classe que representa a execução de um construtor, enquanto `CallerInovocation` representa a chamada a um método (`MethodCalledByMethodInvocation`) ou construtor (`MethodCalledByConstructorInvocation`). E, finalmente, a classe `MethodInvocation` é definida a partir da sua função de representar a execução de um método (ver Figura 11).

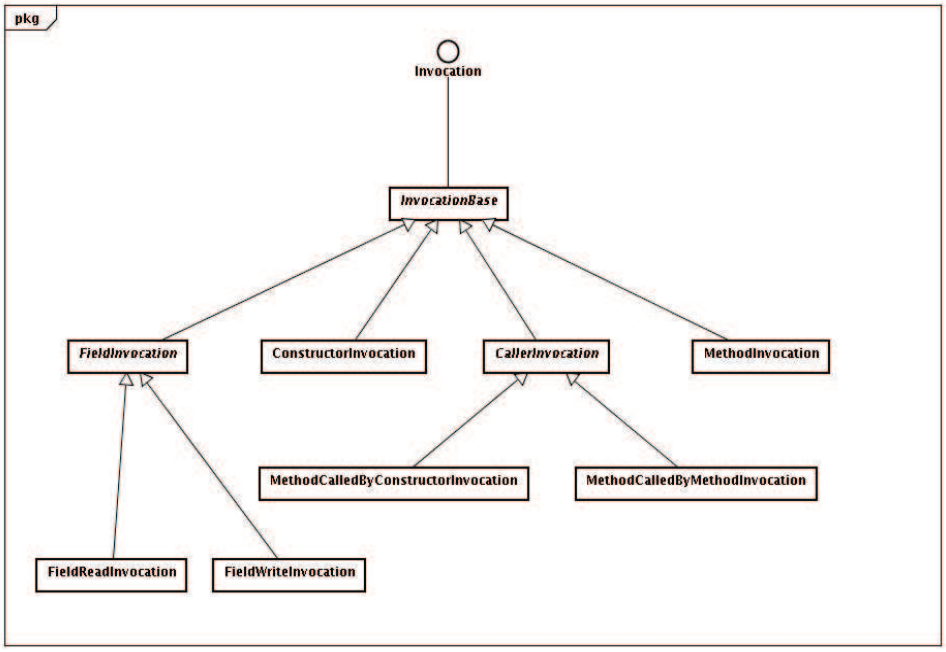

Figura 11 – Diagrama de classes da hierarquia de *Invocation*.

2.1.6.2.Generated Advisor Weaving

O modo de instrumentação Genereted Advisor Weaving foi adicionado na versão 2.0 do JBoss AOP e atualmente é o padrão, ele adiciona algumas características ao JBoss AOP como:(i) manipulação os aspectos a nível de instancia da classe; (ii) simplificação do código a ser adicionado quando os advices são do tipo before, after, after-throwing e finally; e (iii) melhorias na instrumentação de métodos que são sobrescritos por subclasses.

A adição dessas características impactou na forma com que a classe é instrumentada e, como consequência da manipular de advices ao nível de instância, foram definidas algumas regiões protegidas de acesso concorrente, como pode ser visto nas linhas 7, 17, 19, 31, 42 e 44 do Anexo 8.6.

Outro impacto na instrumentação é a adição de advisors à classe interceptada, que tem por finalidade gerenciar a adição e remoção dos advices desta. Por sua vez, o código dessas classes na classe interceptada tem por objetivo remover o uso de reflexão para executá-lo e as citadas classes contém um método (`aop$mc1`, linha 5 do Anexo 8.6) e um atributo (`jpmb1`, linha 24 d o Anexo 8.6) para representar cada joinpoint interceptado. O método, portanto, tem a responsabilidade de inicializar o atributo do respectivo joinpoint (`jpmb1`, no Anexo 8.6) e invocar o método `invokeJoinPoint` para que os advices sejam executados.

Além da geração de código no processo de weaving neste modelo de instrumentação, ocorre a geração de código em tempo de execução. Como pode ser observado nas linhas 64 a 66 do Anexo 8.6, o método `invokeJoinpoint` possui implementação vazia, sendo que, apesar disso, esse método é preenchido em tempo de execução com o código que realiza as chamadas para os advices. Essa transformação ocorre na execução do método `generateJoinPointClass` que é definido na classe `GeneratedClassAdvisor`, que, por sua vez, é a superclasse dos advisors que foram adicionados à classe interceptada, como é mostrado nas linhas 26 a 82 do Anexo 8.6.

Portanto, assim como na instrumentação Classic Weaving, o joinpoint interceptado é substituído por outro código, apesar de na instrumentação Generated Advisor Weaving o código que substitui a construção ser uma chamada ao método criado no advisor para representá-lo (aop$mc1, no Anexo 8.6).

