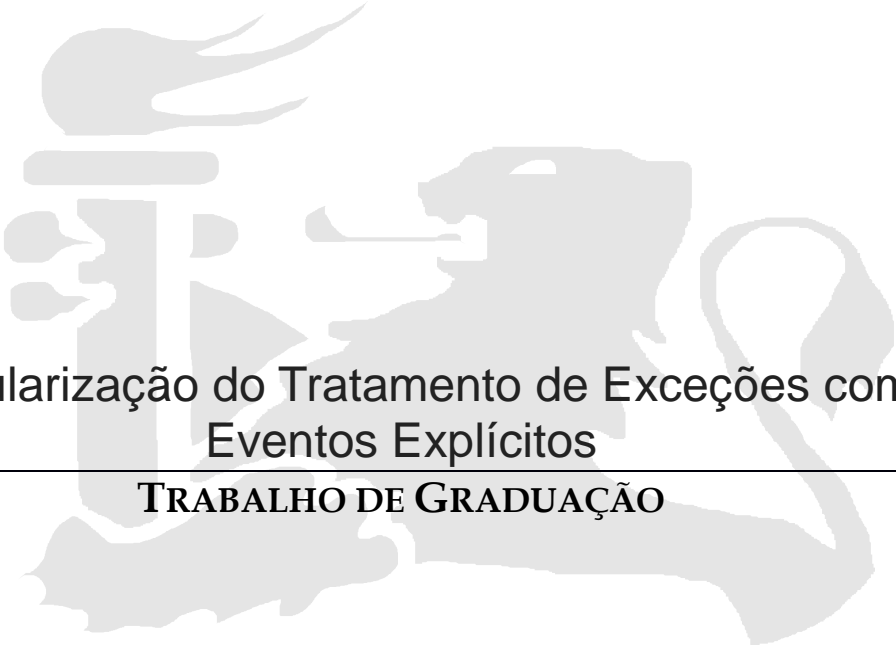




UNIVERSIDADE FEDERAL DE PERNAMBUCO  
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO  
CENTRO DE INFORMÁTICA  
2012.1

---



Modularização do Tratamento de Exceções com  
Eventos Explícitos

---

**TRABALHO DE GRADUAÇÃO**

**Aluno:** Irineu Martins de Lima Moura (imlm2@cin.ufpe.br)

**Orientador:** Fernando José Castor de Lima Filho (fjclf@cin.ufpe.br)

Recife, Julho de 2012.

Universidade Federal de Pernambuco

Centro de Informática

Irineu Martins de Lima Moura

**Modularização do Tratamento de Exceções com  
Eventos Explícitos**

Trabalho de Graduação apresentado ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

**Orientador:** Fernando José Castor de Lima Filho

Recife, Julho de 2012

“Suba o primeiro degrau com fé. Não é necessário que você veja toda escada. Apenas dê o primeiro passo”.

Martin Luther King, Jr

## Resumo

Linguagens orientadas a aspectos promovem a separação de interesses possibilitando a modularização de interesses transversais. Entre essas linguagens podemos definir duas categorias que representam como elas expõem os pontos de junção de um programa, isto é, pontos ou eventos onde os interesses transversais e não transversais interagem. Algumas delas possuem pontos de junção predeterminados (implícitos), enquanto que outras requerem que os desenvolvedores explicitem onde esses interesses podem interagir.

Um exemplo característico de interesse transversal que pode ser modularizado com linguagens orientadas a aspectos é o tratamento de exceções de um programa. Vários trabalhos já foram realizados para estudar a adequação de linguagens orientadas a aspectos com pontos de junção implícitos, como *AspectJ*, para a modularização de tal interesse [Lippert e Lopes 2000; Castor Filho et al. 2009; Taveira et al. 2009]. Este trabalho, por sua vez, intenciona realizar um estudo inicial envolvendo uma linguagem orientada a aspectos com eventos explícitos e o tratamento de exceções de um programa existente. Pretende-se, aqui, apresentar a linguagem *Ptolemy* [Rajan e Leavens 2008; 9] e analisar as capacidades e limitações dessa linguagem para modularizar o tratamento de exceções de um sistema.

## **Abstract**

Aspect oriented languages promote separation of concerns by enabling the modularization of crosscutting concerns. Among these languages two categories can be devised that represent how they expose the joinpoints of a program (points or events in the execution of a program where crosscutting concerns must interact with non-crosscutting ones). Some of them expose a fixed (implicit) set of joinpoints, while others require that developers explicitly indicate where both types of concern are allowed to interact.

A distinctive example of a crosscutting concern that can be modularized with aspect oriented languages is the exception handling of a program. Several studies have been conducted to understand the adequacy of implicit jointpoint aspect oriented languages, like AspectJ, for the modularization of such concern [Lippert e Lopes 2000; Castor Filho et al. 2009; Taveira et al. 2009]. This work, in turn, intends to perform an initial assessment regarding an aspect oriented language with explicit events and the exception handling of an existing program. We intend to present the *Ptolemy* language [Rajan and Leavens 2008; 9] as well as its capabilities and limitations for the exception handling of a system.

## **Agradecimentos**

O apoio de várias pessoas foi fundamental para que eu conseguisse chegar até o final desta graduação e, por isso, lhes sou extremamente grato.

Para não perder o costume e reconhecendo de fato a importância dela, gostaria de agradecer imensamente a minha família, que sempre esteve presente em todas as minhas necessidades e me apoiou na decisão de estudar em Recife. Sobretudo, gostaria de agradecer as minhas queridas mãe e avó, sem as quais o sonho de morar e estudar fora de casa não teria se realizado. Gostaria de agradecer especialmente também a meu tio Júnior e sua família, que me acolheram em Recife e facilitaram o processo de adaptação.

Expresso minha enorme gratidão aos novos amigos que fiz aqui. Aos “nerds” que me ensinaram, mais uma vez, o significado da palavra amizade. Aos “nerds” que me fizeram rir e me zoaram. E aos não tão “nerds” que sempre me motivaram quando eu estive em dúvida. Não esquecendo, é claro, dos amigos que ficaram em Aracaju, os quais, tenho certeza, sempre torceram por mim.

Agradeço também aos colegas, professores e funcionários do Centro de Informática que contribuíram para esse início da minha formação. Em especial, agradeço ao meu professor e orientador, Fernando Castor, por sua (enorme) paciência e dedicação desde que me tornei seu aluno de IC, por quem tenho profunda admiração e com quem espero trabalhar novamente um dia.

Por fim, gostaria de agradecer aos velhos e novos colegas e amigos do Projeto CIn/Motorola por sua cooperação e ensinamentos.

A todos, meus sinceros agradecimentos!

# Índice

|   |    |
|---|----|
| 1. Introdução.....  | 1  |
| 1.1 Contexto.....   | 1  |
| 1.2 Motivação.....  | 1  |
| 1.3 Objetivos .....   | 2  |
| 1.4 Estrutura.....  | 2  |
| 2. Fundamentação.....   | 3  |
| 2.1 Exceções em <i>Java</i> .....                                       | 3  |
| O lançamento de exceções .....  | 3  |
| Tratadores de exceções (A construção <i>try..catch..finally</i> ) ..... | 3  |
| Propagação de exceções .....  | 5  |
| Escopo de variáveis.....  | 5  |
| Tipos e classificações de exceções .....                                | 5  |
| Interfaces de exceções .....  | 6  |
| 2.2 Programação orientada a aspectos (POA) e Eventos.....               | 6  |
| <i>AspectJ</i> .....  | 8  |
| 2.3 Trabalhos anteriores .....  | 9  |
| 2.4 <i>Ptolemy</i> .....  | 11 |
| Eventos em <i>Ptolemy</i> .....   | 12 |
| Anúncio de eventos .....  | 13 |
| Tratadores de Eventos.....  | 14 |
| Cadeia de tratadores e expressões <i>invoke</i> .....                   | 16 |
| 3. Extração do tratamento de exceções com <i>Ptolemy</i> .....          | 17 |
| 3.1 Descrição da abordagem geral para extração .....                    | 17 |
| 3.2 Limitações de <i>Ptolemy</i> .....                                  | 18 |
| 3.3 Classificação do código de tratamento .....                         | 19 |
| 3.4 Cenários de extração .....  | 23 |
| 3.5 <i>Ptolemy</i> e os mecanismos de tratamento de exceções .....      | 31 |

|                           |    |
|---------------------------|----|
| 4. Trabalhos futuros..... | 33 |
| 5. Conclusão.....         | 34 |
| 5. Referências.....       | 35 |



