3
Ferramenta SAFE-JBoss AOP

Este trabalho propõe uma extensão da ferramenta SAFE (Static Analysis for the Flow of Exceptions) (Coelho, 2008), a qual tem por objetivo analisar o código compilado de uma aplicação Java ou AspectJ e extrair métricas a respeito dos fluxos de exceção da aplicação. As extensões propostas neste trabalho são as seguintes: (i) analisar aplicações que utilizem o JBoss AOP; (ii) analisar aplicações que utilizem o framework EJB, mais especificamente, a implementação utilizada pelo servidor de aplicações JBoss AS; e (iii) adicionar algoritmos para análise baseados em Point-to (ANDERSEN, 1994). Chamamos esta versão estendida da ferramenta SAFE de SAFE-JBossAOP (Static Analysis of the Flow of Exceptions for JBoss AOP).

A ferramenta proposta SAFE-JBossAOP recebe como entrada um programa escrito na linguagem Java, um arquivo de configuração da própria ferramenta e um arquivo de configuração do JBoss AOP. A partir desses dados, a ferramenta percorre o código da aplicação analisada de forma semelhante a uma execução, buscando elementos de códigos que lancem ou que capturem exceções para, assim, calcular os caminhos excepcionais do programa. Esta é, portanto, uma análise semelhante à feita para encontrar os pares definição-uso (MYERS, 2004), apesar desta percorrer o programa utilizando as instruções de chamadas a métodos ao invés de analisar grafos de fluxo de dados.

Ao término do processamento, a ferramenta fornece, como saída, métricas relativas ao tratamento de exceção do programa fornecido, tais como (i) o número de exceções por tipo; (ii) o número de exceções encapsuladas por tipo; (iii) o número de exceções relançadas por tipo; (iv) o número de exceções agrupadas por tipo da exceção, tipo da classe que a lançou (aspecto ou classe comum) e que a capturou (aspecto ou classe comum); (v) o número de exceções que escapam, agrupado a partir do tipo da exceção e da classe que a lançou (aspecto ou classe comum); e, por fim, (vi) a listagem de todos os fluxos excepcionais que ocorrem na aplicação.

As seções seguintes visam detalhar alguns conceitos e técnicas utilizadas pela ferramenta, assim como apresentar os principais componentes que fazem parte da mesma e como estes trabalham para calcular os caminhos excepcionais.

3.1.Técnicas Empregadas

As técnicas utilizadas na ferramenta têm por objetivo melhorar a precisão dos caminhos excepcionais e contornar problemas inerentes às tecnologias utilizadas e, a partir desta constatação, podem ser levantadas algumas questões quando se analisa aplicações Java que utilizem o JBoss AOP como as seguintes:

(i) Como são tratadas as classes relativas aos aspectos ?
(ii) Como os blocos finally são analisados?
(iii) Como é tratado o encadeamento de exceções?
(iv) Como detectar que um método é interceptado por aspectos do JBoss AOP (ver Seção 2.1.6.1)?
(v) Como evitar a redundância na análise de métodos (ver Seção 2.4)?
(vi) Como melhorar a precisão dos caminhos excepcionais?

Assim, com o intuito de responder a essas perguntas norteadoras do trabalho, foi necessária a elaboração de um conjunto de técnicas que serão apresentadas na seguinte seção.

3.1.1.Forward Analysis

No JBoss AOP, os aspectos são implementados como classes Java, o que significa dizer que se uma aplicação possui um aspecto `K`, este aspecto será representado pela classe `K`, que , por sua vez, possui um método `m` que será o advice do aspecto (ver Seção 2.1). Sendo assim, se um método `m1` da classe `A` e o método `m2` da classe `B` de uma aplicação são interceptados pelo por um aspecto `K`, cabe dizer que os métodos `m1` e `m2` invocarão o método `m` do aspecto `K`. Essa característica do JBoss AOP, portanto, pode causar a análise de caminhos inexistentes na aplicação (ver Figura 18).

O problema dessa análise ocorre quando o fluxo de chamadas da aplicação possui um ou mais aspectos e estes interceptarem mais de uma classe; e quando esse fluxo de exceção é analisado do ponto em que a mesma é lançada até o ponto em que ela escapa ou é capturada, utilizando como guia o grafo de chamadas previamente construído. Para exemplificar o problema, são apresentados a seguir dois fluxos distintos chamados fluxo 1 e fluxo 2. O fluxo 1 inicia-se no método `m1` da classe `A`, que, por sua vez, chama o método `m2` da classe `B` que chama o método `m3` da classe `C`, onde a exceção `E1` é levantada. O fluxo 2 inicia-se no método `m1` da classe `X`, que, por sua vez, chama o método `m4` da classe `Y` que chama o método `m5` da classe `Z` onde ocorre a exceção `E2`. Desta forma podemos dizer que, inicialmente, não há aspectos envolvidos na aplicação exemplo como pode ser visto na Figura 16 abaixo:

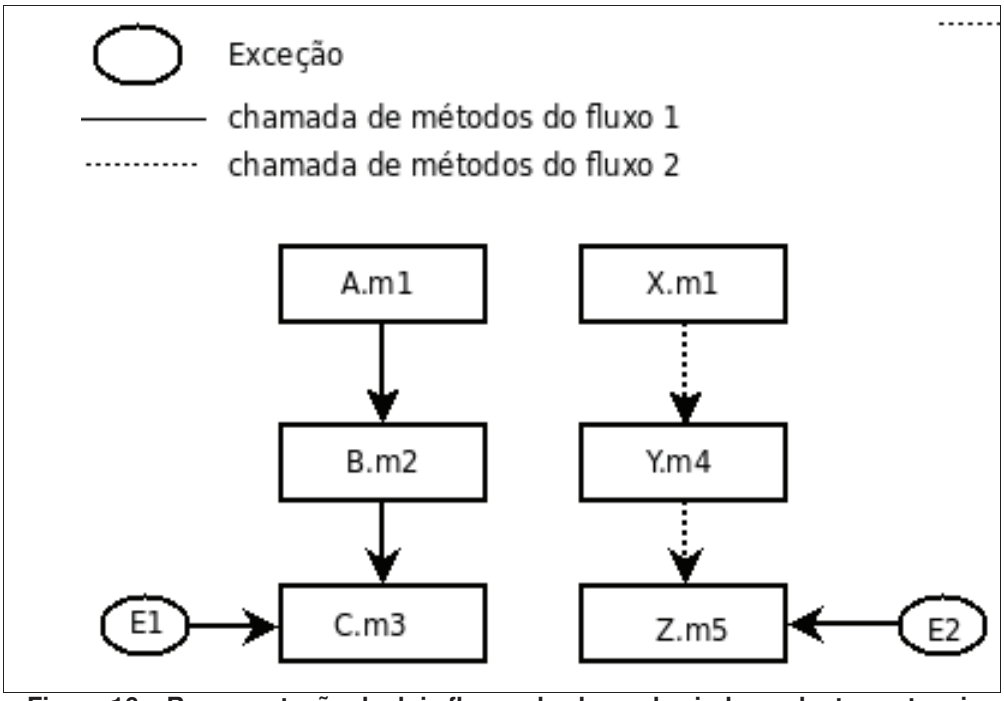

Figura 16 – Representação de dois fluxos de chamadas independentes entre si.

Agora, adicionando o aspecto `K` que tem por pointcut métodos cujas classes possuem métodos cujo nome inicia com `m1` e o método `m` atuando como advice after, que executa após o término do método interceptado (ver Seção 2.1). Sendo assim, as classes `A` e `X` se enquadram no critério, e o aspecto é adicionado ao fluxo. São apresentadas as mudanças nos fluxos com a adição do aspecto na Figura 17.

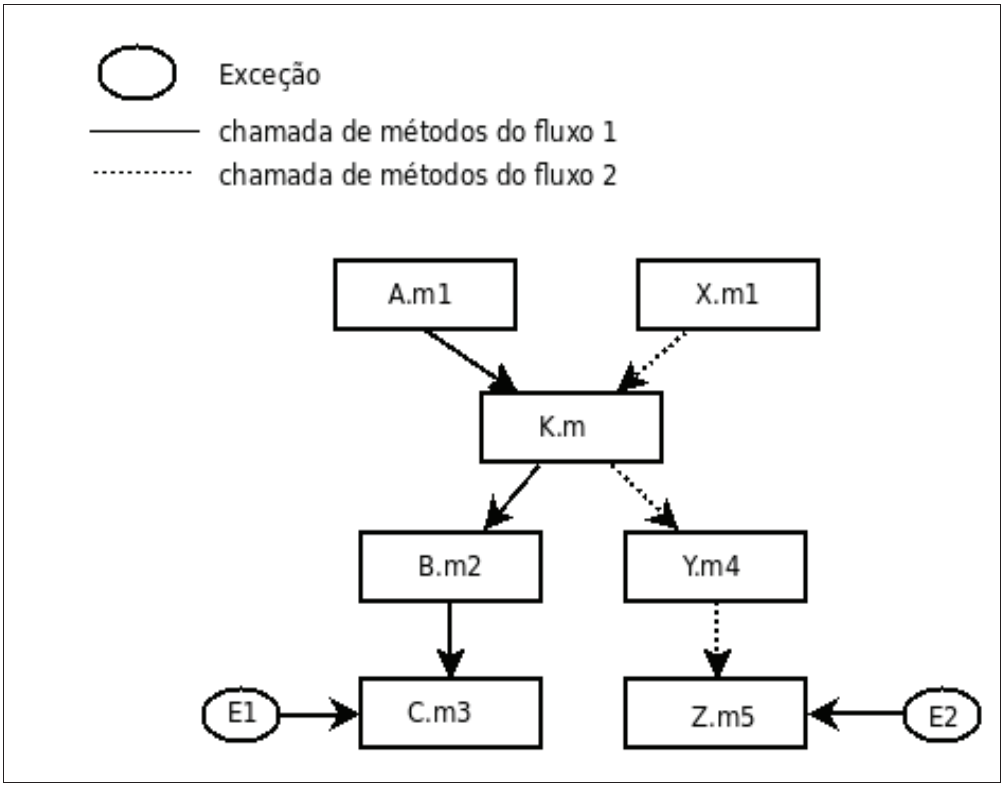

Figura 17 – Representação de dois fluxos de chamadas independentes entre si e em presença de um aspecto que intercepta os dois fluxos.