Dadas as características acima, o fluxo de chamadas, quando um joinpoint interceptado é executado e funciona de acordo com os passos detalhados a seguir: (i) primeiramente, o método que representa o joinpoint no advisor é invocado; (ii) em seguida, o advisor verifica se há advices para o joinpoint, (iii) se não houver, o código do joinpoint original é executado normalmente; (iv) caso haja, o advisor verifica se já existe uma instância da classe que representa o joinpoint, pois (v) se não houver uma instância, o código da classe que representa o joinpoint será modificado em tempo de execução para receber os códigos relativos a invocação dos advices, e uma instância desta será atribuída ao atributo relativo ao joinpoint; (vi) entretanto, se houver, esta instância receberá a invocação do método `invokeJoinPoint`, dando inicio à cadeia de chamadas dos aspectos que interceptam o joinpoint.

Com o intuito de tornar o código do o Anexo 8.6 mais legível, optamos por substituir alguns nomes de variáveis e classes como mostrado na Figura 12 abaixo apresentada:

```
Aop$methodCall__N_158521554798583179 4Pojo_N_88911729678064011
40 = aop$mc1
methodCall__N_158521554798583179 4Pojo_N_8891172967806401140  =
mcN15
JoinPoint_MByM__N_158521554798583179 4Pojo_N_88911729678064011
40 = JP1
joinPoint_MByM__N_158521554798583179 4Pojo_N_88911729678064011
40 = jp1
joinpoint_MByM__N_158521554798583179 4Pojo_N_88911729678064011
40 = jpmb1
```

Figura 12 – Mapeamento entre o valor original no arquivo o valor que o substituiu.

2.2.Mecanismos de Tratamento de Exceções

Um sistema de software consiste em uma série de componentes que colaboram para fornecer um conjunto de serviços. Uma falha ocorre quando o serviço prestado pelo sistema não é o que foi especificado. Erro é um estado computacional não desejado que pode levar a uma falha. Um defeito pode causar

erros, onde este defeito tanto pode ser físico no hardware ou criado a partir de deslizes de programação. Os desenvolvedores de sistemas e muitas vezes referem-se à falhas como exceções, de forma que raramente se espera que se manifestem durante a execução do sistema.

Proporcionalmente ao aumento da complexidade das aplicações e ao número de componentes utilizados para a construção de um sistema, cresce, também, o número de estados computacionais que podem levar a uma falha, tornando-se necessário, assim, o gerenciamento desses estados de forma mais modular. Nesse sentido, linguagens de programação modernas como Java, Ada e C ++ possuem estruturas específicas para lidar com o mecanismo de tratamento de exceção (GOODENOUGH, 1975a; GARCIA E RUBIRA, 2001), permitindo que o código que irá lidar com o fluxo de exceção seja diferente daquele que tratará do fluxo normal da aplicação. Assim, a separação entre o fluxo normal e o fluxo de exceção de um programa, faz com que os desenvolvedores reflitam sobre as condições de erro que podem acontecer durante a execução do programa antes de realizar a codificação e, desta forma, preparar o software para lidar com os fluxos excepcionais.

Esta dissertação tem por foco o mecanismo de manipulação de exceção implementado na linguagem Java – que também é adotado no JBoss AOP. Portanto, as próximas seções buscarão descrever, brevemente, como o mecanismo de tratamento de exceção da linguagem Java funciona e relacionar cada elemento desse mecanismo com os conceitos POA.

2.2.1.Mecanismos de Tratamento de Exceção em Programas JBoss AOP

A fim de facilitar o entendimento sobre fluxos de exceção de sistemas orientados a aspecto, apresentamos, neste trabalho, os principais conceitos do mecanismo de tratamento de exceções implementado na linguagem Java, correlacionando-os com as construções disponíveis no JBoss AOP.

O mecanismo de tratamento de exceção é composto por quatro conceitos principais, sendo eles: (i) a exceção, (i) o sinalizador de exceção, (iii) o bloco de captura de exceção, e (iv) o modelo de exceção que define como sinalizadores e os blocos de captura estão vinculados.