## Lista de Figuras

|   |    |
|---|----|
| Figura 1 - Exemplo de lançamento de uma exceção .....   | 3  |
| Figura 2 - Exemplo de utilização de um bloco <i>try..catch..finally</i> .....   | 4  |
| Figura 3 - Declaração de um tipo de evento .....  | 12 |
| Figura 4 - Anúncio de um evento simples .....   | 13 |
| Figura 5 - Anúncio de evento que sobrescreve um trecho de código .....  | 13 |
| Figura 6 - Método tratador de evento .....  | 15 |
| Figura 7 - Associação de um tipo de evento com um método tratador .....   | 15 |
| Figura 8 - Registro de uma instância para ser notificada sobre ocorrência de eventos                                  | 15 |
| Figura 9 - Guardando <i>invoke()</i> com <i>try..catch</i> .....  | 17 |
| Figura 10 - Tratamento de exceção trivial .....   | 18 |
| Figura 11 - Extração simples de tratamento de exceção trivial .....   | 18 |
| Figura 12 - Extração de <i>try..catch</i> emaranhado e com código não terminal (A) .....                              | 20 |
| Figura 13 - Extração de <i>try..catch</i> emaranhado e com código não terminal (B) .....                              | 20 |
| Figura 14 - Anúncio aninhado .....  | 21 |
| Figura 15 - Bloco <i>try</i> dependente de contexto (esquerda) e modificador de contexto (direita) .....              | 22 |
| Figura 16 - Região protegida parcialmente de retorno .....  | 23 |
| Figura 17 - Exemplo de tratador dependente de contexto e sua extração .....   | 25 |
| Figura 18 - Exemplo de região protegida modificadora de contexto com tratador dependente de contexto .....            | 26 |
| Figura 19 - Exemplo de tratador modificador de contexto .....   | 27 |
| Figura 20 - Exemplo de região protegida totalmente de retorno com tratador totalmente de retorno e sua extração ..... | 28 |
| Figura 21 - Exemplo de região protegida parcialmente de retorno .....   | 29 |
| Figura 22 - Exemplo de região protegida totalmente de retorno com tratador de mascaramento .....                      | 30 |
| Figura 23 - Exemplo de região protegida não de retorno com tratador totalmente de retorno .....                       | 30 |
| Figura 24 - Exemplos de tratador de terminação de laço (esquerda) e continuação de laço (direita) .....               | 31 |
| Figura 25 - Exemplos de região protegida de terminação de laço (esquerda) e continuação de laço (direita) .....       | 31 |
| Figura 26 - Bloco <i>announce</i> suprimindo verificação de exceções .....  | 32 |

## **Lista de Tabelas**

|  |    |
|--|----|
| Tabela 1 - Resumo dos cenários de considerados ..... | 24 |
|--|----|

# 1. Introdução

## 1.1 Contexto

No contexto de engenharia de software, interesses transversais é o nome dado aos interesses cuja implementação (representação) se espalha e/ou se entrelaça pelos artefatos de um sistema [11]. Exemplos de interesses transversais são a detecção e o tratamento de erros, controle de concorrência, *logging*, entre outros.

Abordagens tradicionais de modularização, como a Programação Orientada a Objetos, não dispõem de mecanismos adequados para separar interesses transversais dos não-transversais em pontos (ou eventos) de interesse em um programa. Esses pontos são justamente aqueles onde, em tempo de execução, interesses transversais e não-transversais precisam interagir [Masuhara e Kiczales 2003].

Entre as abordagens sugeridas para lidar com os interesses transversais destaca-se a Programação Orientada a Aspectos (POA) [Kiczales et al. 1997]. POA introduz o conceito de *aspectos*, unidades funcionais que representam os interesses transversais de um sistema. Aspectos podem ser compostos, ligados/desligados e reutilizados independentemente do sistema base e de outros aspectos. A linguagem orientada a aspectos mais amplamente estudada é *AspectJ* [4], uma extensão orientada a aspectos para Java. Os aspectos nesta linguagem podem ser vistos como uma extensão das classes usuais de Java com a adição de (1) pontos de corte, que são descrições de pontos de junção ou conjuntos de pontos de junção de um programa, e (2) adendos, que podem ser vistos como pares consistindo em um gatilho e uma ação a ser executada quando um ponto de corte é encontrado.

Como uma variação das atuais de soluções orientadas a aspectos surgiram linguagens baseadas em eventos explícitos [Rajan et al. 2008]. Essas linguagens se inspiraram fortemente no padrão produtor/assinante [Eugster et al. 2003], no qual produtores publicam eventos específicos e assinantes se registram em uma infraestrutura intermediária para serem notificados sobre tais eventos.

## 1.2 Motivação

O tratamento de exceções pode também ser considerado um interesse transversal, e diversos estudos já discutiram estratégias para a modularização desse interesse [Castor et al. 2009] utilizando linguagens orientadas a aspectos convencionais e/ou avaliaram os efeitos dessa modularização em sistemas existentes, levando-se em conta atributos como tamanho e “plugabilidade” [Lippert e Lopes 2000],

acoplamento, coesão e separação textual [Castor et al. 2009], reusabilidade [Taveira et al. 2009] e confiabilidade [Cacho et al. 2009].

### 1.3 Objetivos

O objetivo principal deste trabalho é sugerir um conjunto de soluções para extrair o tratamento de exceções de um sistema utilizando uma linguagem baseada em eventos explícitos. As soluções propostas devem atender aos seguintes requisitos:

- Devem ser tão simples quanto possível.
- Devem manter o comportamento original da aplicação.
- Devem ser tão genéricas quanto possível

A intenção é que as soluções descritas possam ser automatizadas por alguma ferramenta de refatoramento e, portanto, não deve ser necessário ter algum conhecimento semântico sobre o código para realizá-las.

Para avaliar a viabilidade das soluções propostas para estruturar o tratamento de exceções usando eventos, planeja-se implementá-las em um sistema pré-existente.

### 1.4 Estrutura

Os outros capítulos deste relatório estão estruturados da seguinte maneira

- O capítulo 2 introduz os fundamentos necessários para a elaboração deste trabalho, nesse capítulo são discutidos conceitos básicos sobre tratamento de exceções em *Java*, programação orientada a aspectos, *Ptolemy* (uma linguagem orientada a eventos explícitos) e trabalhos relacionados.
- O capítulo 3 descreve como extrair o tratamento de exceções com *Ptolemy* e as limitações da linguagem para realizar a extração. Nele será detalhada e estendida a classificação do código de tratamento feita por Castor et al. [Castor et al. 2009], serão abordados os possíveis cenários de extração e por fim serão feitas algumas considerações sobre a linguagem.
- O quarto capítulo abordará possíveis trabalhos futuros e no quinto serão feitas as considerações finais sobre este trabalho.

## 2. Fundamentação

Este capítulo tem como objetivo introduzir conceitos, definições e terminologia geral sobre temas que serão abordados neste trabalho.

### 2.1 Exceções em *Java*<sup>1</sup>

O mecanismo de tratamento de exceções em *Java* [1] segue o modelo geral descrito por Garcia et al. [Garcia et al. 2001]. Eles permitem que situações anormais (excepcionais) na execução de um programa sejam indicadas pelo lançamento de uma exceção e possuem construções para delimitar sintaticamente regiões protegidas onde exceções lançadas ou propagadas podem ser capturadas e tratadas adequadamente.

#### O lançamento de exceções

A execução de uma instrução pela máquina virtual *Java* pode resultar no levantamento de uma exceção implicitamente ou explicitamente. Exceções lançadas implicitamente podem ser o resultado de algum erro de programação, como o acesso a um índice fora dos limites de um vetor, ou podem ser ocasionados pela incapacidade da máquina virtual em executar a instrução requisitada, por exemplo, ao tentar alocar memória quando os recursos do sistema estão exauridos. Adicionalmente, *Java* permite o lançamento explícito de exceções através do comando *throw*. Esse comando aceita uma expressão que deve ser avaliada em um tipo de exceção. O lançamento de exceções explícitas permite que desenvolvedores indiquem aos clientes de um método a incapacidade do mesmo em realizar a operação desejada. A figura abaixo exemplifica um uso do lançamento explícito de exceções com o comando *throw*.

```
01 if (! request.isAuthenticated()) {  
02     throw new InvalidSessionException();  
03 }
```

Figura 1 - Exemplo de lançamento de uma exceção

#### Tratadores de exceções (A construção *try..catch..finally*)

*Java* possui mecanismos específicos para tratar situações excepcionais, as construções *try..catch..finally*. Essas construções permitem a separação textual entre fluxo normal e o fluxo excepcional do programa.

---

<sup>1</sup> A implementação mais recente da linguagem, *Java* 1.7, que está disponível desde 7 de junho de 2011 introduziu novos tipos de regiões protegidas e de tratadores. A semântica descrita nesta seção, para fins deste projeto, leva em consideração somente os mecanismos disponíveis em versões anteriores da linguagem.

Um bloco *try* delimita uma região protegida ou contexto de tratamento [Garcia et al. 2001] no qual exceções levantadas durante a execução de uma instrução podem ser capturadas e tratadas por uma cláusula *catch* anexada.