Portanto, o fluxo de chamadas da aplicação, apresentado na Figura 16, é transformado na seguinte sequência de chamadas: (i) o método `m1` das classes `A` e `X` chamam o método `m` do aspecto `K`, (ii) o método `m` chama os métodos `m2` e `m4` das classes `B` e `Y`, respectivamente e (iii) o método `m2` chama o método `m3` da classe `C` e o método `m4` chama o método `m5` da classe `Z` (ver Figura 17).

O erro de análise ocorre quando o fluxo de exceção é analisado no sentido inverso, ou seja, quando a aplicação é analisada, neste exemplo, a partir do método `m3` da classe `C` ou do método `m5` da classe `Z`. Analisando o fluxo a partir do método `m3`, temos que o método invocador de `m3` é o método `m2` da classe `B`, que é invocado pelo método m do aspecto `K`, que, por sua vez, é invocado pelos métodos `m1` das classes `A` e `X`. Entretanto, não existe, no fluxo original, a chamada do método m1 da classe `X` para o método m2 da classe B e também não existe no fluxo com o aspecto a sequência de chamadas `X.m1` → `K.m` → `B.m2` → `C.m3` (ver Figura 18).

Portanto, utilizar grafos na análise de aplicações que são compostas por aspectos do JBoss AOP leva à análise de caminhos inexistentes, causando

imprecisão na análise do fluxo de exceção. Isso de deve ao fato de que, quando analisamos um grafo de chamadas que contêm aspectos, é necessário ter informações de contexto que indiquem o caminho percorrido, ou seja, a aresta `K.m` → `B.m2` só existe se a análise percorreu o elemento `A.m1`.

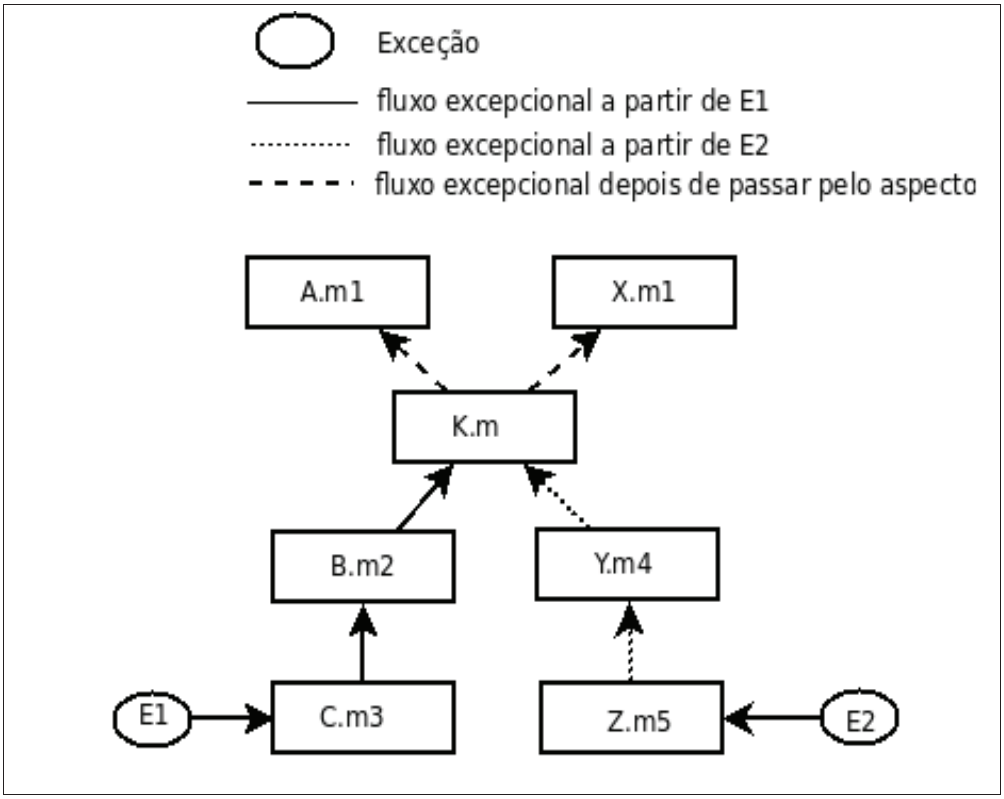

Figura 18 – Representação do fluxo de exceção a ser analisado em presença de aspectos.

A fim de evitar o problema acima demonstrado, a ferramenta proposta SAFE-JBossAOP não constrói um grafo de chamadas, constrói, em contrapartida, listas encadeadas de chamadas para cada fluxo de exceção que contêm os métodos que foram encontrados a partir do que iniciou o fluxo até o que levanta a exceção.

3.1.2.Tratamento para Finally

Blocos `try-finally` ou `try-catch-finally` são utilizados por desenvolvedores quando desejam garantir que um trecho de código será executado independentemente do que ocorra no bloco definido pelo `try`, ou seja, o código será executado mesmo que ocorra uma exceção dentro do bloco definido pelo `try`.

Como exemplo, temos a linha 5 na Figura 19 que é executada independentemente do que ocorra no bloco definido pelo try-finally da linha 2 até na linha 4.

```
1. public static void main(String[] args){
2.   try{
3.     System.out.println("A");
4.   } finally{
5.     System.out.println("B");
6.   }
7. }
```

Figura 19 – Exemplo para ilustrar o funcionamento do bloco finally.

Como podemos perceber nas linhas 4 a 9 da Figura 20, o bloco *finally*, quando compilado, dá origem a um bloco catch - que será analisado pelas ferramentas de análise estática como a SAFE, visto que a mesma não sabe quando um bloco `catch` foi gerado a partir de um `finally`. Desta forma, para evitar que nossa ferramenta computasse como uma exceção que foi relançada e, consequentemente, calculasse todos os caminhos excepcionais a partir dela, usamos a seguinte técnica: **Exceções `Throwable` que são relançadas não são consideradas como capturadas**. Desta forma, o processo de análise em questão buscará o próximo método do fluxo de exceção, e, assim, irá desconsiderar o bloco `catch`.

```
1. public static void main(String[] args){
2.   try{
3.     System.out.println("A");
4.   } catch (Throwable t){
5.     System.out.println("B");
6.     throw t;
7.   }
8.   System.out.println("B");
9. }
```

Figura 20 – Código decompilado com a ferramenta Jad.

3.1.3.Encadeamento de Exceções

Frequentemente, aplicações utilizam padrões de projeto que definem camadas e, naturalmente, o objetivo dessas camadas é separar as responsabilidades e promover a independência entre elas. Desta forma, exceções de uma camada não devem chegar à superior, pois isto violaria o padrão de camadas.

A fim de evitar, então, que o tratamento de exceção viole o padrão em camadas e também para não perder a rastreabilidade do erro, exceções de uma camada inferior são, geralmente, encapsuladas em outra exceção e relançadas (FU E RAYDER, 2007), mantendo-se, assim, o padrão em camadas da aplicação (ver Figura 21).

```
public void processarInformacao(int valor) throws AppException{
      try{
            gravaValorNoBancoDeDados(valor);
      } catch (Exception e){
            throw new AppException(e);
      }
     }
```

Figura 21 – Exemplo do encapsulamento de uma exceção.

Em alguns casos, a exceção capturada não é encapsulada em outra e re-lançada, mas apenas algumas informações da exceção capturada são extraídas e repassadas para a nova exceção que é relançada (ver Figura 22).

```
public void processarInformacao(int valor) throws AppException{
      try{
            gravaValorNoBancoDeDados(valor);
      } catch (Exception e){
            throw new AppException("Ocorreu erro no acesso ao
banco de dados" + e.getMessage());
      }
     }
```

Figura 22 – Exemplo da passagem de dados de uma exceção para outra.

Portanto, na nossa ferramenta, utilizamos a mesma técnica utilizada por Fu e Ryder (2007) que consiste em fazer uma análise do bloco `catch` e verificar se alguma informação que veio da exceção capturada foi passada para o construtor da exceção criada, pois caso tenha sido, a exceção é adicionada as estatísticas como encapsulada e a análise prossegue para o próximo método do fluxo de exceção utilizando a nova exceção para buscar os respectivos blocos `catch`. Maiores detalhes sobre esta análise encontram-se na Seção 3.1.6.

3.1.4.Programação Dinâmica

Analisar o fluxo de chamadas de uma aplicação e extrair métricas de seu fluxo de exceção é semelhante a calcular os números de Fibonacci, pois suas funções de cálculo são semelhantes. Observa-se esta semelhança na Figura 23 e na Figura 24, sendo que a primeira apresenta a função de Fibonacci e a segunda, a função para cálculo das métricas de uma aplicação, onde `FA(X)` é uma função que retorna os métodos que são chamados a partir do método `X`.

```
fib(n) = fib(n-1)  + fib(n-2)
fib(1)=1
fib(0)=0
```

Figura 23 – Função para cálculo do número de Fibonacci.

`FM(P) = ∑FM(K)`, onde `K= FA(P)` e `FA(X)`

Figura 24 – Função para cálculo de métricas.

Uma abordagem para calcular os números de Fibonacci é utilizar a técnica "dividir e conquistar", na qual um problema `P` é subdivido recursivamente em subproblemas `(P1 ... PN)` até que o subproblema não seja divisível, sendo que a solução do problema `P` consiste na combinação das soluções dos subproblemas (TARDOS E KLEINBERG, 2005). Contudo, existem algumas classes de problemas que têm por subproblemas um número exponencial e, dentre eles, encontra-se o cálculo do número de Fibonacci.