- Sinalizador de Exceção – Uma exceção é gerada por um elemento – o método ou estruturas semelhantes a este, como, por exemplo, um advice – quando um estado anormal é detectado. Consequentemente, sempre que uma exceção é gerada dentro de um elemento incapaz de lidar com a mesma, esta é sinalizada para o elemento que o chamou. Assim, o sinalizador de exceção é o elemento que detecta o estado anormal e levanta a exceção. Na Figura 13, o advice ad2 detecta a condição anormal e levanta a exceção EX. Uma vez que este advice intercepta o C método, tal exceção será incluída no citado método juntamente com o comportamento adicional encapsulado no advice.
- Bloco de captura de exceção – O bloco de captura de exceção é o código chamando em resposta a uma exceção levantada e pode ser ligado a regiões protegidas, como, por exemplo, métodos, classes e blocos de código (GARCIA E RUBIRA, 2001). Os blocos de captura de exceção são responsáveis por executar as ações que tem por função trazer o sistema de software de volta ao estado normal e, sempre que isto não é possível, parar o sistema de forma segura. Portanto, em programas orientados a aspecto, um bloco de captura pode ser definido em um método ou advice, sendo que existem tipos específicos de advices capazes de lidar com as exceções como é o caso de after-throwing, por exemplo, não sendo geral essa propriedade aos advices.
- Modelo de exceção – Em muitas linguagens, a busca pelo bloco de captura de exceção, cuja funcionalidade é tratar a exceção, ocorre por toda a cadeia de invocação dinâmica (do inglês: method call chain), aumentando, em contrapartida, a reutilização de software, uma vez que o responsável por chamar uma operação pode manipulá-lo quando em contexto mais amplo (GOODENOUGH, 1975a). Portanto, nos programas orientados a aspecto, o manipulador de uma exceção pode estar presente em um dos métodos na cadeia de chamadas até o método que levanta a exceção, ou em um aspecto que intercepta qualquer um dos métodos da cadeia de chamadas.

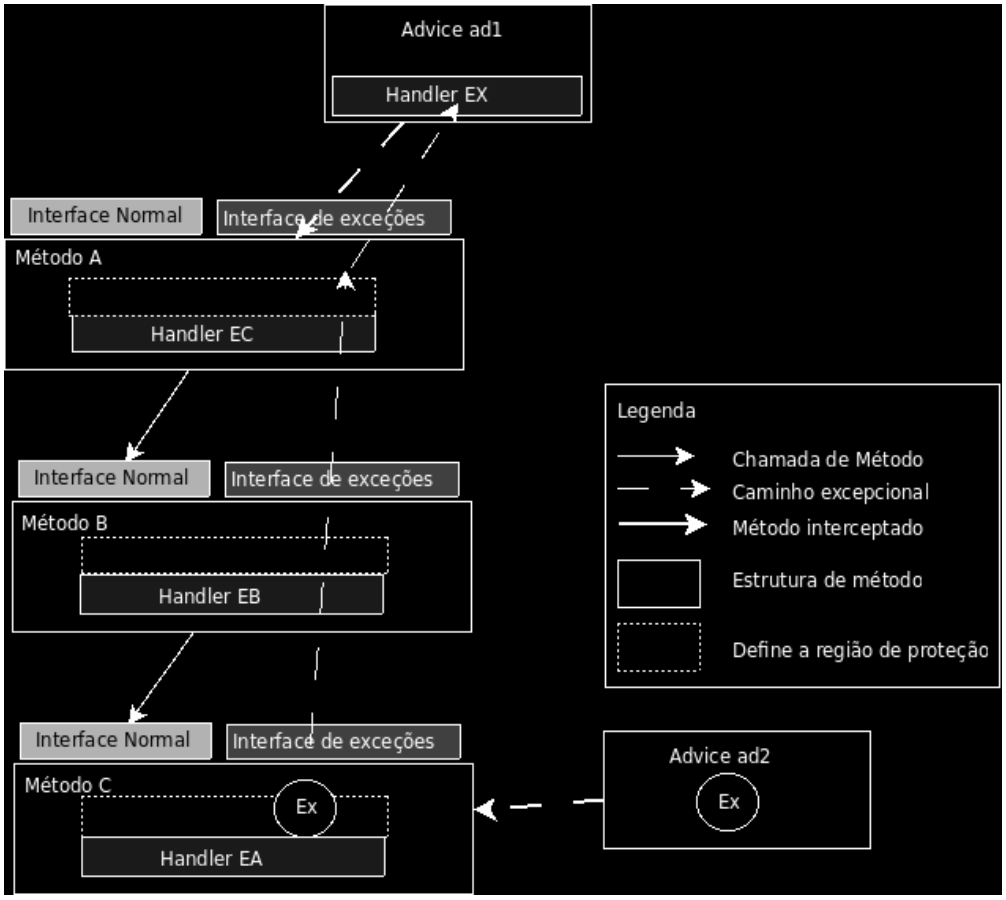

Figura 13 – Exemplo de fluxo de exceção na presença de aspectos.