Cláusulas *catch* especificam tipos de exceções que podem ser capturadas em um bloco *try* e estão associadas a um bloco de código que permite a definição de ações a serem tomadas caso uma das exceções especificadas seja levantada dentro do bloco. Cada cláusula *catch* deve especificar exatamente um<sup>1</sup> tipo de exceção e mais de uma cláusula pode estar associada a um determinado bloco *try*. Cláusulas que definem tipos de exceções mais específicas devem, obrigatoriamente, preceder cláusulas com exceções mais genéricas. Além de especificar tipos de exceções capturáveis, elas expõem também a instância da exceção capturada para o bloco tratador.

Além de cláusulas *catch*, *Java* permite também que finalizadores, ou blocos *finally*, sejam anexados a regiões protegidas. Finalizadores são blocos de código que sempre são executados independentemente de uma exceção ser lançada dentro de um bloco *try* e se essa exceção é tratada ou não. Finalizadores permitem que o desenvolvedor especifique ações obrigatórias, como o fechamento de um arquivo caso a leitura do mesmo seja bem sucedida ou não. No máximo um bloco *finally* pode estar anexado a um *try*.

Na Figura 2 podemos ver a sintaxe completa de um bloco *try..catch..finally*.

```
01 public String lerArquivo(String caminho) {
02     FileReader leitor = null;
03     StringBuffer buffer = new StringBuffer();
04     try {
05         leitor = new FileReader(caminho);
06         char lido;
07         while((lido = (char) leitor.read()) != -1) {
08             buffer.append(lido);
09         }
10     } catch (FileNotFoundException e) {
11         // Tratando...
12     } catch (IOException e) {
13         // Tratando...
14     } finally {
15         // Assegurando liberação do recurso
16         fechar(leitor);
17     }
18     return buffer.toString();
19 }
```

**Figura 2 - Exemplo de utilização de um bloco *try..catch..finally***

Quando uma exceção é levantada dentro de uma região protegida, o ambiente de execução irá procurar um tratador adequado na ordem especificada pelas cláusulas *catch* anexadas. Caso um tratador apropriado seja encontrado, ocorrerá um desvio no fluxo de controle da instrução problemática para o bloco de tratamento escolhido. Caso contrário, a exceção escapará o bloco *try* e será propagada para o escopo mais externo. Se o bloco de código de uma cláusula *catch* lançar uma exceção essa nova exceção escapará o bloco *catch* e será propagada para o escopo mais externo. Após a execução normal do bloco do tratador, o fluxo de controle continua a partir da primeira instrução que segue a construção *try..catch..finally*, e as instruções remanescentes do bloco *try* não são executadas. Adicionalmente, um bloco *finally* pode estar anexado a um bloco *try* junto a zero ou mais cláusulas *catch*. Nesse caso, como descrito anteriormente, o bloco finalizador sempre será executado após o bloco *try* ou, possivelmente, após um bloco *catch*, independentemente de uma exceção ser levantada dentro da região protegida ou do bloco tratador.

### **Propagação de exceções**

Quando uma exceção é levantada dentro uma região protegida e não há um tratador apropriado para mesma, a exceção escapa o bloco *try* e é propagada para o escopo mais externo. Se o escopo externo for o bloco de um método, a exceção acaba escapando o método e iniciando uma cadeia de propagação, ou seja, a exceção é propagada para os clientes do método em questão, que por sua vez devem tratá-la ou também continuar propagando a mesma. Caso nenhum método na cadeia de propagação consiga tratar a exceção levantada, então a thread de execução na qual a exceção foi lançada é terminada. Exceções levantadas fora de regiões protegidas automaticamente escapam o escopo atual.

### **Escopo de variáveis**

O escopo de variáveis em construções *try..catch..finally* obedece às regras usuais de escopo da linguagem. Variáveis definidas antes e num escopo léxico mais externo a uma construção *try..catch..finally* são acessíveis dentro dos blocos *try*, *catch* ou *finally*. Variáveis definidas dentro de regiões protegidas não são acessíveis aos blocos *catch* e *finally* e vice-versa, isto é, variáveis definidas em blocos *catch* ou *finally* não são acessíveis ao bloco *try* ou entre si.

### **Tipos e classificações de exceções**

Todo tipo de exceção em *Java* é, obrigatoriamente, um herdeiro direto da classe *java.lang.Throwable* (*Throwable*) ou de uma de suas subclasses. Por serem

instâncias de um tipo, elas, naturalmente, obedecem às várias regras do sistema de tipos da linguagem, por exemplo, uma cláusula `catch` que especifica uma exceção do tipo *Exception* pode capturar qualquer instância de *Exception* ou de uma de suas subclasses. Elas podem também, através de seus atributos ou métodos, expor informações contextuais para o código que porventura irá tratá-las.

Exceções em *Java* podem ser classificadas em dois<sup>2</sup> grupos, exceções checadas ou exceções não checadas. Exceções checadas são exceções que devem ser explicitamente tratadas ou, caso contrário, devem fazer parte da interface de exceções do método de onde estão sendo propagadas. O seu nome se deve ao fato de que o compilador *Java* verifica (ou *checa*) se tais exceções estão sendo tratadas ou se estão sendo devidamente identificadas na interface de exceções do método. Todo tipo de exceção filha de *Throwable* que não é uma subclasse de *java.lang.Error* (*Error*) ou *java.lang.RuntimeException* (*RuntimeException*) é considerada uma exceção checada pelo compilador. Inversamente, exceções não checadas são exceções que podem ser lançadas ou propagadas por um método, mas não são verificadas pelo compilador. Toda instância de *RuntimeException*, *Error* ou de uma de suas subclasses é uma exceção não checada.

### Interfaces de exceções

Métodos em *Java* podem conter, além da sua assinatura usual, uma cláusula *throws* e uma lista de exceções que indica quais exceções podem ser propagadas durante a execução desses métodos. Essa cláusula e a lista de exceções formam a interface de exceções de um método. Essas interfaces são utilizadas primariamente para indicar as potenciais exceções checadas<sup>3</sup> de um método, mas a linguagem permite também a identificação de exceções não checadas na cláusula *throws*, embora tais exceções ainda assim não sejam verificadas pelo compilador. Métodos que enumeram em suas interfaces uma exceção do tipo *T* podem propagar instâncias de *T* ou de qualquer subclasse de *T*.

## 2.2 Programação orientada a aspectos (POA) e Eventos

A orientação a aspectos [Kiczales et al. 1997] surgiu como uma evolução da orientação a objetos a partir da necessidade de transpor certas limitações impostas

---

<sup>2</sup> Exceções em *Java* também podem ser classificadas em síncronas ou assíncronas, embora tal distinção não seja relevante para este trabalho.

<sup>3</sup> A verificação de exceções checadas e de interfaces de exceções é feita estritamente pelo compilador *Java*. A nível de *bytecode*, a máquina virtual *Java* não verifica esse tipo de semântica [2].



pelo paradigma OO. Sistemas orientados a objetos tradicionais nem sempre podem ser decompostos de maneira que possam capturar adequadamente todos os possíveis interesses do sistema. Na maioria das vezes, sistemas OO sofrem da “tirania da decomposição dominante” [Tarr et al. 1999], isto é, a decomposição dominante do sistema não permite que certos interesses sejam realizados de maneira independente ou modular. Os interesses que não se encaixam nessa decomposição são comumente chamados de interesses transversais, pois suas implementações acabam se espalhando por vários módulos e/ou se entrelaçando com as implementações de outros interesses. Exemplos de tais interesses são *logging*, segurança, concorrência, tratamento de exceções, entre outros.

Linguagens orientadas a aspectos tradicionais permitem modularizar interesses transversais e possibilitam que esses interesses interajam com os interesses não transversais, ou código base. Essas linguagens estendem o paradigma OO com o conceito de aspectos [Kiczales et al. 1997], entidades que realizam os interesses transversais de um sistema. Em algumas linguagens, a separação entre a implementação de interesses transversais e código base é assimétrica, o que significa que (1) o código que implementa os interesses transversais possui meios para alterar o comportamento do código base, mas a premissa inversa não é verdadeira, e que (2) existe uma distinção clara entre os elementos que implementam interesses transversais (aspectos) e o código base (classes). Por outro lado, algumas linguagens, como *Ptolemy* e *Eos-U* [Rajan e Sullivan 2005], são simétricas. Neste caso, a linguagem não faz uma diferenciação explícita entre os elementos de código que implementam interesses transversais e os que implementam o código base. Aspectos funcionam, portanto, como abstrações que encapsulam os interesses transversais do sistema e que interagem com o código base para a realização de tais interesses. Interesses que outrora estariam espalhados pelo sistema agora estão modularizados em uma única unidade funcional, o aspecto. Deste modo, um dos principais objetivos da orientação a aspectos é de promover a separação de interesses. Essa separação deve facilitar a manutenção e a evolução tanto do sistema base quanto da implementação dos interesses transversais.