Analisando a complexidade da implementação da Figura 23, intuitivamente, percebe-se que a execução deste algoritmo possui complexidade

exponencial de `O(2^n)`. Para entender melhor o algoritmo, um exemplo de sua execução pode ser visto na Figura 25.

```
     fib(5) = fib(4) + fib(3)
     fib(4) = fib(3) + fib(2)
     fib(3) = fib(1) + fib(2)
     fib(2) = fib(0) + fib(1)
     fib(1) = 1
fib(0) = 0
```

Figura 25 – Execução do algoritmo de Fibonacci para o valor 5.

Uma forma de reduzir a complexidade para calcular os números de Fibonacci é utilizar a técnica de programação dinâmica (TARDOS E KLEINBERG, 2005) que consiste em estender a técnica de "dividir e conquistar" e, ao invés de calcular todos os subproblemas, o algoritmo armazena a solução dos subproblemas já calculados numa área de memória AM e verifica se o mesmo já foi solucionado, pois, se isto ocorre a solução armazenada em AM é utilizada e, caso contrário, a mesma será calculada.

Portanto, utilizamos a técnica de programação dinâmica para melhorar a performance da ferramenta da seguinte forma: (i) são armazenadas para cada contexto de execução o número de fluxos de exceção que escapam para cada tipo de exceção, estes valores é que é chamado de estatística das exceções do contexto; (ii) quando o analisador encontra um contexto que já foi analisado, ele analisa para cada exceção, que foi armazenada nas estatísticas, os método da cadeia de chamadas, adicionando às respectivas estatísticas de contexto os valores encontrados no método analisado.

3.1.5.Identificando Advices

O JBoss AOP determina quais os aspectos que interceptam os métodos de uma classe durante o processo de carregamento da classe pela máquina virtual. Por conta disto é necessário acessar a API (Application Programming Interface) do JBoss AOP para conhecer quais os aspectos interceptam um método.

Contudo, executar esta operação para todos os métodos da aplicação analisada leva a um aumento desnecessário no tempo de processamento, pois, no processo de weaving, os métodos sofrem modificações e estas modificações

possuem padrões conhecidos (ver Seção 2.1.6). Assim, é computacionalmente menos custoso detectar estes padrões e só então acessar a API do JBoss AOP para recuperar quais os aspectos que interceptam o joinpoint.

Existem dois tipos de padrões que o JBoss AOP utiliza para modificar o código que são o Classic Weaving e o Generated Advisor Weaving. A ferramenta SAFE-JBossAOP foi desenvolvida para analisar a Classic Weaving, detalhes destes dois modelos de instrumentação de código podem ser vistos na Seção 2.1.6 abordada anteriormente.

Para identificar se um método, no modelo Classic Weaving, foi interceptado, verifica-se se a classe à qual o método pertence implementa a interface `Advised` e se o método analisado possui uma instrução de chamada para o método `invokeNext` da classe que implemente a interface `Invocation`. Esse modelo pode ser visto no código obtido através da ferramenta Jad (JAD, 2000) e expresso abaixo:

```
public void setSessionContext(SessionContext sessioncontext)
    {
        MethodInfo methodinfo =
(MethodInfo)aop$MethodInfo_setSessionContext794397754050783862.ge
t();
        ClassInstanceAdvisor classinstanceadvisor =
(ClassInstanceAdvisor)_getInstanceAdvisor();
        Interceptor ainterceptor[] =
methodinfo.getInterceptors();
        if(ainterceptor != (Object[])null || classinstanceadvisor
!= null && classinstanceadvisor.hasInstanceAspects)
        {
            if(classinstanceadvisor != null)
                ainterceptor =
classinstanceadvisor.getInterceptors(ainterceptor);
            setSessionContext_794397754050783862
setsessioncontext_794397754050783862 = new
setSessionContext_794397754050783862(methodinfo, ainterceptor);
            setsessioncontext_794397754050783862.arg0 =
sessioncontext;

setsessioncontext_794397754050783862.setTargetObject(this);

setsessioncontext_794397754050783862.typedTargetObject = this;

setsessioncontext_794397754050783862.setAdvisor(aop$classAdvisor$
aop);
            setsessioncontext_794397754050783862.invokeNext();
        } else
        {

coshms$ejb$emergency$PharmacyBean$setSessionContext$aop(sessionco
ntext);
        }
    }
```

Figura 26 – Trecho de código que mostra o código decompilado de um método que foi interceptado por um aspecto.

3.1.6.Point-to

Em linguagens orientadas a objeto existem três tipos de invocação de métodos: (i) invocação de um método de instância, (ii) invocação de um método de interface, e (iii) invocação de métodos estáticos. Na invocação de um método estático o corpo do método está atrelado à classe, ou seja, os códigos relativos à implementação do método estão vinculados à classe `C`. Já em relação aos métodos de instância, tem-se que, quando ocorre a invocação de um método `m` de uma classe `C`, o corpo do método que será executado pode não ser o corpo do método `m` que está codificado na classe `C`, visto que esta classe pode ter uma ou mais subclasses em que o método em questão pode ser sobrescrito. Desta forma, é necessário determinar qual o tipo da instância para saber com precisão qual o corpo do método que deve ser analisado. Por fim, os métodos de interface são métodos que não possuem códigos de implementação, sendo assim, para que este seja executado é necessário que uma classe contenha os códigos de implementação relativos ao método. E, assim, podemos dizer que se assemelha aos métodos de instância, visto que nesses, é necessário conhecer a instância a qual o método esta vinculado para determinar qual o corpo do método que será analisado.

Desta forma, para determinar com precisão qual o corpo de método a ser analisado, a nossa ferramenta utiliza uma abordagem semelhante à análise point-to, que tem por objetivo determinar o valor das variáveis do programa analisado (ANDERSEN, 1994). Entretanto, ao invés de construir um grafo como é feito na abordagem point-to, nós construíremos uma mapa de valores (`K` → `V`) que representa associações, onde `K` pode ser uma variável ou um atributo de classe/objeto e `V` é o valor que a variável/atributo possui. Esta decisão de projeto foi tomada porque o objetivo da ferramenta não é determinar a corretude do fluxo de dados e sim determinar o valor que uma variável possui, para então extrair a informação de tipo. Assim, o mapa de valores funciona como uma heap de memória, ou seja, guardando os valores relativos às variáveis e atributos da aplicação.

Desta forma, o algoritmo desta abordagem analisa o método, instrução a instrução, buscando por instruções de associação. Uma instrução de associação, (`A->B`) indicando que `A` recebe o valor `B`, atribui um valor que está à direita `B` a

variável/atributo à esquerda `A`. O valor `B` pode ser (i) outra variável/atributo, (ii) uma operação de cast, (iii) um parâmetro, (iv) a instância do próprio objeto, (v) uma invocação de método, (vi) uma instrução de alocação de instância `new`, ou (vii) uma constante.

Sendo assim, quando o valor `B` da associação é uma instrução de alocação de instância, esta instrução é associada com a variável/atributo `A`, pois, esta instrução é a que determina qual a classe da instância (ver Tabela 3). Por outro lado, quando o valor `B` é uma constante então o valor da constante é associada com a variável/atributo `A`, pois ela também determina o tipo da instância (ver Tabela 3).

Outra forma de associar valores pode ser feita através de atributos que tanto podem estar associados a uma instância quanto serem estáticos, este último representado no mapa como uma tupla (classe, atributo) (ver Tabela 3).

Tabela 3 – Representação da heap quando um valor é atribuído a um atributo.

<table>
<tr><th>Aplicação A2</th><th>Heap(mapa de valores)</th></tr>
<tr><td>

```
Class A {
 public static String valor;
 public static void main
(String[] args){
     a.valor = "a1";
     }
}
```

</td><td>
<table>
<tr><th>Variável</th><th>Valor</th></tr>
<tr><td>(A, valor)</td><td>“a1”</td></tr>
</table>
</td></tr>
</table>

Entretanto, quando o atributo está associado a uma instância, o mesmo será representado como uma tupla (instância, atributo) (ver Tabela 4). A instância de um objeto é representada (i) por uma instrução new, quando a mesma for um objeto que foi alocado; (ii) pelo valor da constante, quando a instância for de um tipo básico (`Integer`, `Float`, `Double`, `String`, dentre outros); e (iii) por um valor definido pela nossa ferramenta, chamado Dumb, para representar que não há informações sobre a instância.

Tabela 4 – Representação da heap quando um valor é atribuído a um atributo.

Aplicação A2	Heap(mapa de valores)

`Class A {`
` public String valor;`
` public static void main`
`(String[] args){`
`  A a = new A();`
`  a.valor = "a1";`
`  String[] values = args;`
`  }`
`}`
	Variável / **Valor**
	a / New A
	(New A, valor) / "a1"
	values / dumb

Uma instrução de associação ainda pode se apresentar de uma outra maneira e isso se dá quando a associação é realizada entre variáveis/atributos, pois o valor que está sendo referenciado pela variável/atributo `B` passa a ser recuperado do mapa de valores e copiado para a variável/atributo `A` (ver Tabela 5).

Tabela 5 – Representação da heap para exemplificar associação simples da aplicação A1.

Aplicação A1	Heap(mapa de valores)

`String a= "a1";`
`String b=a;`
`System.out.println(b);`
	Variável / **Valor**
	A / "a1"
	B / "a1"

A associação entre variáveis ocorre de forma similar, pois quando o valor `B` da associação é uma instrução de `cast`, a ferramenta copia o valor referenciado pela instrução de para a variável/atributo `A` da associação (ver **Erro! Fonte de referência não encontrada.**).

Tabela 6 – Representação da heap para exemplificar associação com cast da aplicação A1.