Para fins didáticos, chamaremos, ao longo desta dissertação, de caminho da exceção o caminho percorrido pela mesma na cadeia de chamadas dinâmicas desde o ponto em que é lançada até ser capturada ou escapar do programa. Assim, na Figura 13 ilustrada acima apresenta um exemplo de caminho de exceção em que a exceção `Ex` é lançada pelo advice `ad2` e capturada pelo `advice ad1`, `seguindo` o caminho `ad2 → C → B → A → ad1`.

No entanto, além desses três principais conceitos que compõem um mecanismo de tratamento de exceção, existem alguns outros relacionados com o citado mecanismo no que diz respeito à linguagem Java – que será usada ao longo deste trabalho – e que serão descritos a seguir.

O primeiro deles é chamado de Interfaces de exceção (MILLER E TRIPATHI, 1997) e será abordado mais detalhadamente nesta seção. Primeiramente, constata-se que o responsável por chamar um método precisa saber quais exceções que podem vir a partir da interface do mesmo, sendo que,

assim, podemos dizer que quem chama o método é o componente capaz de preparar o código antecipadamente para as exceções que podem vir a acontecer durante a execução do sistema. Por este motivo, algumas linguagens permitem associar à assinatura de um método, uma lista de exceções que podem ser lançadas por este. No entanto, isso, além de fornecer informações para quem chama o método, pode ser utilizado para verificar, em tempo de compilação, se os blocos de captura foram definidos para cada exceção que foi especificada, sendo essa lista de exceções definida por Miller e Tripathi (1997) como a especificação de exceções ou interface exceção de um método.

Idealmente, a lista de exceções definida na interface deveria proporcionar informações completas e precisas para o usuário do método, porém algumas linguagens, tais como Java e JBoss AOP, permitem ao desenvolvedor contornar este mecanismo. Nessas linguagens, as exceções podem ser (i) checadas, quando precisam ser declaradas na assinatura do método que as levanta; e (ii) não-checadas, quando não precisam ser declaradas no método que as levanta; e, como consequência, o cliente do método não conhece quais exceções não-checadas podem ser lançadas por este, a menos que ele, recursivamente, inspecione cada método chamado.

Por conveniência, esta dissertação divide este conceito de interface de exceções em duas categorias, sendo a primeira delas chamada de interface de exceções explícita por fazer parte de uma assinatura de estrutura (método ou estruturas semelhantes a métodos) e declarar explicitamente as exceções; e a segunda intitulada interface completa de exceções por conter todas as exceções lançadas por um módulo, incluindo as implícitas, ou seja, as não especificadas na assinatura.

No restante desta dissertação, a menos que seja explicitamente mencionado, usaremos a expressão interface de exceções para nos referirmos à interface completa de exceções, pois, embora tanto a interface normal (assinatura do método, por exemplo) quanto a interface de exceção de um método possam evoluir ao longo do ciclo de vida do software, o impacto dessas alterações no sistema varia significativamente. Desta forma, quando a assinatura de um método é modificada, o sistema é afetado localmente, ou seja, somente os clientes do método são diretamente afetados. Todavia, por outro lado, a retirada ou inclusão de novas exceções em uma interface de exceções pode ter impacto no sistema

como um todo, uma vez que os blocos de captura de exceção podem estar em qualquer lugar do código. Conforme mostrado na Figura 13, um aspecto pode adicionar comportamento a um método sem alterar a interface normal desse método, ao mesmo tempo em que a adição do comportamento pode acrescentar novos tipos de exceções e, portanto, impactar a interface de exceções desse método.

O segundo conceito que compõe o mecanismo de tratamento de exceção e que merece destaque na presente dissertação é a hierarquia de captura de exceções. Em linguagens orientadas a objeto, normalmente, esse conceito suporta a classificação de exceções em hierarquias de tipo de exceção (MILLER E TRIPATHI, 1997; e RUBIRA E GARCIA, 2001). Já em Java, por sua vez, a interface de exceção é composta pelos tipos das exceções que podem ser lançadas por um método. Assim, cada bloco de captura é associado a um tipo de exceção, que especifica as que podem ser tratadas por ele. Desta forma, a representação das exceções em hierarquias de tipo permite que superclasses capturem subclasses (MILLER E TRIPATHI, 1997; ROBILLARD E MURPHY, 1999).

O terceiro e último conceito a ser destacado nesta dissertação é definido por Goodenough (1975) e chamado de Contextos de Captura de Exceção (doravante CCE) que podem ser definidos como sendo as regiões específicas em um programa onde as classes de exceções são tratadas sempre da mesma forma. Assim, cada CCE está associado a um ou mais blocos de captura responsáveis pelo tratamento de exceções de uma determinada região. Portanto, quando uma exceção é sinalizada dentro de um CCE, um bloco de captura pode ser escolhido dentre os que foram associados ao CCE de acordo com o tipo da exceção sinalizada pelo mesmo. Porém, se não houver nenhum bloco associado ao CCE para o tipo ou supertipo da exceção levantada, esta escapará para o próximo método do fluxo de chamadas e não será sinalizada por esse mecanismo.