Linguagens orientadas a aspectos possuem construções que permitem a descrição de pontos ou eventos de interesse no sistema nos quais os aspectos devem interagir com o código base. Tais eventos podem ser expostos implicitamente pela linguagem ou explicitamente pelo código base. Linguagens AO com eventos implícitos permitem que o desenvolvedor do aspecto especifique e referencie certos tipos de eventos predeterminados que podem ocorrer durante a execução do programa, não há

necessidade de modificações no código base, ou seja, o desenvolvedor do código base é indiferente (do inglês: *oblivious* [Filman e Friedman 2000]) à implementação dos aspectos. Em linguagens com eventos explícitos, o próprio código base expõe os eventos de interesse para os aspectos envolvidos e este deve ser anotado (modificado) com certas informações para indicar a ocorrência de um determinado evento. Essas linguagens se afastam um pouco da noção de indiferença total por parte do desenvolvedor do código base em relação aos aspectos já que o mesmo agora deve ter algum conhecimento *a priori* sobre os pontos no programa que podem ser modificados pelos aspectos. É importante salientar, entretanto, que existem estudos que questionam essa necessidade de indiferença total por parte do código base como [Hoffman e Eugster 2008, 2009; Castor et al. 2009].

### **AspectJ**

Um dos mais conhecidos exemplos de linguagem orientada a aspectos com eventos implícitos é *AspectJ* [4]. *AspectJ* é uma extensão orientada a aspectos da linguagem *Java*. Nela os eventos implícitos são chamados de pontos de junção e especificam pontos bem definidos no fluxo de um programa, como a chamada a um método, a execução de um tratador de exceções ou acesso a um atributo de uma instância de uma classe. Nessa linguagem, aspectos são entidades similares a classes e possuem construções especiais chamadas de descritores de pontos de corte (DPCs) e adendos.

Descritores de pontos de corte servem para descrever um conjunto de pontos de junção e expor informações contextuais dos pontos de junção para os adendos. DPCs também podem ser compostos utilizando operadores booleanos para formar outros DPCs mais complexos, além disso, eles também podem receber nomes para que possam ser reutilizados. O conjunto de DPCs definidos pela linguagem limita os tipos de pontos de junção sobre os quais os aspectos podem atuar.

Adendos são construções que realizam de fato os interesses transversais, eles são compostos por um DPC e um bloco de código que define ações a serem tomadas quando o DPC for encontrado durante a execução do programa. A interface do adendo e seu DPC definem que informações contextuais (variáveis) estarão disponíveis para o bloco de código. A ordem execução do bloco de código em relação ao código base é definida pelo tipo de adendo, podendo ser executado antes, depois ou ao invés do código base.

## 2.3 Trabalhos anteriores

Vários estudos já avaliaram a adequação de linguagens orientadas a aspectos para a modularização e reuso do tratamento de exceções e propuseram sugestões ou ferramentas para auxiliar na extração desse interesse transversal para aspectos.

Lippert e Lopes [Lippert e Lopes 2000] utilizaram *AspectJ* para extrair para aspectos as especificações de contratos e o tratamento de exceções de um framework existente e avaliaram quantitativamente e qualitativamente os ganhos com essa extração. O estudo mostrou que, para o framework alvo, a extração resultou em um ganho com reuso do código de tratamento e em uma diminuição de até quatro vezes no código desse interesse. Os autores também concluíram que *AspectJ* pode prover um melhor suporte para diferentes configurações (“pugabilidade”) do tratamento de exceções e é mais tolerante a modificações na especificação do comportamento excepcional.

Castor et al. [Castor et al. 2009] realizaram um estudo aprofundado sobre modularização e reuso do tratamento de exceções com aspectos. Como em outros estudos, este também comparou quantitativamente e qualitativamente versões em *AspectJ* e utilizando OO de vários sistemas para avaliar a adequação de *AspectJ* para lidar com o código de tratamento de exceções. O estudo propôs também soluções para extrair tratadores de exceções para *AspectJ*. Para propor as soluções o estudo classificou o código de tratamento de exceções levando em consideração os possíveis fatores que poderiam complicar a extração em *AspectJ*. A partir dessa classificação, vários cenários foram definidos e a eles foram atribuídas pontuações determinando a facilidade com a qual o código de tratamento de exceção poderia ser extraído para um aspecto. A classificação e os cenários do estudo também serão utilizados neste trabalho e serão detalhados no Capítulo 3.

O estudo de Taveira et al. [Taveira et al. 2009] tentou avaliar os ganhos reais com reuso ao extrair o código de tratamento de exceções para aspectos. Nesse o código de tratamento de exceções de três sistemas de média/larga escala utilizando duas abordagens, uma em *AspectJ* e outra utilizando classes OO usuais. Para as versões refatoradas, todos os tratadores de exceções foram examinados e foram combinados tratadores idênticos ou cuja combinação não modificaria o comportamento do programa. Foram então comparadas as três versões (a original e as duas refatoradas) utilizando métricas de tamanho e de difusão de interesses. Concluiu-se que a modularização do tratamento de exceções para aspectos de fato promove um melhor reuso do código de tratamento, embora, contradizendo estudos anteriores, esse reuso não

necessariamente implique em uma diminuição na quantidade de linhas de código do sistema como um todo.

Para tentar resolver ou amenizar dois importante problemas com tratamento de exceções em linguagens tradicionais, a existência de fluxos implícitos de exceções e a incapacidade de reutilizar tratadores, Cacho et al.[Cacho et al. 2008] apresentaram a linguagem *EJFlow*. *EJFlow* é uma extensão de *AspectJ* que permite ao desenvolvedor explicitar os fluxos de exceções de um programa através da especificação de canais de propagação e anexar tratadores “plugáveis” a esses canais. *EJFlow* objetiva melhorar a manutenibilidade de sistemas, pois facilita o entendimento do fluxo de exceções, que está explícito e localizado (em um único aspecto), e promover o reuso do código de tratamento através do tratadores “plugáveis”. Para avaliar se esse modelo de fluxos explícitos de exceções poderia melhorar a manutenibilidade e robustez de um sistema de software [Cacho et al. 2009] conduziram um estudo exploratório utilizando *EJFlow*. Esse estudo comparou a performance de três grupos de estudantes para realizar tarefas relacionadas ao tratamento de exceções de dois sistemas em *AspectJ*, *EJFlow* e utilizando orientação a objetos tradicional. Essa comparação levou em consideração indicadores de usabilidade de software como tempo para executar as tarefas, número de exceções não capturadas e número de respostas incorretas. Analisando os indicadores obtidos, os autores concluíram que um modelo de fluxos explícitos de exceções pode de fato aumentar a confiabilidade do sistema.

Hoffman e Eugster [Hoffman e Eugster 2009] introduziram pontos de junção explícitos em *AspectJ* e definiram o conceito de programação orientada a aspectos cooperativa. Pontos de junção explícitos (PJE) permitem que o código base, agora ciente da existência de aspectos, interaja com os mesmos através de novos pontos de junção definidos explicitamente pelos desenvolvedores. PJE facilitam a troca de informações entre o código base e os aspectos e permitem que trechos arbitrários de código sejam controlados pelos aspectos. O termo programação orientada a aspectos cooperativa se deriva, portanto, do fato de que o código base deve cooperar ativamente (e explicitamente) com os aspectos para realizar todos os interesses do sistema. Os objetivos dos PJE são bastante similares aos eventos da linguagem utilizada neste trabalho, isto é, o de funcionar como uma interface entre o código base e os aspectos, desacoplando-os e facilitando a comunicação entre ambos. Como os outros estudos, este também realizou uma análise comparativa com *AspectJ* (tradicional), focada principalmente na modularização do tratamento de exceções, para avaliar a extensibilidade de programas modularizados utilizando PJE. A análise

concluiu que os PJE's melhoraram a extensibilidade do sistema estudado quando comparados a *AspectJ*.

No trabalho realizado por Coelho et al. [Coelho et al. 2011] os autores apresentaram uma abordagem auxiliada por uma ferramenta de análise estática para facilitar o entendimento e a verificação do código de tratamento de exceções de programas orientados a aspectos. O estudo apresentou também diretrizes para o tratamento de exceções lançadas por aspectos, a fim de garantir a robustez do código base na ocorrência desses eventos excepcionais. A abordagem sugerida permite que desenvolvedores especifiquem contratos de tratamento de exceções e que esses contratos sejam verificados pela ferramenta de análise estática *SAFE*. Essa ferramenta é capaz de computar as interfaces de exceções dos adendos de *AspectJ* e de descobrir os caminhos de exceções de um sistema AO com o intuito de verificar se os contratos especificados pelos desenvolvedores são respeitados. Os autores também avaliaram a abordagem sugerida aplicando-a a três sistemas AO e concluíram que a ferramenta *SAFE* foi precisa o suficiente para descobrir caminhos de exceções relevantes, e que a abordagem é viável e útil para verificar a confiabilidade do código de tratamento de exceções dos sistemas estudados.