Aplicação A1	**Heap(mapa de valores)**	

`String a= "a1";`
`Object b=a;`
`String c=(String)b;`
`System.out.println(c);`
	Variável	**Valor**

	a	"a1"
	b	"a1"
	c	"a1"

Assim, quando um método `m` que possui parâmetros é invocado, os valores dos mesmos e a instância do método `m` são extraídos do mapa de valores (heap) e passados como parâmetro para o método da recursão do algoritmo do componente `ForwardCallAnalyser` que analisará o método `m` (ver Seção 3.2.1.2.1 na linha 16 da Figura 31). Consequentemente, quando for feita a análise point-to do método, as informações dos parâmetros serão associadas às suas respectivas variáveis, permitindo, desta forma, a análise inter-procedural da aplicação (ver Seção 3.2.1.2.1 na linha 7 da Figura 31).

Tabela 7 – Representação da heap quanto a análise inter-procedural.

Aplicação A3	**Heap(mapa de valores)**	

`Class A {`
` private String valor;`
` public void setValor(String s){`
`  valor=s;`
` }`
` public String getValor(){`
`  return valor;`
` }`
` public static void main`
`(String[] args){`
`  A a = new A();`
`  a.setValor("a1");`
`  }`
`}`
	Variável	**Valor**

	a	New A
	s	"a1"
	(New A, valor)	"a1"

Por fim, quando o valor `B` da associação é uma chamada a um método, o método é avaliado e seu retorno é associado ao atributo/variável `A` da instrução de associação (ver Tabela 8).

Tabela 8 – Representação da heap quando uma variável recebe o retorno de um método.

Aplicação A4	Heap(mapa de valores)

```
Class A {
 private String valor;
 public void setValor(String s){
   valor=s;
 }
 public String getValor(){
    return valor;
}
 public static void main
(String[] args){
     A a = new A();
     a.setValor("a1");
     String b= a.getValor();
     }
}
```

Variável	Valor
a	New A
s	"a1"
(New A, valor)	"a1"
b	"a1"

3.1.6.1.Point-to na Presença de Aspectos

No JBoss AOP, a invocação dos advices e do joinpoint é feita através de uma classe intermediária chamada de Inovation Bean (ver Seção 2.1.6) em que cada execução do joinpoint cria uma nova instância dessa classe para receber o contexto de invocação do mesmo. Sendo assim, a instância e os parâmetros de chamada são informados aos atributos da invocation bean e passados ao joinpoint através de reflexão.

A técnica utilizada para contornar este problema foi armazenar o contexto de execução, uma vez detectada a chamada a um aspecto (ver Seção 3.1.5), ou seja, o valor da instância e os parâmetros do método são armazenados numa área de memória até que o joinpoint seja executado e os valores sejam repassados para ele.

3.2.Ferramenta SAFE-JBossAOP

A ferramenta SAFE-JBossAOP foi desenvolvida com o objetivo de analisar os programas que utilizam JBoss AOP e foi construída utilizando o SOOT – um framework para análise estática a partir do código compilado. A Ferramenta SAFE-JBossAOP busca por todos os tipos de exceções checadas e não-checadas, desde que a exceção seja explicitamente lançada pelo código da aplicação, bibliotecas ou dos aspectos utilizados.

O processo para análise do fluxo de exceções de uma aplicação é realizado em três etapas: (i) preparar o ambiente para análise, (ii) classes e métodos são percorridos para montar uma coleção de fluxos excepcionais; (iii) cada fluxo de exceção[3] gerado é analisado separadamente para que sejam calculadas estatísticas sobre as características destes fluxos (e.g., número de exceções por classe da exceção, número de exceções que escaparam, etc.). Cada etapa corresponde a um componente na arquitetura da ferramenta.

3.2.1.Arquitetura da Ferramenta

O objetivo desta seção é apresentar uma visão de alto nível dos componentes e suas responsabilidades além de apresentar como é realizada a comunicação entre eles. A Figura 27 ilustra a arquitetura da ferramenta que é composta por três componentes, sendo eles:

- **Pré-processador**: é responsável por preparar o ambiente para análise, isto é, responsável por (i) extrair as classes do arquivo da aplicação em diretórios; (ii) analisar a estrutura de aplicações corporativas para identificação dos aspectos; e (iii) compor o código dos aspectos com o código da aplicação nas classes em que se encontram nos diretórios;
- **ForwardCallAnalyser**: é responsável por percorrer as classes e métodos da aplicação analisada e montar a estrutura para análise dos fluxos excepcionais.

[3] Fluxo excepcional é uma sequência de chamadas de métodos semelhante a uma pilha de execução. A base da pilha contém o método que inicia o fluxo e o topo da pilha contém o método que lança a exceção.

- **ExceptionPathAnalyser**: responsável por analisar o fluxo de exceção extraindo informações a respeito de como a exceção foi lançada, quem a capturou, qual a ação tomada na captura, se ela foi relançada (ver Seção 3.1.3). A partir destas informações são geradas métricas, as quais são a saída da ferramenta.

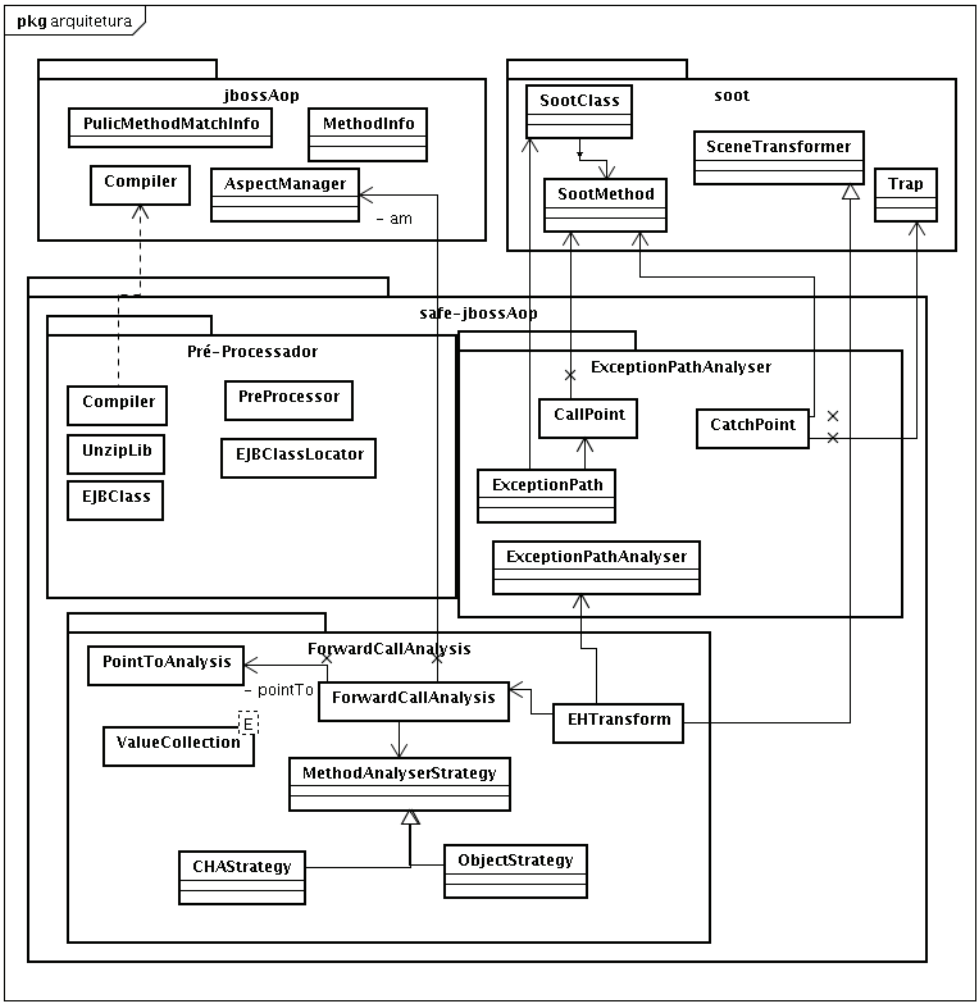

Figura 27 – Arquitetura da ferramenta.

3.2.1.1.Pré-Processador

Aplicações escritas na linguagem Java são, normalmente, distribuídas no formato de um único arquivo que pode estar em um dos formatos de empacotamento descritos abaixo. Cada um destes formatos é, por sua vez, a versão compactada de uma estrutura de diretórios e cada um deles possui uma estrutura e um objetivo específicos.

- **JAR** (Java ARchive) é um arquivo que contém uma ou mais classes, podendo ter ou não outros tipos de arquivos.

- **WAR** (Web ARchive) é um arquivo que contém estrutura para aplicações Web, este pode conter Servlets, Jsps, arquivos JAR, dentre outros.

- **EAR** (Enterprise ARchive) é um arquivo que contém a estrutura de aplicações corporativas podendo conter mais de uma aplicação Web, estruturas EJB (Enterprise Java Beans) e bibliotecas auxiliares.

Outra característica de aplicações Java, quando os aspectos são utilizados, é que alguns frameworks orientados a aspectos permitem que estes sejam combinados com o código base em tempo de compilação ou em tempo de carregamento. Contudo, para analisar aplicações cujos aspectos são adicionados em tempo de carregamento é necessário combinar o código dos aspectos e o código da aplicação antes do início da análise, adicionando, assim, os códigos relativos aos aspectos na aplicação analisada. Porém, o processo de combinação do código dos aspectos com o código da aplicação não é possível se a aplicação estiver empacotada.

Apesar do Soot conseguir analisar aplicações que estejam no formato Jar, ele não consegue analisar os outros formatos. Assim, o objetivo do componente de pré-processamento é converter a estrutura compactada de uma aplicação Java numa estrutura de diretórios e arquivos que torne possível (i) a combinação dos aspectos com o código original da aplicação para a realização da análise estática; (ii) a análise de aplicações que estejam no formato WAR ou EAR.