2.2.2.Construções para Captura de Exceções em JBoss AOP

Tanto em JBoss AOP como em Java, blocos `try` definem contextos para o tratamento de exceções, blocos `catch` definem os blocos de captura de exceção, e o bloco `finally` define instruções que serão executadas independentemente de ocorrer ou não um exceção (GOSLING et alii, 1996). Nesses casos, as exceções

são tratadas como pertecentes a uma hierarquia representada neste trabalho na Figura 14 abaixo:

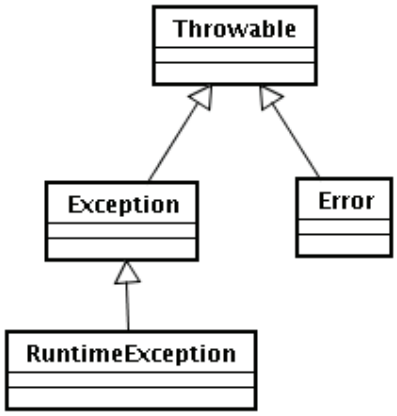

Figura 14 – Hierarquia de exceções na linguagem Java.

De acordo com essa estrutura, podemos constatar que toda exceção deve ser considerada uma instância da classe `Throwable`, sendo que as exceções definidas pelo usuário devem ser (i) checadas, quando se estendem de `Exception;` ou (ii) não-checadas, quando a sua extensão surge de `RuntimeException`. Por convenção, uma exceção que se desdobra de `Error` representa uma condição irrecuperável e, normalmente, representa problemas na plataforma (GOSLING et alii, 1996; ROBILLARD E MURPHY, 2003).

Desta forma, as exceções checadas devem ser declaradas como parte da assinatura do método que as propagou e o uso de exceções checadas permite que o compilador estático verifique se os blocos de captura adequados foram fornecidos pelo sistema. O uso de exceções checadas, no entanto, é custoso de se manter (DOOREN E STEEGMANS, 2005), uma vez que se uma nova exceção for criada, todos os métodos do fluxo de chamadas devem declarar essa exceção (na cláusula `throws` definido em sua assinatura) ou capturá-la.

Por outro lado, as exceções não-checadas não precisam ser declaradas na interface de seus sinalizadores, mas, como consequência, o compilador não verifica se há ou não um bloco que seja capaz de tratá-la. Desta forma, fica vetado ao cliente de um método saber facilmente quais as exceções não-checadas que podem ser lançadas por este a menos que ele, cuidadosamente, inspecione o

código do método e os que foram chamados a partir deste – o que pode levar um longo tempo ou até mesmo ser inviável.

Em contrapartida, quando as bibliotecas são utilizadas pela aplicação, o colaborador não tem acesso aos códigos-fonte e por isso precisa confiar na documentação da biblioteca sobre as exceções que podem ocorrer em tempo de execução - que, muitas vezes, não são nem completas e nem precisas (THOMAS, 2002; SACRAMENTO et alii, 2006). Como consequência, não capturar exceções não-checadas pode ser visto como uma das principais fontes de erros nos sistemas em Java (JO et alii., 2004).

O JBoss AOP reutiliza o mesmo mecanismo de tratamento de exceções de Java para levantar (`throw`), tratar (`try-catch-finally`) e especificar (`throws`) exceções na assinatura do método. Em JBoss AOP, as interfaces de exceção do advice devem ser baseadas nas interfaces de exceção dos métodos interceptados e devem seguir uma regra semelhante à "Regra Conformidades de Exceção" (MATSUOKA E YONEZAWA, 1993; MILLER E TRIPATHI, 1997) aplicada durante a herança, quando os métodos são sobrescritos. Como resultado, um advice só pode lançar uma exceção checada, se esta for lançada por "todo" o método interceptado e, para burlar esta limitação, a maioria dos advices lança exceções não-checadas que, por sua vez, não precisam ser especificadas por cada método interceptado. Contudo, a verificação das assinaturas em JBoss AOP só é realizada quando o advice é associado ao método interceptado, e isto ocorre, ou em tempo de carregamento do aspecto, ou em tempo de execução, quando a adição de advices é feita programaticamente. Portanto, durante o processo de weaving do JBoss AOP não é realizada nenhuma checagem em relação ao tratamento de exceção.