## 2.4 Ptolemy

As soluções descritas para a extração do tratamento de exceções deste trabalho foram desenvolvidas utilizando a linguagem *Ptolemy* [Rajan e Leavens, 2008; 9]. *Ptolemy* é uma linguagem orientada a eventos bastante similar a Java e que tenta assimilar as vantagens de linguagens orientadas a aspectos tradicionais e de linguagens baseadas no paradigma de invocação implícita.

A principal característica de *Ptolemy* que a diferencia de linguagens OO ou AO tradicionais é seu modelo de eventos explícitos, tipificados e que podem ser quantificados. Eventos são explícitos porque, para expor sua ocorrência aos aspectos, o código base deve explicitamente referenciá-los (anunciá-los). São tipificados porque todo evento possui um tipo, que é definido pelo desenvolvedor. E podem ser quantificados, pois a linguagem permite que os tratadores de eventos (aspectos) expressem seu interesse em todas as ocorrências (instâncias) de um determinado tipo de evento.

A linguagem se aproxima bastante do modelo produtor/assinante [Eugster et al. 2003]. Nela é possível que classes produtoras anunciem ou publiquem eventos e o

ambiente de execução se encarrega de entregar tais eventos aos tratadores interessados. É importante observar, entretanto, que a mesma não dá suporte completo a este paradigma de interação. De fato, apesar de existir um desacoplamento espacial em relação às entidades participantes, já que o código OO que anuncia um evento não tem conhecimento de quantos e quais são os tratadores daquele evento e vice-versa, não há desacoplamento temporal e de sincronização, pois ambas as partes devem estar ativas e anunciar e/ou tratar eventos bloqueia tanto o produtor quanto o tratador (já que eventos são anunciados e tratados em uma mesma<sup>4</sup> thread de execução).

### Eventos em *Ptolemy*

Eventos em *Ptolemy* são abstrações que funcionam como interfaces entre o código base e o código dos aspectos. A declaração de um tipo de evento possui três elementos:

1. O nome do tipo do evento. Esse nome será usado tanto pelo tratador, para declarar seu interesse nesse tipo de evento, quanto pelo código base, para anunciar instâncias desse evento.
2. Uma lista de variáveis de contexto. Essas variáveis permitem que o código base disponibilize informações contextuais para os tratadores de eventos
3. Um tipo de retorno. O anúncio de eventos pode retornar<sup>5</sup> valores para o código OO ou para tratadores.

Como pode ser observado na Figura 3, um evento é declarado utilizando a palavra reservada *event* e pode conter, além dos elementos já citados, um modificador de acesso. O exemplo declara um tipo de evento de visibilidade pública chamado *ProcessEvent* que expõe para os tratadores a linha de comando que será executada, através da variável de contexto *command*, e retorna uma instância de *Process* representando o processo iniciado.

```
01 public Process event ProcessEvent {  
02 String command;  
03 }
```

**Figura 3 - Declaração de um tipo de evento**

<sup>4</sup> Esta informação foi obtida experimentalmente e confirmada pelos criadores de *Ptolemy* como sendo uma decisão de design da linguagem.

<sup>5</sup> Na versão do compilador utilizada neste estudo essa funcionalidade não estava funcionando corretamente. Não era possível para o código que anuncia o evento obter os resultados da execução da expressão *announce*.

Por funcionarem como interfaces, tipos de eventos possibilitam a compilação independente do código base e dos tratadores de eventos (aspectos).

## Anúncio de eventos

Eventos em *Ptolemy* podem ser publicados ou anunciados utilizando a expressão *announce*. Uma expressão *announce* deve ser acompanhada do tipo do evento que está sendo publicado, como pode ser visto na figura abaixo.

```
01 public void spawnProcess() throws ... {
02     Process proc = new ProcessBuilder("...").start();
03     proc.waitFor();
04     announce ProcessFinishedEvent(proc);
05 }
```

Figura 4 - Anúncio de um evento simples

A expressão *announce* permite associar valores locais às variáveis de contexto do evento que está sendo anunciado. Essa associação é feita passando os valores que devem ser expostos como argumentos do evento. A ordem dos tipos e a quantidade de argumentos devem obedecer à ordem e à quantidade de variáveis de contexto descritas na declaração do evento, isto é, o tipo do primeiro argumento deve ser compatível com o tipo da primeira variável de contexto, o segundo deve ser compatível com o tipo da segunda variável e assim sucessivamente. Na linha 4 do exemplo acima a variável local *proc* está sendo associada a uma possível variável de contexto do tipo *Process* definida durante a declaração do tipo *ProcessFinishedEvent*.

Uma característica distinta desse mecanismo de publicação de eventos é que ele permite que trechos de código sejam sobrescritos. Como será demonstrado na seção de tratadores, isso permite simular os vários tipos de adendos existentes em *AspectJ*. De fato, essa característica foi a base para o desenvolvimento das soluções propostas por este trabalho.

```
01 public Process spawnProcess(String cmd) {
02     Process process = null;
03     announce ProcessEvent(cmd) {
04         ProcessBuilder pb = new ProcessBuilder(cmd);
05         process = pb.start();
06         return process;
07     }
08     return process;
09 }
```

Figura 5 - Anúncio de evento que sobrescreve um trecho de código

O bloco de código sobrescrito é chamado de *closure* do evento e sua execução está condicionada ao comportamento dos tratadores, isto é, um determinado tratador pode decidir arbitrariamente que ela não deve mais ser executada. Como pode ser

observado no exemplo da Figura 5, o bloco de código de uma expressão *announce* pode conter um comando *return*, isso possibilita que a *closure* do evento retorne um valor para os tratadores ou para o código que anuncia o evento<sup>5</sup>.

A semântica do bloco de código do anúncio de um evento difere sensivelmente de outros blocos de código da linguagem. Apesar da leitura e da escrita de variáveis locais funcionarem da mesma maneira, o mesmo não pode ser considerado para algumas instruções de desvio de fluxo de controle. Como visto acima, o comando *return* se refere ao retorno do evento e não ao retorno do método em que se encontra. Além disso, blocos *announce* que estão contidos em laços não podem conter comandos *break* ou *continue* (a menos que esses comandos estejam dentro de laços contidos no próprio anúncio). De maneira simplificada, a execução do bloco de um evento funciona como uma chamada a um método que pode alterar valores locais do método cliente, mas não pode alterar seu fluxo de controle diretamente exceto, possivelmente, através do lançamento de uma exceção.

### **Tratadores de Eventos**

Classes que reagem a anúncio de eventos são chamadas de classes tratadoras. Essas classes são análogas aos aspectos em *AspectJ*, pois é nelas que está contido o código que implementa os interesses transversais. Por não distinguir aspectos de classes *Ptolemy* permite que haja simetria entre o sistema base e os tratadores. A definição do equivalente aos adendos possui duas partes: a implementação de um método com uma interface específica e a declaração de uma associação entre um tipo de evento e o método implementado.

Métodos que devem ser executados em resposta a eventos precisam definir uma interface específica. O tipo de retorno deve ser compatível com o tipo de retorno do evento que pretendem tratar, e a lista de parâmetros deve conter somente uma variável que deve ser do tipo do evento tratado. A figura abaixo apresenta um possível método tratador para o tipo de evento *ProcessEvent* definido na Figura 3:



```

01 public Process handle(ProcessEvent next) throws Throwable{
02     try {
03         Process proc = next.invoke();
04         proc.waitFor();
05         return proc;
06     } catch(IOException e){
07         ProcessBuilder builder = new ProcessBuilder(next.command());
08         Process proc = builder.start();
09         return proc;
10     }
11 }

```

**Figura 6 - Método tratador de evento**

A interface de exceções do método tratador pode conter exceções arbitrárias, mas devido a um detalhe de implementação quase sempre deverá conter o tipo *Throwable*.

Após implementar o método que tratará as instâncias de um evento, é necessário realizar uma associação explícita entre o tipo do evento e seu método tratador. Essa associação é feita utilizando a construção *when tipo\_da\_exceção do nome\_método\_tratador*, como pode ser visto no exemplo abaixo:

```

01 when ProcessEvent do handle;

```

**Figura 7 - Associação de um tipo de evento com um método tratador**

A associação do exemplo estabelece que quando uma instância do evento *ProcessEvent* ocorre a mesma deve ser tratada com o método *handle* definido na classe tratadora.