Este processo possui algumas etapas e a primeira delas é verificar o formato da aplicação (JAR, WAR, EAR) para determinar se ações de descompactação de arquivo e de composição de códigos com aspectos serão necessárias para preparar o ambiente para a análise estática. Essa etapa é realizada pela classe `PreProcessador` e, uma vez determinado o formato da aplicação pela classe *`PreProcessador`*, a mesma delega para a classe *`UnzipLib`* a criação dos diretórios de trabalho de onde serão extraídas e analisadas as classes da aplicação. Em seguida, se a aplicação utilizar o framework EJB, a classe `PreProcessador` solicita a classe `EJBClassLoader` que análise os deployment descriptors encontrados no diretório de trabalho. O objetivo dessa etapa é configurar a classe `Compiler` para só compilar as classes que forem utilizar o framework EJB (ver Seção 2.3.1, mas, se a aplicação não utilizar o framework EJB, a classe `PreProcessador` solicita a compilação de todas as classes da aplicação.

3.2.1.1.1.Aspectos do Framework EJB

Um caso específico na adição de aspectos em tempo de carregamento ocorre quando a aplicação a ser analisada utiliza o framework EJB. No servidor de aplicação JBoss AS os aspectos relativos ao framework EJB, são adicionados em tempo de carregamento e são definidos no arquivo ejb3-interceptors.xml como mostrado na Figura 28 abaixo, que compreende uma versão parcial do arquivo.

```
<aop>
...
<domain name="Stateless Bean">
...
<bind pointcut="execution(public * *->*(..))">
...
    <interceptor-ref name="org.jboss.ejb3.asynchronous.AsynchronousInterceptor"/>
...
    <interceptor-ref name="org.jboss.ejb3.ENCPropagationInterceptor"/>
...
</bind>
...
</domain>
...
</aop>
```

Figura 28 – Trecho do arquivo ejb3-interceptors-aop.xml.

`AspectManager` é um componente do JBoss AOP que gerencia os aspectos que interceptam as classes da aplicação. Já um domínio (Domain), no entanto, tem por finalidade definir um conjunto de aspectos e pointcuts de forma restrita, ou seja, de forma que somente as classes definidas como pertencentes ao domínio sejam interceptadas pelos aspectos definidos no domínio.

O servidor de aplicações JBoss AS utiliza os domínios para a implementação do framework EJB 2.1 (ver Seção 2.1.6) e define alguns domínios e aspectos, sendo estes últimos apresentados nas Tabela 9 e na Tabela 10 abaixo:

Tabela 9 – Aspectos definidos para os domínios statelessSessionBean e StateFulSessionBean.

Nome do Aspecto	Classe
Chamadas assíncronas	`org.jboss.ejb3.asynchronous.AsynchronousInterceptor`
Autenticação	`org.jboss.ejb3.security.AuthenticationInterceptorFactory`
Segurança	`org.jboss.ejb3.security.RunAsSecurityInterceptorFactory`
Transações	`org.jboss.aspects.tx.TxPropagationInterceptor, org.jboss.ejb3.tx.TxInterceptorFactory, org.jboss.ejb3.entity.TransactionScopedEntityManagerInterceptor`
Definidas pelo usuário	`org.jboss.ejb3.interceptor.EJB3InterceptorsFactory`

Tabela 10 – Aspectos definidos para o domínio MessageDrivenBean.

Nome do Aspecto	Classe
Chamadas assíncronas	`org.jboss.ejb3.asynchronous. AsynchronousInterceptor`
Autenticação	`org.jboss.ejb3.security. AuthenticationInterceptorFactory`
Segurança	`org.jboss.ejb3.security. RunAsSecurityInterceptorFactory`
Transações	`org.jboss.aspects.tx. TxPropagationInterceptor, org.jboss.ejb3.tx. TxInterceptorFactory, org.jboss.ejb3.entity. TransactionScopedEntityManagerInterceptor`
Definidas pelo usuário	`org.jboss.ejb3.interceptor.EJB3InterceptorsFactory`

No framework EJB 2.1, o componente EJB é a composição de interface de criação (`Home`), interface de aceso local ou remota (`Remote`/`Local`), classe `bean` e um arquivo de configuração chamado deployment descriptor (ver Seção 2.3.1), que contém as informações necessárias para o servidor de aplicação carregar o citado componente adequadamente. Na Figura 29, podem ser vistas essas informações de forma mais detalhada. A partir das informações `ejb-class` e `session-type` do deployment descriptor do EJB (ver Figura 29) é possível identificar quais são as classes que utilizam o framework EJB, pois as que utiizarem serão combinadas com os aspectos do domínio como mostrado também na Tabela 11 abaixo:

Tabela 11 – Mapeamento de tipo de bean para domínio de aspectos.

Tipo	Domínio
Stateless	Stateless Session Bean
Stateful	Stateful Session Bean
Entity	Não possui
Message	Message Driven Bean

Portanto, se uma classe `B`, por exemplo, é um componente EJB e este é um stateless session bean, então a classe `B` será combinada com os aspectos definidos no domínio Stateless Session Bean (ver Tabela 11).

```
<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">
     <enterprise-beans>
     <session>
     <ejb-name>PharmacyBean</ejb-name>
     <home>coshms.ejb.emergency.PharmacyRemoteHome</home>
     <remote>coshms.ejb.emergency.PharmacyRemote</remote>
     <ejb-class>coshms.ejb.emergency.PharmacyBean</ejb-class>
     <session-type>Stateless</session-type>
     <transaction-type>Container</transaction-type>
     </session>
     …
     </enterprise-beans>
     </ejb-jar>
```

Figura 29 – Exemplo do arquivo de configuração ejb-jar.xml da aplicação hospital care.

Desta forma, o pré-processador precisa checar se a aplicação a ser avaliada utiliza ou não o framework EJB, pois, caso o utilize, o mesmo analisa os arquivos `ejb-jar.xml` encontrados na aplicação com o objetivo de determinar quais classes fazem uso do framework EJB. Em seguida, analisando o deployment descriptor verifica-se quais classes utilizadas fazem parte de um componente EJB e quais os seus respectivos tipos, determinando, assim, o domínio a que cada classe pertence e, consequentemente, os aspectos que a interceptam. Desta forma, a informação de domínio é passada para o compilador do JBoss AOP e a classe é compilada para um domínio informado (ver **Erro! Fonte de referência não encontrada.**).

Contudo, existem alguns tipos de aspectos do EJB que não podem ser analisados, pois estes utilizam uma fábrica de aspectos (ver Seção 2.1.6) e para

determinar se a classe será interceptada por um aspecto. Portanto, para analisar este tipo de aspecto, seria necessário avaliar a execução do método de fábrica, o que não é viável numa análise estática. Um exemplo disto é o aspecto para gerenciamento de transações da aplicação no framework EJB 2.1, visto que esse gerenciamento de transações deste framework define uma política padrão para todos os métodos e estas políticas podem ser especificadas para cada método ou classe do componente EJB. Desta forma, a fábrica de aspectos utiliza configurações do deployment descriptor para definir qual o aspecto que irá interceptar o método.

3.2.1.2.ForwardCallAnalyser

Uma vez que o componente pré-processador montou a infra-estrutura para a análise da aplicação, o componente `ForwardCallAnalyser` é responsável por analisar as classes e métodos e construir os fluxos excepcionais e, para isso, ele utiliza o framework SOOT para montar a representação intermediária do código da aplicação, chamada Jimple.

Uma vez que o SOOT termine de montar a representação intermediária, o framework chama a classe `EHTransform` que tem a responsabilidade de iniciar o processo de análise. Essa classe recupera as informações de configuração da ferramenta através da classe `ConfigUtil` e, a partir destas, determina alguns parâmetros a serem utilizados no processo de análise, tais como a estratégia para análise da aplicação (ver Seção 3.2.2) e as classes que são inicializadoras de fluxo, sendo que neste último caso, utilizamos o termo classe inicializadora de fluxo para designar uma ou mais classes da aplicação que contenham um método que seja invocado quando ocorrer algum evento na aplicação, ou seja, é o método que inicia o fluxo de chamadas na aplicação.[4] Em seguida a nossa ferramenta analisa as classes e métodos de acordo com o algoritmo apresentado em seguida.

[4] Em muitas aplicações, esta classe é a que contém o método main a partir do qual as demais classes são alcançadas.

3.2.1.2.1.Algoritmo para Análise dos Métodos

Para analisar as classes e os métodos de uma aplicação, a ferramenta em questão utiliza o parâmetro *inputPackages* do arquivo de configuração obtido, sendo este o parâmetro definidor de quais classes serão tomadas como inicializadoras de fluxo. A classe *EHTransform* é responsável por, a partir das classes que foram configuradas, percorrer cada classe e cada método (ver Figura 30) definidos pela configuração e, para cada método encontrado, a mesma delega para ForwardCallAnalyser, uma segunda classe, a função de percorrer recursivamente os métodos que são invocados até encontrar o lançamento de uma exceção.

```
1. Main
2.  pa = vazio;
3.  e = estrategia selecionada;
4.  pto = inicializa point-to;
5.  p = pilha vazia;
6.  para todas as classes c que forem inicializadoras de fluxo
7.    para todo método m de c
8.     analisa(m,pa,e,pto);
```

Figura 30 – Pseudocódigo do ponto de inicialização do algoritmo que analisa um método.

Cada método m encontrado é analisado em três passos: (i) análise point-to (ver Seção 3.1.6), (ii) busca por instruções de chamadas de métodos e (iii) busca por instruções que levantam exceção.

A análise point-to (ver linha 7 da Figura 31) tem por objetivo determinar as instâncias das variáveis do método (ver Seção 3.1.6) e, uma vez determinada, sabe-se qual o real corpo de método a ser analisado, melhorando, assim, a precisão da análise do fluxo de chamadas. Neste caso, então, a responsabilidade de realizar esta análise point-to cabe à classe PointToAnalysis.