Algumas das construções do JBoss AOP podem ser usadas para capturar exceções, como apresentados a seguir:

- Advice finally e after-throwing – estes tipos de advice permitem que os aspectos sejam invocados quando uma exceção é lançada por um método, desta forma, permitem executar um trecho de código quando uma exceção é lançada. Esses códigos podem, por exemplo, encapsular a exceção original em uma nova.
- Advice around e interceptor – estes advices encapsulam o joinpoint interceptado, e são capazes de incluir comportamento antes ou

depois dele, ou até mesmo substituir o joinpoint. Estes advices podem ser utilizados para tratar exceções lançadas dentro do joinpoint interceptado e desta forma, retornando o programa para o seu fluxo normal.

2.3.JBoss AS

As empresas atualmente precisam ampliar seu alcance, reduzir seus custos e diminuir o tempo de resposta dos seus serviços aos clientes, colaboradores e fornecedores e, normalmente, os aplicativos que oferecem esses serviços combinam os sistemas de informações coorporativas (SIC) já existentes na empresa, adicionando somente novas funções. Tais serviços devem possuir algumas características necessárias e básicas para o seu pleno funcionamento como serem, por exemplo, (i) altamente disponíveis, para atender as necessidades do ambiente empresarial atual; (ii) seguros, com a finalidade de proteger a privacidade dos usuários e a integridade da empresa; e (iii) confiáveis e escaláveis, a fim de garantir que as transações ocorram com precisão e sejam prontamente efetuadas.

Na maioria dos casos, a arquitetura utilizada para o desenvolvimento de novos serviços é dividida em múltiplas camadas que aproveitam as intermediárias por elas integrarem os SICs existentes com as funções de negócios e dados dos serviços que estão sendo criados. Nesse sentido, temos os servidores de aplicação que são plataformas de middleware para desenvolvimento e implantação de software baseada em componentes e que oferecem um ambiente no qual os desenvolvedores podem implantar e gerenciar componentes de software (FLEURY E REVERBEL, 2003) que, por sua vez, tanto podem ser desenvolvidos pelos próprios desenvolvedores ou por terceiros. Assim, a maioria dos servidores de aplicação implementa um dos padrões da indústria atualmente adotados para a aplicação server-side componentes, podendo ser Java Enterprise Edition (doravante JEE), .NET ou CORBA. Desta forma, cada uma destas implementações define um modelo de componentes adequados para uma classe de aplicação de componentes diversa.

Dentre os padrões citados anteriormente, o JEE tem se destacado no desenvolvimento de aplicações coorporativas e define quatro tipos de

componentes que um produto que o implemente deve suportar, sendo eles (i) aplicação cliente, (ii) applets, (iii) servlets e páginas JSP e (iv) Enterprise JavaBeans (EJB). O primeiro deles consiste em programas na linguagem de programação Java que fazem a interface com o usuário e, normalmente, são executados em um computador desktop, enquanto que o segundo é composto de componentes de interface gráfica que, geralmente, são executados em um navegador Web, mas podem ser executados em uma variedade de outros ambientes ou dispositivos, desde que estes suportem o modelo de programação applet. O terceiro compreende componentes que são, normalmente, executados em container Web e respondem a requisições HTTP de clientes Web. Servlets e páginas JSP são referidos como "Componentes *Web"* e podem ser utilizados para gerar páginas HTML que podem ser a interface de usuário de um aplicativo e, também, podem ser usados para gerar XML ou outro formato de dados que são consumidos por outros componentes. Aplicações Web, por sua vez, são compostas de componentes web e são executadas em um container Web. Há, ainda, um tipo especial de Servlet que fornece suporte para serviços da Web utilizando o SOAP sobre o protocolo HTTP. Por fim, os últimos deles são formados por componentes de negócio e são executados em um ambiente gerenciado que oferece suporte a transações.

Um container tem por objetivo dar suporte para a execução de componentes JEE e fornece uma visão unificada das APIs suportadas pelo padrão JEE. Os componentes de uma aplicação desse tipo nunca interagem diretamente com outros componentes JEE e usam os protocolos e métodos de um container para interagir uns com os outros e com os serviços da plataforma (SHANNON, 2004). Assim, o container se interpõe entre os componentes de aplicação e os serviços JEE e, assim, permite que o container de forma transparente possa injetar os serviços definidos descritores de implantação (do inglês deployment descriptors) dos componentes, tais como gerenciamento de transações declarativas, as configurações de segurança e o gerenciamento de recursos, como, por exemplo, conexões com um banco de dados. Os descritores de implantação são, por sua vez, arquivos de configuração que determinam as configurações dos serviços que o container ira fornecer à aplicação.

O Servidor de aplicações JBoss AS implementa as especificações JEE e a versão deste servidor que estudamos para esta dissertação é a versão 4.2.2 que

implementa a especificação 1.4 do JEE. Por conveniência desta dissertação será apresentado somente o container EJB o qual é utilizado nos experimentos.