Para que as associações sejam ativadas, isto é, para que os métodos tratadores possam ser invocados durante o anúncio de um evento, é necessário que as instâncias das classes tratadoras se registrem no ambiente de execução da linguagem. Instâncias podem se registrar utilizando a expressão *register*, que recebe como parâmetro o objeto que será o alvo das chamadas para tratar possíveis eventos. O exemplo da Figura 8 mostra uma instância de *ExceptionHandler* se registrando durante a execução de seu construtor.

```

01 public ExceptionHandler() {
02     register(this);
03 }

```

**Figura 8 - Registro de uma instância para ser notificada sobre ocorrência de eventos**

Métodos tratadores recebem como parâmetro a *closure* do evento que está sendo anunciado. A *closure* permite que os tratadores acessem as variáveis de contexto do evento, como pode ser visto na linha sete do trecho de código da Figura 6 onde o tratador recupera o valor da linha de comando do processo através da chamada *next.command()*. Além disso, a *closure* possui o código necessário para

executar outros tratadores ou o bloco de código definido pela expressão *announce* do evento.

### **Cadeia de tratadores e expressões *invoke***

Métodos que tratam um mesmo tipo de evento formam uma cadeia de tratadores. Durante o anúncio de um evento, os tratadores são invocados na ordem em que foram registrados, embora esse comportamento possa ser alterado (pelos próprios tratadores). Um tratador pode transferir o controle para o próximo tratador através da expressão *invoke()*. A chamada a expressão *invoke()* executa o próximo tratador registrado, que por sua vez deve também chamar *invoke()* para executar o próximo tratador. Caso não haja mais tratadores nessa cadeia então o bloco de código da expressão *announce* é executado. Se um tratador omitir a chamada a *invoke()* então a cadeia é quebrada e os próximos tratadores e/ou o bloco do evento anunciado não serão executados. As expressões *invoke()* permitem que os tratadores definam ações a serem tomadas antes, depois ou no lugar de ações definidas no bloco *announce* (ou outros tratadores) muito similarmente aos tipos de adendos em *AspectJ*. O valor retornado por um expressão *invoke()* depende dos próximos tratadores executados ou do valor retornado pelo bloco do evento, mas é do tipo definido como tipo de retorno do evento que está sendo tratado. Na linha três da Figura 6 podemos ver a sintaxe de uma chamada a *invoke()* na execução do tratador *handle*.

Na implementação corrente da linguagem, tratadores que chamam *invoke()* devem listar *Throwable* em sua interface de exceções ou cobrir a chamada a *invoke()* com um bloco *try..catch* que capture *Throwable*. Isso se deve ao fato de que a execução de uma *closure* arbitrária pode lançar qualquer exceção.

### 3. Extração do tratamento de exceções com *Ptolemy*

Este capítulo descreve como extrair o tratamento de exceções de um sistema utilizando *Ptolemy* e detalha quais as possíveis limitações da linguagem para realizar esse processo. Primeiro será dada uma visão geral sobre a abordagem genérica utilizada para realizar uma extração. Em seguida, serão abordadas, de maneira abstrata, as principais limitações da linguagem para realizar a extração utilizando a abordagem descrita anteriormente. Para ilustrar melhor o processo de extração e suas limitações, as próximas duas seções serão utilizadas para classificar e descrever cenários de tratamento de exceções, detalhando como esses cenários podem ser extraídos ou porque eles não podem ser. Por fim, serão feitas algumas considerações sobre *Ptolemy* e os mecanismos de tratamento de exceções.

#### 3.1 Descrição da abordagem geral para extração

Como descrito na seção sobre *Ptolemy* do Capítulo 2, a linguagem permite que o anúncio de um evento sobrescreva um determinado bloco de código, chamado de *closure* do evento, e condiciona a execução desse bloco à chamada da expressão *invoke()* durante a execução dos tratadores de evento. Isso permite que os tratadores de evento possam controlar quando e como o bloco de código da *closure* é executado. Por exemplo, para tratar exceções levantadas durante a execução do bloco de código de um evento, um método tratador de eventos poderia simplesmente cobrir a chamada a *invoke()* com um bloco *try*, especificando cada uma das exceções desejadas nas cláusulas *catch* e, possivelmente, adicionando um finalizador.

```
01 try{
02     next.invoke();
03 } catch(Exception1 e1) {
04     // ...
05 } catch(Exception2 e2) {
06     // ...
07 } /* Possivelmente outras cláusulas
08     catch ou bloco finally */
```

Figura 9 - Guardando *invoke()* com *try..catch*

Essa capacidade de sobrescrever blocos de código e de guardá-los (com blocos *try*) indiretamente é o que forma a base para a abordagem genérica de extração do tratamento de exceções para tratadores de eventos.

Para extrair o tratamento utilizou-se a abordagem descrita inicialmente por Dyer et al. [Dyer et al. 2012]. Dada uma situação de tratamento de exceções trivial como a Figura 10, poderíamos extrair o tratamento criando um novo tipo evento *FacadeInitEvent*, transformando o bloco *try* em um bloco *announce* para o evento

*FacadeInitEvent* e movendo o tratamento de exceções para um tratador do evento anunciado, neste caso *FacadeExceptionHandler*. A Figura 11 mostra o resultado desse processo. Pode-se notar que caso uma *CommunicationException* seja levantada durante a execução da *closure* do evento, o fluxo de controle será transferido para tratador e, após a execução normal do mesmo, o fluxo continuará a partir da primeira<sup>6</sup> instrução que segue o bloco do anúncio.

```

01 void init(){
02     try {
03         initFacade();
04     }catch (CommunicationException e) {
05         e.printStackTrace();
06     }
07     // ...
08 }

```

**Figura 10 - Tratamento de exceção trivial**

|  |   |
|--|---|
| <pre> 01 public void event FacadeInitEvent{} 01 void init(){ 02     announce FacadeInitEvent() { 03         initFacade(); 04     } 05     // ... 06 } </pre> | <pre> 01 public class FacadeExceptionHandler{ 02     public FacadeExceptionHandler() { 03         register(this); 04     } 05     void handle(FacadeInitEvent next){ 06         try{ 07             next.invoke(); 08         }catch(CommunicationException e){ 09             e.printStackTrace(); 10         } 11     } 12 } 13 when FacadeInitEvent do handle; 14 } </pre> |
|--|---|

**Figura 11 - Extração simples de tratamento de exceção trivial**

### 3.2 Limitações de *Ptolemy*

Apesar de a abordagem geral ser aplicável em muitos casos, existem algumas limitações da linguagem que podem impossibilitar a extração do código de tratamento de exceções em vários outros cenários. Podemos dizer, de uma maneira abstrata, que os problemas encontrados neste trabalho se resumem às diferenças nos contextos em que são executados o código de tratamento extraído (executado no contexto do método tratador de eventos), a *closure* do evento e o código do método original, alvo da extração.

Essas diferenças de contexto podem resultar em problemas para manter uma equivalência entre o fluxo de controle (excepcional e/ou não excepcional) do método original e o fluxo de controle após a extração. Caso um bloco *try* ou um bloco *catch* possuam uma instrução de retorno, após a extração, essas instruções não mais se

---

<sup>6</sup> Caso haja outros tratadores de evento registrados para aquele tipo de evento, o fluxo de controle após a execução do bloco tratador de exceções deve continuar a partir da execução desses tratadores (de eventos) antes de prosseguir com a execução do método alvo da extração.

referirão ao método (contexto) original. Existem complicações ainda maiores quando a região protegida possui um comando de retorno e o tratador de exceções não, ou vice-versa, já que teremos um contexto que pode retornar e outro não. Além disso, comandos de controle de laço como *break* e *continue* também são problemáticos, pois eles não podem se referir a laços em um contexto diferente.

Outros problemas resultantes dessa diferença de contexto de execução estão relacionados às interações com variáveis locais. Excetuando-se a semântica dos blocos *announce*, o contexto de um método não pode escrever uma variável local do contexto de outro método. Escritas em variáveis locais dentro da *closure* de um evento não são visíveis para os blocos tratadores extraídos.

### 3.3 Classificação do código de tratamento

O trabalho realizado por Castor et al. [Castor et al. 2009] classificou o código de tratamento de exceções baseado em fatores que poderiam impactar a extração para aspectos com *AspectJ*. Esta seção descreve a classificação original e como ela se relaciona com *Ptolemy*. Adicionalmente, este trabalho estende o original com a classificação de outros fatores que também podem dificultar a extração no caso específico de *Ptolemy*.

No trabalho mencionado, cinco categorias para o código de tratamento de exceção são abordadas: (1) Emaranhamento de blocos *try..catch*, (2) Código terminal de lançamento de exceção, (3) Aninhamento de blocos *try..catch*, (4) Dependência de blocos tratadores em variáveis locais e (5) Fluxo de controle após execução do bloco tratador. As categorias 1, 2 e 3 não apresentam impacto direto para a extração devido à maneira como o código é extraído e as categorias 4 e 5 podem, de fato, representar problemas para a extração com *Ptolemy*.