Ao término da análise point-to, cada instrução do método é analisada e, quando uma instrução de chamada a um método é encontrada, a classe ForwardCallAnalyser delega para MethodAnalyserStrategy a decisão de quais serão os métodos que, realmente, deverão ser analisados (ver Figura 32) e, a

partir da informação de quais serão estes, inicia-se o processo de recursão. No início do processo de análise do método, este é adicionado à pilha de chamadas auxiliar a fim de evitar recursão infinita e de montar a estrutura de dados utilizada pela programção dinâmica para tornar mais eficiente o processamento do fluxo de exceção (ver Seção 3.1.4). Assim, cada elemento da pilha é representado pela classe `CallPoint`, que armazena a instância em que o método foi invocado e os parâmetros que recebe, ou seja, o contexto de execução.

Com a finalidade de melhorar a eficiência computacional da aplicação, o algoritmo utiliza a programação dinâmica (ver Seção 3.1.4). Assim, quando um método `A` chama um método `B` que já foi processado, as estatísticas deste último são combinadas com as estatísticas do primeiro, evitando que toda a cadeia de fluxos do método `B` seja analisada novamente (ver linhas 2 a 5 da Figura 31) (ver Seção 3.1.4 e 3.2.1.2.2).

```
1. analisa(método m, parâmetros pa, pilha p, estrategia e,
        Point-to pto)
2. se e.metodoAnalisado(m,pto,pa);
3.   cfe = estatistica do fluxo de exceções do método m;
4.   adiciona a estatística cfe a cada método da pilha p;
5.   retorna valor do processamento armazenado do método;
6. caso contrário marca o método como analisado;
7. atualiza point-to (pto,pa);
8. adiciona a chamada à pilha p;
9. enquanto houver instruções no corpo do método
10.   se a instrução i for uma chamada de método
11.     mi = método que esta sendo invocado;
12.     se mi for método de aspecto adiciona os aspectos na
         estratégia(e);
13.     mr = e.métodos reais(mi,pto);
14.     para todo método n de mr
15.        pai = parâmetros de invocação do método (n,pto);
16.        atualiza pto com o resultado de analisa(n,pai,p,e);
17.   se a instrução i for um lançamento de uma exceção e não
        estiver num bloco catch
18.      ept = cria objeto exceptionPathTrack com a exceção e a
        cópia da pilha p ;
19.      analisa ept;
```

Figura 31 – Pseudocódigo do método "analisa" do algoritmo que analisa um método.

Quando o algoritmo encontra uma instrução que levanta uma exceção (`throw`), dentro do processo de análise de todas as instruções do método, o objeto `ExceptionStackPath` é criado usando a pilha de chamadas auxiliar e a exceção que foi levantada, sendo que este objeto visa armazenar informações a respeito do caminho que foi percorrido até o lançamento da exceção para, em seguida, ser passado para o componente `ExceptionPathAnalyser` (ver linhas 17 a 19 da Figura 31). Contudo, se a instrução que levanta a exceção estiver dentro de um bloco catch, a ocorrência da instrução `throw` é ignorada, pois ela será analisada quando o fluxo de exceção for analisado.


```
1. métodos reais( método m, Point-to pto)
2. se m for invokeNext retorna o próximo aspecto
3. se m tiver uma instância definida em pto, retorna o método n
   associado ao tipo da instância;
4. caso contrário retorna a lista de métodos que foram
   sobrescritos na hierarquia da classe que define o método m;
```

Figura 32 – Pseudocódigo do método "métodos reais" do algoritmo que analisa um método.

3.2.1.2.2.Programação Dinâmica

A Programação Dinâmica é uma técnica que permite acelerar o desempenho da aplicação utilizando dados previamente calculados (ver Seção 3.1.4) e, na ferramenta proposta, essa técnica é implementada armazenando em uma estrutura de dados (ver **Erro! Fonte de referência não encontrada.**), que fica na classe `MethodAnalyserStrategy`, as informações referentes à classe da exceção, à quantidade de exceções que escapam e ao identificador. Desta forma, essa estrutura de dados está associada a cada contexto de invocação de método, tendo por objetivo evitar que um método que já foi analisado seja processado novamente. Já o identificador, por sua vez, é um conjunto de caracteres que

identifica o contexto de chamada do método. Assim, quando o contexto de chamada é identificado como processado, ao invés de recalcular as estatísticas, o algoritmo irá buscá-las na estrutura de dados e adicioná-las aos métodos da pilha de chamadas.

Um exemplo da estrutura de dados necessária para guardar as informações, considerando que o contexto de chamada é a assinatura do método, pode ser visto abaixo na Tabela 12.

Tabela 12– Estrutura das estatíticas em memória.

Identificador	Dados	
`org.jboss.aop.InstanceAdvisorDelegate: void initialize()`	**Exceção**	**Quantidade**
	`java.lang.RuntimeException`	5
	`java.lang.NullPointerException`	3
`com.meo.safe.teste.interceptor.PojoInterceptor: java.lang.String imprimeERetorna(java.lang.String)`	**Exceção**	**Quantidade**
	`java.lang.IllegallArgumentException`	5
	`java.lang.NullPointerException`	3

3.2.2.Estratégias para Analisar Classes e Métodos

A ferramenta proposta nesta dissertação possui diferentes algoritmos para percorrer o código da aplicação e, a fim de permitir a troca destes de forma simples, foi implementado o padrão de projeto Strategy (GOF, 1995). Este padrão define (i) uma interface chamada `Strategy`, que define as responsabilidades da estratégia e que possui várias implementação chamadas `ConcreteStrategy`; e (ii) a estratégia usada por uma classe chamada `Context`.

Assim, segundo esta ferramenta, a estratégia é representada pela classe `MethodAnalyserStrategy`, que, por sua vez, define quais métodos serão analisados a partir de um método dado e qual é a abstração de contexto[5] utilizada. Determinando assim, a complexidade de tempo e espaço do algoritmo que percorre as classes e métodos da aplicação analisada.

[5] Abstração de contexto é a forma com que o algoritmo representa o contexto de chamada de um método.

Frequentemente, em linguagens orientadas a objeto, o método que está sendo invocado não é o que será executado. E um exemplo disto ocorre quando o método é sobrescrito por uma subclasse como pode ser visto nas linhas 6 e 7 da Figura 33, em que o corpo do método executado não é o método da classe `A` e sim o da classe `A1` (ver Seção 3.1.6). Outro exemplo desse fenômeno ocorre quando um joinpoint é interceptado por um aspecto do JBoss AOP, pois o joinpoint interceptado é substituído por um conjunto de instruções e, dentre elas, ocorre a chamada ao método *`Invocation.invokeNext`* que inicia a execução dos advices que interceptam o joinpoint (ver Seção 2.1.6). Em JBoss AOP não existe ligação entre o advice do aspecto e o método interceptado a nível de códigos na linguagem Java, sendo, portanto, responsabilidade da estratégia encontrar quais advices serão analisados quando um método como o *`invokeNext`* for encontrado (ver Seção 3.2.3).


```
1. public class A {
2.   public void m(){
3.     System.out.println("a");
4.   }
5.   public static void main(String[] args){
6.     A a = new A1();
7.     a.m();
8.   }
9. }
10. public class A1 extends A {
11.   public void m(){
12.      System.out.println("a1");
13.   }
14. }
```

Figura 33 – Exemplo de um método sobrescrito pela subclasse.

Outra responsabilidade da estratégia é definir a abstração de contexto, que é entendida nesta dissertação como sendo a forma com que o algoritmo representa o contexto de chamada de um método e, a ferramenta utiliza um conjunto de caracteres para identificar unicamente o contexto de chamada de um método. Então, fazendo uso dessa estratégia de abstração de contexto, a classe

`ForwardCallAnalyser` verifica se o método já foi analisado ou não (ver linha 2 da Figura 31).

Como, atualmente, existem diversas abstrações de contexto e cada uma delas possui vantagens e desvantagens, implementaremos algumas estratégias que serão explicadas a seguir.

3.2.2.1.CHA (Class Hierarchy Analysis)

Um dos algoritmos mais simples quando se pretende lidar com a análise de fluxos é o CHA (Class Hierarchy Analysis), proposto por Grove e Chambers (2001), pois este considera, por exemplo: se um método `m1`, definido numa classe `A`, é sobrescrito em n subclasses e um método `m2` invoca `A.m1`, para analisar `m2` é necessário analisar todas as classes na qual `m1` é sobrescrito. Assim, o citado algoritmo não leva em consideração quaisquer contextos de invocação do método, permitindo, desta forma, algumas distorções durante o processo de análise como, por exemplo, a invocação do método `toString` da classe `Object`. No trecho de código na Figura 34, se o algoritmo CHA for utilizado, a ferramenta analisará todos os métodos `toString` que foram sobrescritos na aplicação ou em alguma biblioteca.

```
1. public class Exemplo {
2. public static void main(String[] args){
3.     String s = "a";
4.     Exemplo e = new Exemplo();
5.     e.imprime(s);
6. }
7. private void imprime(Object o){
8.     System.out.println(o.toString());
9. }
10.       }
```

Figura 34 – Trecho de código para exemplificar as distorções causadas pelo algoritmo CHA.

A implementação do algoritmo CHA como uma estratégia da ferramenta consiste em usar a assinatura do método como abstração de contexto, significando que um método não é analisado duas vezes, e em verificar, na hierarquia da classe

do método, quais são os métodos que o sobrescrevem para, assim, analisar todos os encontrados.

3.2.2.2.Algoritmo Sensível ao Contexto Usando Objetos

Análise sensível ao contexto baseada objetos foi inicialmente proposta por Milanova et al. (2002) e Milanova (2003). Esta abordagem consiste em analisar cada método de instância e cada construtor separadamente para cada objeto no qual o método ou construtor pode ser invocado. A abordagem usa nomes para objetos para representar os objetos que são alocados em tempo de execução; e se um método ou construtor pode ser invocado a partir de um objeto, o qual será representado por um nome `o`, a abordagem manterá um contexto separado para o método ou construtor que corresponde ao contexto de invocação `o`.