2.3.1.EJB

Desenvolver componentes reutilizáveis, seguros, consistentes e escaláveis é o objetivo de todo desenvolvedor de software e foi com essa finalidade que a especificação de componentes EJB foi desenvolvida. Componentes EJB normalmente contêm a regra de negócio da aplicação, e assim eles são chamados de componentes de negócio. Contudo, a construção desses componentes normalmente possui outros requisitos além da própria regra de negócio, tais como a segurança, a persistência e a distribuição. Quando o framework EJB é utilizado para o desenvolvimento de componentes, estes requisitos são deixados a cargo do container EJB e, em se tratando do servidor de aplicações JBoss AS, o mesmo implementa os requisitos utilizando aspectos, mais especificamente o JBoss AOP (FLEURY E REVERBEL, 2003).

Segundo a especificação do framework EJB 2.1 (DEMICHIEL, 2003), o modelo de componentes do framework EJB é flexível de forma que um componente EJB pode representar (i) um serviço sem estado (Stateless Session Beans); (ii) um serviço sem estado que recebe as requisições de um webservice (Stateless Session Bean); (iii) um serviço sem estado cuja invocação é assíncrona e orientada a chegada de mensagens JMS (Message Driven Beans); (iv) um serviço que possui estado para uma conversação com um cliente em particular. Tais serviços de sessão, o seu estado de conversação é mantido automaticamente por todos os métodos que o cliente chamar (Statefull Session Beans) e; (v) um objeto persistente e que contém regras de negócio podendo ser compartilhado entre vários clientes (Entity Beans).

Além disso, para que os componentes realizem suas tarefas, há um conjunto de serviços associados que são desempenhados de forma transparente pelo container, sendo eles (i) o gerenciamento do ciclo de vida de um componente, visto que, desta forma, a instanciação e a disponibilidade do mesmo não ficam a cargo do desenvolvedor e sim gerenciadas pelo container através de arquivos de configuração; (ii) o gerenciamento de transações, apesar das definições sobre como estas transações serão gerenciadas serem realizadas pelo desenvolvedor

através de arquivo de configuração; (iii) a distribuição, ainda que a possibilidade de que um componente seja local ou distribuído fique a cargo das configurações realizadas para o componente; (iv) a segurança, pois neste aspecto a definição de quem tem autorização para acessar o componente é feita pelo arquivo de configuração, que, por sua vez, é utilizado para definir o funcionamento de diversos serviços do container para o componente e é chamado de descritor de implantação (do inglês deployment descriptor) como pode ser visto na Figura 15 abaixo.

```
1. <ejb-jar version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
2.     <display-name>hcare-ejb</display-name>
3.     <enterprise-beans>
4.         <session>
5.             <display-name>PharmacySB</display-name>
6.             <ejb-name>PharmacyBean</ejb-name>
7.
   <home>coshms.ejb.emergency.PharmacyRemoteHome</home>
8.           <remote>coshms.ejb.emergency.PharmacyRemote</remote>
9.             <ejb-class>coshms.ejb.emergency.PharmacyBean</ejb-
   class>
10.               <session-type>Stateless</session-type>
11.               <transaction-type>Container</transaction-type>
12.           </session>
13.   <assembly-descriptor>
14.           <container-transaction>
15.               <method>
16.                   <ejb-name>PharmacyBean</ejb-name>
17.                   <method-name>*</method-name>
18.               </method>
19.               <trans-attribute>Required</trans-attribute>
20.           </container-transaction>
21.   </assembly-descriptor>
22.   </ejb-jar>
```

Figura 15 – Trecho do descritor de implantação da aplicação health care.

No exemplo acima é apresentado a definição de um Session Stateless Bean (linhas 4 a 12) o qual define que seu gerenciamento de transação será realizado pelo container (linha 11) e que todos os métodos requerem estar num contexto transacional (linhas 15 a 21).

No framework EJB, um componente EJB é a composição dos seguintes elementos (i) descriptor de implantação (deployment descriptor), que contém as configurações para os serviços que o container disponibiliza; (ii) bean, classe que contém a regra de negócio; (iii) interface Home, é a interface que o desenvolvedor utiliza para solicitar ao container instancias do componente de negócio; (iv) interface Remote, é a interface de negócio do bean e, é através dela que o cliente do bean interage com o objeto de negócio que está no container; Esta interface define que o bean será acessado a partir de outra máquina virtual;(v) interface Local, tem as mesmas caracteríscas da interface remote, contudo os beans estão na mesma máquina virtual.