A categoria 1 indica onde, no corpo de um método, um bloco *try..catch* aparece. Um bloco *try..catch* é dito *emaranhado* se existe algum trecho de código que precede ou sucede o bloco. Blocos *try..catch* emaranhados podem dificultar a extração em *AspectJ* pois é difícil para o aspecto tratador capturar o ponto de junção de interesse. Além disso, ele também pode tornar mais difícil para um aspecto tratador manter o fluxo de controle do código original. Esses problemas não se aplicam à abordagem geral para extração descrita neste capítulo, pois existe uma correspondência direta entre um bloco *try* e um bloco *announce*, ou seja, não há diferenças em relação ao trecho de código coberto pelo *try* original e pela classe tratadora. Adicionalmente, não há problemas para simular o fluxo de controle original

já que, após a execução do tratador de exceções no método tratador de eventos, o fluxo continua exatamente após o bloco *announce* (caso o tratador não levante uma exceção), identicamente ao código original. As Figuras 12 e 13 ilustram essa capacidade de Ptolemy.

|   |  |
|---|--|
| <pre> 01 public String read(String path){ 02   File file = new File(path); 03   StringBuffer contents = ...; 04   try { 05     FileInputStream fis = ...; 06     char c; 07     while(/* read stream*/) { 08       contents.append(c); 09     } 10   }catch (FileNotFoundException e) { 11     // ... 12   }catch (IOException e) { 13     // ... 14   } 15   return contents.toString(); 16 } </pre> | <pre> 01 public String read(String path){ 02   File file = new File(path); 03   StringBuffer contents = ...; 04   announce FileReadEvent(f) { 05     FileInputStream fis = ...; 06     char c; 07     while(/* read stream*/) { 08       contents.append(c); 09     } 10   } 11   return contents.toString(); </pre> |
|---|--|

Figura 12 - Extração de *try..catch* emaranhado e com código não terminal (A)

|  |  |
|--|--|
| <pre> 01 public String read(String path){ 02   File file = new File(path); 03   StringBuffer contents = ...; 04   announce FileReadEvent(f) { 05     FileInputStream fis = ...; 06     char c; 07     while(/* read stream*/) { 08       contents.append(c); 09     } 10   } 11   return contents.toString(); </pre> | <pre> 01 void handle(FileReadEvent next){ 02   try { 03     next.invoke(); 04   } 05   catch (FileNotFoundException e){ 06     // ... 07   } 08   catch (IOException e) { 09     // ... 10   } 11 } </pre> |
|--|--|

Figura 13 - Extração de *try..catch* emaranhado e com código não terminal (B)

A categoria 2 diz respeito a instruções que podem lançar exceções dentro de um bloco *try*. Uma instrução que pode lançar exceções que podem ser capturadas por um bloco *catch* é dita *terminal* se a mesma for a última instrução no fluxo do bloco *try* associado. Para o estudo em *AspectJ*, blocos com instruções não terminais eram difíceis de serem extraídos, pois em certos cenários manter o fluxo de controle do programa original era quase impossível sem modificar significativamente o código original. Por motivos semelhantes aos da categoria 1, esse tipo de problema não afeta a abordagem geral de extração já que o mesmo conjunto de instruções continua sendo coberto depois da extração. Essa característica pode ser observada também na Figura 12 e na Figura 13.

A categoria 3 classifica o código de tratamento de acordo com seu aninhamento em relação a outros blocos `try..catch`. Um bloco está *aninhado* em outro bloco `try..catch` se o mesmo está contido na região protegida do segundo. Em *AspectJ*, esse tipo de situação requer atenção especial na hora de ordenar os adendos. Em *Ptolemy* esse fato é abstraído para os aspectos já que blocos `announce` podem ser facilmente aninhados<sup>7</sup> para simular a estrutura de aninhamentos dos `try..catchs` originais, um exemplo desse aninhamento pode ser visto na Figura 14. Um bloco `try..catch` também pode estar aninhado dentro um bloco tratador, entretanto, como descrito no trabalho original, esses blocos não são levados em consideração ao realizar a extração já que os mesmos fazem parte do comportamento excepcional do sistema e devem ser extraídos como um todo.

```

01 try{
02   String contents = readToEnd(file);
03   try{
04     return parseContents(contents)
05   } catch(ParseException pe){
06     // ...
07   }
08 } catch(IOException e){
09   // ...
10 }

```

```

01 announce IOEvent(){
02   String contents = readToEnd(file);
03   announce ParseEvent(){
04     return parseContents(contents)
05   }
06 }

```

**Figura 14 - Anúncio aninhado**

A categoria 4 classifica os blocos tratadores levando em conta suas interações com variáveis locais. Um bloco tratador é dito *dependente de contexto* se o mesmo lê uma variável local definida anteriormente ao bloco `try..catch` mas não escreve (realiza atribuição) em nenhuma dessas variáveis. Se um bloco tratador escreve uma variável local ele é dito *modificador de contexto*. Blocos tratadores que interagem com variáveis locais são difíceis de serem extraídos em *AspectJ*, pois a linguagem não oferece suporte adequado para acessar contexto local. Em *Ptolemy*, esses tipos de tratadores também podem dificultar a extração, embora por outros motivos. Variáveis locais podem ser expostas através das variáveis de contexto do evento, entretanto, elas não podem ser escritas. No contexto deste trabalho, podemos estender essa classificação dos blocos tratadores para as regiões protegidas. Analogamente, dizemos que um bloco `try` é dependente de contexto se ele lê (mas não atribui) uma variável local e modificador de contexto se ele atribui algum valor a uma variável local de um escopo mais externo. A Figura 15 mostra exemplos de cenários com blocos `try` dependentes de contexto (esquerda) e modificadores de contexto (direita).

<sup>7</sup> O aninhamento léxico de blocos `announce` não estava funcionando corretamente na versão do compilador utilizada, entretanto o bug já havia sido identificado pelos desenvolvedores.

```

01 String sql = "..";
02 try{
03 // ...
04 stmt.executeQuery(sql);
05 } catch (SQLException e) {
06 // ...
07 }

```

```

01 String sql = "...";
02 try{
03 // ...
04 sql += employee.getLogin();
05 // ...
06 stmt.executeUpdate(sql);
07 } catch (SQLException e) {
08 // ...
09 }

```

**Figura 15 - Bloco try dependente de contexto (esquerda) e modificador de contexto (direita)**

A quinta e última categoria concerne o fluxo de controle após a execução de um bloco tratador de exceções. Um bloco tratador que termina sua execução com uma instrução de retorno é classificado como *de retorno*. Blocos que terminam sua execução com instruções *break* ou *continue* são classificados como *de terminação de laço* e *de continuação de laço*, respectivamente. Tratadores que terminam sua execução lançando uma exceção são denominados *de relançamento*, e tratadores que terminam normalmente são ditos *de mascaramento*. Em *AspectJ*, os tratadores nessa categoria são, geralmente, difíceis de extrair. Em *Ptolemy*, tratadores de mascaramento ou relançamento são triviais já que a extração não altera o fluxo de controle após a execução desses tipos de tratadores. Por outro lado, tratadores de retorno só podem ser extraídos em cenários específicos, e tratadores de continuação ou terminação de laço virtualmente não podem ser extraídos. Assim como feito na categoria anterior, nós também podemos estender esta categoria para regiões protegidas, podendo classificá-las como de retorno, de continuação de laço e de terminação de laço. Adicionalmente, considerando apenas a terminação não excepcional de blocos, podemos classificar tratadores e regiões protegidas de retorno como de *completamente (ou totalmente) de retorno* ou *parcialmente de retorno*. Blocos completamente de retorno terminam, em todos os seus possíveis pontos de saída, com um comando de retorno caso contrário eles são dito parcialmente de retorno. Um exemplo de um bloco parcialmente de retorno acontece quando uma das alternativas de um *if..else* contém um comando de retorno e outras não, nesses casos a terminação do bloco poderia acabar com o retorno do método ou continuar normalmente com a execução do mesmo. A Figura 16 mostra um exemplo de bloco *try* parcialmente de retorno.



```

01 try {
02   StringBuffer sb = new StringBuffer();
03   char c;
04   while((c = (char) reader.read()) != -1) {
05     sb.append(c);
06   }
07   if(sb.length() > 0){
08     return sb.toString();
09 }
10 } catch (IOException e) {
11   // ...
12 }

```

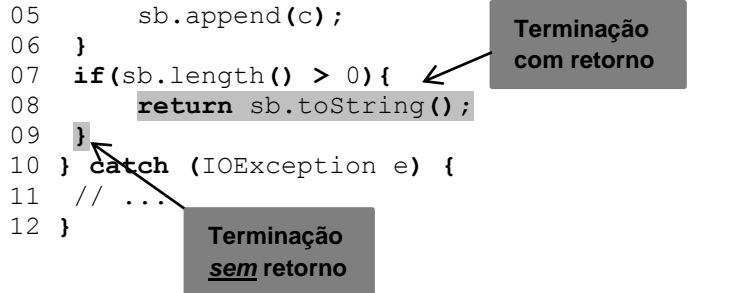


Figura 16 - Região protegida parcialmente de retorno

### 3.4 Cenários de extração

A lista de cenários descrita nesta seção não é exaustiva. Alguns dos cenários foram observados durante a extração de um sistema preexistente, o *HealthWatcher* (HW)[Soares et al. 2002]. O HW é um sistema web que permite aos usuários registrar ocorrências sobre problemas de saúde pública. Esse sistema foi desenvolvido, inicialmente, como uma *testbed* e já foi utilizado em vários estudos sobre a extração, modularização e reuso do código de tratamento de exceções com *AspectJ* [Castor et al. 2009; Cacho et al. 2009; Coelho et al. 2011]. Apesar de seu código de tratamento de exceções ser relativamente simples, boa parte dos tratadores nesse sistema não poderia ser facilmente extraída, pois eles se enquadram em alguns dos cenários não extraíveis descritos abaixo. Outra fonte para os cenários foi o já recorrentemente citado [Castor et al. 2009], partes do cenários descritos nesse trabalho também serão analisadas aqui. O restante dos cenários foi concebido através da composição de algumas das categorias descritas anteriormente.