A implementação desta estratégia consiste em (i) usar a referência do objeto concatenado à assinatura do método para abstração de contexto, o que significa dizer que um método não é analisado duas vezes para o mesmo objeto; e em (ii) usar o método que foi definido na classe da instância, pois, caso a mesma não seja conhecida, faz-se necessário que se verifique na hierarquia da classe que define o método quais os métodos das classes que sobrescrevem o método dado e, assim, analisar todos os encontrados.

3.2.3.Funcionamento do Algoritmo na Presença de Aspectos do JBoss AOP

Os aspectos que interceptam as classes e métodos são definidos através de um arquivo XML ou anotações no código do aspecto. Sendo assim, analisando somente código Java não é possível atingir código de um aspecto a partir de um método que foi interceptado.

Portanto, a fim de tornar possível a análise estática de aspectos com JBoss AOP a ferramenta analisa o corpo do método e infere se o método foi interceptado ou não. Se o método foi interceptado (ver Seção 3.1.5), ela utilizará a biblioteca de aspectos do JBoss AOP para retornar a lista dos aspectos que interceptaram o método da aplicação, sendo esta ordenada de acordo com o critério da Tabela 13 e armazenada na classe `MethodAnalyserStrategy`. Assim, quando a classe

`ForwardCallAnalyser` solicitar qual o método que deve ser analisado a classe *`MethodAnalyserStrategy`* verifica (i) se o método chamado é *`invokeNext`* de alguma classe que implemente a interface *`org.jboss.aop.joinpoint.Invocation`* e (ii) se ela está dentro de um contexto de chamada de um aspecto. Se ocorrer as duas condições a lista de aspectos que interceptam o joinpoint é carregada e colocada numa pilha, e a estratégia utiliza a pilha para retornar o próximo método. No entanto, caso as duas condições não forem verdadeiras, o método será analisado normalmente.

Tabela 13 – Tabela para ordenação da chamada de métodos e aspectos.

Advice	Prioridade
Before	0
Around/Interceptor	1
Método Original	1
After	2
Afterthrow	3
Finally	3

Sendo assim, os métodos pertencentes às classes que são aspectos são sempre analisados, a fim de evitar imprecisão na análise, como detalhado na Seção 2.1.

3.2.3.1.ExceptionPathAnalyser

A análise de cada fluxo de exceção é iniciada a cada vez que o componente `ForwardCallAnalyser` encontra a instrução de lançamento de uma exceção ou um método previamente analisado que possui exceções que escapam. Para representar o fluxo de exceção, o componente `ForwardCallAnalyser` constrói uma estrutura representada pelo objeto `ept` da classe `ExceptionPathTrack`. Já o mecanismo de análise de fluxo de exceção, por sua vez, é implementado pela classe `ExceptionPathAnalyser` e consiste em desempilhar as chamadas de métodos contidas na pilha de chamadas até que um

bloco catch que capture a exceção lançada seja encontrado ou a pilha esteja vazia, indicando que a exceção escapou, para que ambas, tanto as informações de pilha de chamadas, quanto a exceção lançada, sejam encontradas no objeto `ept`.

Assim, esse processo de análise tem por objetivo encontrar informações a respeito do que ocorre quando uma exceção é lançada e com essas informações extrair métricas relativas a três variáveis:

- **Tipo do método que lança a exceção** - indica o tipo da classe que lançou a exceção. Existem dois tipos de classes: as que são aspectos e as classes comuns.
- **Tipo do método que captura/deixa escapar a exceção** - indica o tipo da classe que capturou ou deixou escapar a exceção. Existem dois tipos de classes: as que são aspectos e as classes comuns.
- **Tipo de captura** – indica o padrão de captura que pode ser: a exceção relançada, o bloco de captura que constitui um bloco `finally` e a exceção capturada que é encapsulada em outra exceção. Essa variável avalia, além do padrão de captura, a forma como a exceção foi capturada que pode ser por um bloco catch específico para a exceção capturada ou por um catch de alguma de suas super-classes. Maiores detalhes na Seção 3.2.3.1.1.

Utilizando estas três variáveis é realizada a totalização do número de ocorrências de cada tipo de fluxo de exceção, esta totalização também é feita pelo tipo da classe da exceção que foi levantada, cuja classe responsável por contabilizar as estatísticas é *`Statistics`*.

```
public class Exemplo {
void metodoQueLancaExcecao(){
   throw new NullPointerException("null");
}
public static void main(String args[]){
 try{
   metodoQueLancaExcecao();
 } catch (Exception e){
   System.out.println("Erro"+e.getMessage());
}
```

```
    }
```

Figura 35 – Exemplo de captura de exceção.

O exemplo acima (Figura 35) mostra o método de uma classe comum levantar uma exceção ao mesmo tempo em que esta é capturada por uma super-classe (linha 8) gerando uma mensagem (linha 9). As estatísticas geradas pela ferramenta são apresentadas em dois formatos, (i) sepradados por vírgula (CSV) e (ii) para leitura do usuário da ferramenta. Para o exemplo acima, as estatísticas, no formato de leitura para o usuário, são apresentadas na Figura 36 abaixo.

```
    Totais: 1
    Número de exceções lançadas por classes e capturadas por
classe:1
       Exata: 1
       Super-classe: 0
       Relançamentos: 0
       Encapsulada: 0
       NullpointerException:
    Número de exceções lançadas por classes e capturadas por
classe na profundidade 1: 1
       Exata:1
       Super-classe:0
       Relançamentos: 0
       Encapsulada: 0
```

Figura 36 – Exemplo da listagem se saída da ferramenta.

A ferramenta SAFE-JBossAOP provê informações a respeito do tratamento de exceção, estas informações permitem a detecção de anomalias no código. Assim, para facilitar encontrar tais anomalias, a ferramenta grava no arquivo de saída a assinatura de cada método analisado, indicando o caminho percorrido pela exceção.

3.2.3.1.1.Exceção Capturada

Analisar a maneira como as exceções são capturadas nos permite ter uma melhor noção a respeito da robustez da aplicação. Esta robustez pode ser

verificada pela presença ou ausência e patterns ou anti-patterns para captura de exceções, um conjunto destes padrões pode ser encontrado em Coelho (2008).

Analisando a captura da exceção, podem ocorrer diversas formas de tratamento, estas são classificadas de acordo com o conteúdo do bloco que as capturam. Assim, facilitando a análise da aplicação com respeito ao tratamento de exceção.

A nossa ferramenta analisa os seguintes padrões de tratamento de exceção: (i) exceção relançada (Figura 18), (ii) bloco de captura constitui um bloco `finally`, (iii) exceção capturada é encapsulada em outra exceção (Figura 19 e 20) e, (iv) percebendo quando a exceção é capturada por subsunção.

Relançar uma exceção é capturar a exceção `ex` e lançar novamente a exceção `ex` como pode ser visto na Figura 37 abaixo.


```
1. try{
2.   metodoQueLancaExcecao();
3. } catch (Exception e){
4.   System.out.println("Erro"+e.getMessage());
5.   throw e;
6. }
```

Figura 37 – Exemplo de relançamento de exceção.

O código gerado por um bloco `finally` é idêntico ao código de um relançamento de uma exceção exceto quando a exceção que é capturada e relançada é da classe `Throwable` (ver Seção 3.1.2). Portanto, uma exceção `E1` é considerada encapsulada por outra `E2`, se esta contiver a exceção `E1` (ver Figura 38) ou qualquer um de seus atributos (ver Figura 39).

```
1. try{
2.   metodoQueLancaExcecao();
3. } catch (Exception e){
4.   System.out.println("Erro"+e.getMessage());
5.   throw new RuntimeException(e);
6. }
```

Figura 38 – Exemplo de encapsulamento de uma exceção por adição da exceção original.

```
1. try{
2.    metodoQueLancaExcecao();
3. } catch (Exception e){
4.    System.out.println("Erro"+e.getMessage());
5.    throw new RuntimeException(e.getMessage());
6. }
```

Figura 39 – Exemplo de encapsulamento de uma exceção por adição de atributo da exceção original.

Caso não haja passagem de informação da exceção capturada `E1` para a exceção `E2` a exceção é considerada substituída (ver Figura 40). Esta técnica é muito utilizada para separação de camadas de uma aplicação.

```
1. try{
2.    metodoQueLancaExcecao();
3. } catch (Exception e){
4.    System.out.println("Erro"+e.getMessage());
5.    throw new RuntimeException("Ocorreu erro");
6. }
```

Figura 40 – Exemplo da substituição da exceção por outra.

3.2.4.Saída da Ferramenta

Entender o funcionamento do tratamento de exceção é fundamental para assegurar o funcionamento correto de uma aplicação. Tendo isto em mente, a nossa ferramenta provê uma listagem dos fluxos de exceção devidamente classificados com relação ao tratamento de exceção (ver Seção 3.2.4.2) e um conjunto de estatíscas relativas ao fluxo de exceções permitindo visualizar de forma mais abrangente o funcionamento do tratamento de exceções na aplicação (ver Seção 3.2.4.1). Desta forma, o usuário da ferramenta deve inicialmente analisar as estatísticas a fim de identificar anomalias e em seguida, analisar a

listagem dos fluxos de exceção, para identificar no código da aplicação anomalias que por ventura tenham sido encontradas.

3.2.4.1.Estatísticas

As estatísticas da ferramenta são produzidas de forma a quantificar algumas características do tratamento de exceção, e a profundidade dos fluxos de exceção encontrados, de forma a apontar onde as exceções estão sendo levantadas, e assim permitir que seja realizada uma busca nas listagens identificando cada fluxo que compõe as estatísticas (ver Seção 3.2.4.2). Um exemplo de como a ferramenta apresenta as informações estaíticas pode ser encontrado na Figura 36. Abaixo são apresentadas a listagem das características do tratamento de exceção e como interpretá-los na ferramenta.

Exceção não-capturada – ocorre quando não existe nenhum bloco de captura para a exceção que foi lançada. Assim, analisando o fluxo de uma exceção e identificando que neste fluxo a exceção escapa, isto indica uma falha em potencial da aplicação que ocorrerá quando a aplicação percorrer o fluxo identificado. Uma forma de entender o número de exceções que escapam é que este número é diretamente proporcial a possibilidade da aplicação falhar e não se recuperar da falha, pois não haverá nenhum código para executar a recuperação.