2.4.SAFE

A ferramenta SAFE (Static Analysis for the Flow of Exceptions) (Coelho, 2008) foi desenvolvida para estaticamente analisar o bytecode de programas interceptados pos aspectos AspectJ, para encontrar: (i) as interfaces de exceção de aplicação e métodos de advice, e (ii) os caminhos que cada exceção percorre quando é levantada. Ela utiliza uma série de heurísticas para analisar as aplicações interceptadas por aspectos maiores detalhes em Coelho (2008).

A ferramenta baseia-se no framework de SOOT para a análise estática do bytecode, e ela usa o SPARK, um dos construtores de grafo de chamadas fornecido pelo SOOT. A ferramenta SAFE considera que todas as exceções não-checadas e checadas, desde que sejam explicitamente lançadas por aspectos ou classes do aplicativo, ou que as exceções sejam implicitamente lançadas (por exemplo, através de métodos da biblioteca).

Desta forma, podemos constatar que essa ferramenta fornece informações a respeito do tratamento de exceções da aplicação analisada que não seriam facilmente encontradas através de uma análise manual. Além disso, essas informações permitem a detecção de padrões de defeitos no código e, utilizando

as informações dos caminhos de exceção fornecidos pela ferramenta, identificam o exato ponto do defeito para poder, assim, corrigi-lo da melhor forma.

Entretanto, apesar das características e das vantagens apresentadas nesta seção, a ferramenta em questão possui algumas limitações como (i) a baixa performance, de forma a dificultar ou até mesmo a inviabilizar a análise das bibliotecas utilizadas pela aplicação; (ii) o elevado consumo de memória, pois além da representação Jimple do código analisado, a SAFE armazena o grafo de chamadas da aplicação em memória, corroborando para uma degradação da eficiência da ferramenta; (iii) o uso do algoritmo CHA (ver Seção 3.2.2.1), que pode levar a análise de caminhos que não ocorrem na aplicação, elevando o número de nós do grafo de modo a ser necessário filtrar manualmente os caminhos que não existem na aplicação; e, por fim, (iv) a não reutilização da análise dos caminhos já percorridos, uma vez que o número de caminhos excepcionais que uma aplicação pode ter é exponencial (ver Seção 3.1.4).

2.5.Resumo

Neste capítulo foram apresentadas as ideias e ferramentas que deram suporte a esta dissertação e, dentre elas, tem-se o JBoss AOP que é uma implementação do paradigma de orientação a aspectos e que possibilita que os aspectos sejam compostos com o código da aplicação tanto em tempo de compilação quanto em tempo de carregamento da aplicação. O JBoss AOP provê duas formas de compor os códigos dos aspectos com os da aplicação que são a Classic Weaving e a Generated Advisor Weaving, em que tanto uma quando a outra não validam as interfaces de exceção dos aspectos durante o weaving, já que a validação só ocorre quando a aplicação é carregada.

Outro suporte da dissertação que foi apresentado é o mecanismo de exceções em Java e JBoss AOP, que fazem uso, ambos, dos mesmos conceitos de (i) exceção, (ii) sinalizador de exceção, (iii) bloco de captura de exceção, e (iv) modelo de exceção. Adicionalmente, o JBoss AOP provê os advices finally e after-throwing, que atuam diretamente na captura de exceções e os advices around e interceptor, que encapsulam o joinpoint interceptado, permitindo, desta forma, que a exceção seja capturada.

O framework EJB faz parte da especificação JEE que é utilizada no servidor de aplicações JBoss AS para implementar os serviços prestados pelo container EJB e o JBoss AS faz uso do JBoss AOP com a mesma finalidade. No entanto, devido ao fato de o modelo de componentes do framework EJB ser flexível, permitindo que componentes com características diferentes participem do mesmo framework, o container EJB utiliza a funcionalidade de domínios do JBoss AOP para especificar um domínio para cada tipo de componente EJB, determinando, desta forma, quais os aspectos que interceptam cada tipo de componente EJB descrito.

Por fim, tratamos também da SAFE, que é a idéia base da dissertação, por ser essa uma ferramenta de análise estática que tem por finalidade descobrir fluxos de exceção em programas Java e AspectJ. Além disso, essa ferramenta também percorre uma representação do programa, obtida através do framework SOOT, utilizando um grafo de chamadas em busca de todas as exceções que podem escapar do programa e do caminho excepcional de cada exceção do programa. Porém, assim como qualquer outra ferramenta, SAFE possui algumas limitações e dificuldades que foram apresentadas nesta dissertação tais como, como qualquer ferramenta, ela possui limitações e dificuldades, tais como (i) o fato de analisar somente as linguagens Java e Aspect J; (ii) a performance; e a (iii) necessidade de intervenção manual para filtrar caminho que não são possíveis de ocorrer devido ao uso do CHA (Call Hierarchy Analysis).