Exceções capturadas por bloco especializado – ocorre quando a exceção que foi lançada é capturada por um bloco de captura do mesmo tipo da exceção lançada. O valor do número de exceções capturadas de forma exata é entendido como: quanto maior número de exceções capturadas pelo mesmo tipo indica que o tratamento de exceções é mais preciso e a aplicação conhece a interface de exceções dos métodos que são executados.

Exceções capturadas por subsunção – ocorre quando a exceção é capturada por um bloco de um dos super tipos da hierarquia da exceção. Capturar exceções por subsunção, dificulta a identificação da causa da exceção . além do bloco que captura a exceção pode não ser apropriado para tratar a exceção capturada de forma adequada (ROBILLARD e MURPHY, 2003). Esta estatística

indica a quantidade de fluxos na aplicação que desconhecem as exceções que podem ocorrer dentro de um método, e assim tratando-as de forma genérica.

Exceções encapsuladas capturadas por bloco especializado – ocorre quando uma exceção é capturada por um bloco do mesmo tipo da exceção e ela ou uma parte da informação contida nela é repassada para uma nova exceção. O número de exceções encapsuladas inidica que a aplicação tem conhecimento a repeito das exceções que o método pode lançar, contudo o desenvolvedor da aplicação decidiu que não é interessante repassar a exceção original que foi lançada, mas a informação contida na exceção é relevante para ser repassada na forma de uma outra exceção. Isto ocorre frquentemente quando a aplicação é construída utilizando o padrão de projeto camadas (BUCHMAN et alii, 1996), por exemplo: seja uma aplicação cuja arquitetura é definida em três camadas, camada de apresentação que acessa a camada de negócios, e esta acessa a camada de dados; supondo que uma exceção ocorra na camada de dados, então a camada de negócios deve capturá-la e lançar uma outra exceção, com as informações da exceção capturada, de forma a preservar a abstração das camadas da aplicação e ainda repassar a informação de erro para a camada superior. Então, podemos entender o encapsulamento de exceções como uma forma de preservar a interface de exceções, e o número de fluxos de exceções como um indicativo do quando um componente ou biblioteca mantém o nível de abstração de quem os utiliza em relação ao fluxo de exceções.

Exceções Encapsuladas por Subsunção – ocorre quando uma exceção é capturada por um bloco de um dos super tipos da hierarquia da exceção e a exceção ou uma parte da informação da exceção capturada é repassada para uma nova exceção. O número de exceções encapsuladas indica a aplicação ou biblioteca tem por objetivo agrupar um conjunto de exceções de forma a preservar a interface de exceções. Este número pode ser usado para identificar se as exceções que são lançadas a partir de um componente ou bibliteca estão sendo efetivamente convertidas em exceções que obedeçam ao contrato de exceções.

Substituição de Exceções – ocorre quando uma exceção é capturada por um bloco `catch` e outra exceção é lançada sem repssar nenhuma informação da

exceção capturada para a nova exceção. A substituição de exceções tem objetivo idêntico ao encapsulamento de exceções, contudo na substituição nenhuma informação oriunda da exceção capturada é repassada para a nova exceção. O número deste tipo de fluxos indica que informações relativas à exceção original foram removidas, e assim dificultando a identificação da causa raiz do erro, além de apresentar para o usuário ou ainda para o próprio desenvolvedor informações incompletas a respeito do problema.

Relançamento de Exceções – ocorre quando uma exceção é capturada por um bloco `catch` e a mesma exceção é lançada novamente. O relançamento de exceções tem objetivo realizar uma ou mais operações quando uma exceção ocorre. O número de relançamentos de exceções indica que existem operações estão sendo realizadas nos fluxos de exceções e este tipo de operação pode não ser desejado na aplicação ou biblioteca.

3.2.4.2.Listagem dos Fluxos de Exceção

Identificar com precisão os fluxos de uma aplicação é um desafio, pois o número de fluxos de uma aplicação cresce exponencialmente relativo ao seu tamanho. De tal forma que a simples listagem de cada fluxo de exceção demandaria espaço de armazenamento e tempo de execução exponenciais. Sendo assim, utilizamos, para as listagens, a mesma estratégia utilizada para percorrer a aplicação, ou seja, cada método dos fluxos serão impressos até encontrar um método que já foi impresso, por exemplo: sejam os fluxos de exceção fluxo 1: `A` → `B` → `C` e fluxo2 : `F` → `D` → `B` → `C`, e que uma exceção ocorre no método `C` , e supondo a ordem de impressão seja fluxo 1 e em seguida o fluxo 2, a impressão se dará como mostrada na Figura 41.

```
Exceção: java.lang.Exception - Escapou

A {1}

B {2}

C {3}

Exceção: java.lang.Exception - Capturada Extata

F {1}

D {2}

B {3}
```

Figura 41 - Exemplo de impressão de fluxos de exceção da ferramenta

Desta forma, o método `C` não será impresso na listagem dos fluxos de exceção, pois já foi impresso no fluxo anterior.

3.2.5.Testes

A abordagem adotada para garantir a confiabilidade da ferramenta foi a construção de micro aplicativos para cada maneira com que os aspectos podem interceptar uma classe ou outro aspecto. Em cada exemplo é esperado um resultado que foi conferido manualmente a partir da análise dos arquivos de saída gerados pela ferramenta. Desta forma, foram construídos exemplos para os seguintes casos repetidos em cada um dos tipos de interceptador a que o JBoss AOP dá suporte, ou seja, `before`, `after`, `around`, `throwing`, `finally` e `interceptor`:

- **Aspecto – Aspecto** : Quando uma exceção é levantada num aspecto e capturada por outro. Para esta situação foram construídas duas aplicações: uma que captura qualquer exceção e outra que captura somente a exceção específica que foi lançada no aspecto;
- **Aspecto – Classe** : Quando uma exceção é levantada num aspecto e capturada por uma classe. Para esta situação foram construídas duas aplicações: uma na qual a classe captura qualquer exceção e outra

que captura somente a exceção específica que foi lançada no aspecto;

- **Classe – Aspecto** : Quando uma exceção é levantada numa classe e capturada por um aspecto. Para esta situação foram construídas duas aplicações: uma que na qual o aspecto captura qualquer exceção e outra que captura somente a exceção específica que foi lançada na classe;
- **Classe – Classe**: Quando uma exceção é levantada numa classe e capturada por outra classe. Para esta situação foram construídas duas aplicações: uma que na qual a classe captura qualquer exceção e outra que captura somente a exceção específica que foi lançada na classe;
- **Aspecto – Escapa** : Quando uma exceção é levantada num aspecto e não é capturada;
- **Classe – Escapa** : Quando uma exceção é levantada numa classe e não é capturada;

Neste trabalho, foram criados 60 (sessenta) aplicativos para os testes, sendo eles uma combinação entre os tipos de advices e o tipo de captura.

3.3.Resumo

Nessa seção, foi apresentada em detalhes a implementação da ferramenta abordada nesta dissertação, mostrando a arquitetura, as estratégias e as técnicas utilizadas em sua construção. Portanto, foi apresentado que a ferramenta SAFE foi modificada para analisar aplicações JBoss AOP sendo esta nova versão chamada de SAFE-JBossAOP.

Quando o objetivo é analisar estaticamente as aplicações JBoss AOP, é possível que se faça necessário pré-processar a aplicação, ou seja, transformar a aplicação numa estrutura de arquivos e diretórios que permitam a análise estática. A decisão de pré-processá-la é baseada em alguns fatores como a dúvida se o weaving ocorre em tempo de carregamento ou se a forma de empacotamento da aplicação é WAR, EAR, pois, se qualquer uma destas condições ocorrer, far-se-á necessário o pré-processamento.

A ferramenta SAFE-JBossAOP não constrói o grafo de chamadas e sim percorre a aplicação a partir dos métodos inicializadores de fluxo - utilizando a classe `ForwardAnalysis` e a representação intermediária Jimple gerada pelo framework SOOT – buscando cláusulas que lancem exceção. Uma vez encontrada a cláusula, o caminho percorrido é analisado deste ponto até o método inicializador de fluxo, buscando blocos que capturem a exceção ou até que a mesma escape do programa. Entretanto, para percorrer a aplicação, pode ser utilizada alguma das estratégias fornecidas pela ferramenta e o objetivo destas é determinar a abstração de contexto e quais serão os métodos a serem analisados a partir de um método dado. A ferramenta utilizada ainda provê duas estratégias que são: (i) CHA, que utiliza como abstração de contexto a assinatura do método e analisa todos os métodos de uma hierarquia (ver Seção 3.2.2.1); e (ii) Contexto Objeto, que utiliza como abstração de contexto o objeto que recebe a invocação do método analisando, desta forma, somente o método referente ao tipo do objeto (ver Seção 3.2.2.2). No entanto, ambas as estratégias utilizam a técnica point-to, que tem por objetivo determinar o valor das variáveis do programa analisado (ANDERSEN, 1994).

Com o intuito de fornecer informações mais relevantes a respeito do tratamento de exceção, cada fluxo excepcional é rotulado a partir das seguintes designações: (i) a exceção escapou, se não houver nenhum bloco de captura para a exceção no caminho dela; (ii) a exceção foi capturada, se houver um bloco de captura para a exceção e a aplicação voltar ao estado normal; (iii) a exceção foi capturada e relançada, se houver um bloco de captura para a exceção e for realizado algum tratamento, mas, mesmo assim, a exceção for lançada novamente tal qual foi capturada; (iv) a exceção foi capturada e trocada, se houver um bloco de captura para a exceção e uma nova exceção for lançada sem levar nenhuma informação da capturada; e (v) a exceção foi capturada e encapsulada em outra, se existir um bloco de captura para a exceção e uma nova exceção for lançada levando informações da exceção capturada.