Os cenários considerados neste trabalho são resumidos na Tabela 1, juntamente com a classificação indicando se o mesmo é extraível ou não. Um cenário extraível é aquele em que a abordagem geral para extração pode ser executada sem que haja a necessidade de alguma modificação manual significativa no código base anteriormente ou posteriormente à extração. Isto significa que não deve ser necessário ter algum conhecimento sobre o código que está sendo extraído. O intuito é que um cenário extraível possa também ser extraído por alguma ferramenta automática de refatoramento. Por outro lado, cenários não extraíveis requerem algum conhecimento semântico sobre o código que está sendo extraído e precisam de alguma forma ser refatorados para possibilitar a extração.

O cenário 1 é extraível, pois pode-se expor variáveis locais para tratadores de eventos, e conseqüentemente os tratadores de exceções extraídos, através das variáveis de contexto do evento anunciado. O cenário 2 não é extraível porque atribuições em variáveis locais escritas em uma região protegida não são visíveis para os tratadores de exceções extraídos já que ambos (o bloco *try* e os blocos tratadores) são executados em contextos diferentes. O cenário 3 pode ser extraído se utilizarmos o resultado da expressão *announce* para modificar o valor da variável local que é atribuída dentro da construção *try..catch*. Pode-se extrair o cenário 4 também utilizando o valor retornado por uma expressão *announce*, neste caso o valor retornado deve ser utilizado como retorno do método que contém o bloco *try..catch*. Os cenários 5 e 6 não são extraíveis, pois não é possível manter o mesmo fluxo de controle não excepcional após a extração, nesses cenários alguns dos possíveis fluxos de controle não retornariam e isto seria incompatível com a extração já que a *closure* e os tratadores de evento devem retornar em todos os seus fluxos. O sétimo e último cenário não é extraível porque instruções de controle de laço dentro de um bloco *announce* não podem se referir a um laço mais externo ao bloco e, no caso dos tratadores, essas instruções fariam parte de outro método (o do tratador de eventos) após a extração.

| Número | Nome do cenário  | Extraível |
|--------|--|-----------|
| 1      | Tratador dependente de contexto  | Sim       |
| 2      | Região protegida modificadora de contexto com tratador dependente de contexto  | Não       |
| 3      | Tratador modificador de contexto   | Sim       |
| 4      | Região protegida totalmente de retorno com tratador totalmente de retorno  | Sim       |
| 5      | Região protegida ou tratador parcialmente de retorno   | Não       |
| 6      | Região protegida não de retorno com tratador totalmente de retorno ou região protegida totalmente de retorno com tratador de mascaramento. | Não       |
| 7      | Tratador ou região protegida de terminação de laço ou continuação de laço  | Não       |

Tabela 1 - Resumo dos cenários de considerados

### 1 - *Tratador dependente de contexto*

Esse cenário extraível é composto por um ou mais tratadores classificados como dependente de contexto com a adição de, possivelmente, algum outro cenário extraível. Apesar da abordagem geral de extração poder ser utilizada, algumas modificações devem ser feitas. Uma vez que o tratador de exceções é extraído, as variáveis

locais do método alvo da extração não podem mais ser referenciadas diretamente, pois o tratador agora se encontra em um novo contexto (o do método tratador de eventos). Para expor as variáveis locais aos tratadores extraídos, é necessário definir, para cada variável local, uma variável de contexto na declaração do evento que será anunciado e associar cada variável local a sua respectiva variável de contexto durante o anúncio do evento. Adicionalmente, as referências às variáveis locais nos tratadores extraídos devem ser substituídas por referências às suas respectivas variáveis de contexto da *closure* do método tratador de eventos. A Figura 17 exemplifica todo o processo de extração de um tratador dependente de contexto combinado com um tratador de bloco. Se mais de um tratador dependente de contexto estiver associado a uma região protegida então todas as variáveis locais das quais ele depende também devem ser expostas durante o anúncio do evento.



Figura 17 - Exemplo de tratador dependente de contexto e sua extração

## 2 - Região protegida modificadora de contexto com tratador dependente de contexto

Esse cenário demonstra como a interação entre duas categorias individualmente extraíveis pode resultar em um cenário não extraível. O cenário é o que ocorre quando um tratador dependente de contexto depende de uma variável local modificada por uma região protegida. Esse cenário remete à limitação de *Ptolemy* em compartilhar variáveis locais adequadamente entre diferentes contextos de execução, neste caso o contexto da *closure* do evento e o do método tratador de eventos. A extração não é possível.

vel porque os valores expostos pela *closure* do evento para o código do método tratador de eventos são aqueles disponibilizados como variáveis de contexto durante o anúncio do evento, ou seja, se o bloco do *announce* modifica uma variável local disponibilizada como variável de contexto do evento, essa modificação *não* será refletida (exposta) para o código que tratará o evento. Tomando como referência a Figura 18, em A temos uma instância do cenário não extraível e em B e C temos uma extração hipotética. Como podemos ver em B, o valor da variável *sql* é disponibilizado como uma variável de contexto durante o anúncio do evento *SQLExecEvent* na linha 2, e essa variável é modificada posteriormente dentro do bloco do anúncio (linha 4). Caso uma *SQLException* seja levanta durante a execução do método *executeQuery* na linha 5, a mesma será capturada pelo tratador de exceções adequado em C. Durante a execução do bloco do tratador de exceções em C, a variável de contexto *sql* é acessada (linha 5) para construir uma exceção que será propagada, entretanto o valor da variável de contexto não reflete o valor corrente da variável local *sql* já que a mesma foi modificada durante a execução da linha 4 em B e o valor da variável de contexto foi definido durante a execução da linha 2 também em B. Esse contraste entre o valor da variável de contexto e o valor corrente da variável local durante a execução do tratador é o que impossibilita a extração. Não há como garantir que o comportamento do programa será o mesmo, enquanto no trecho não refatorado (A) o tratador sempre terá acesso ao valor corrente da variável local *sql*, no tratador refatorado o mesmo não pode ser dito.

```

01 String sql = null;
02 try {
03     stmt = (Statement) mp.getCommunicationChannel();
04     sql = "...";
05     resultSet = stmt.executeQuery (sql);
06     // ...
07 } catch (SQLException e){
08     throw new SQLPMEException (ExceptionMessages.EXC_FALHA_BD, sql);
09 }

```

A

```

01 String sql = null;
02 announce SQLExecEvent (sql) {
03     stmt = (Statement) mp.getCommunicationChannel();
04     sql = "...";
05     resultSet = stmt.executeQuery (sql);
06     // ...
07 }

```

B

```

01 void handle (SQLExecEvent next){
02     try{
03         next.invoke ();
04     } catch (SQLException e){
05         throw new SQLPMEException (ExceptionMessages.EXC_FALHA_BD, next.sql ());
06     }
07 }

```


C

**Figura 18 - Exemplo de região protegida modificadora de contexto com tratador dependente de contexto**

### 3 - Tratador modificador de contexto

Este cenário extraível envolve tratadores que modificam variáveis locais do método alvo. A extração pode ser realizada se utilizarmos o resultado do bloco *announce* como o a valor que deve ser atribuído à variável após a execução do bloco *try..catch*. Portanto, após a extração, tanto a *closure* do evento quanto os tratadores de evento devem retornar o valor dessa variável local. A Figura 19 mostra um exemplo dessa extração. É necessário definir um evento com um tipo de retorno igual ao da variável que será atribuída e inserir comandos de retorno no final das execuções do tratador (de eventos) e da *closure*. Ao final do anúncio do evento, a variável receberá o valor computado pela *closure*, para o caso do fluxo não excepcional, ou pelo tratador de eventos, para o caso em que uma exceção foi lançada. As limitações desse cenário são semelhantes ao do refatoramento *Extract Method* [Fowler 1999]. Para casos em que mais de uma variável é atribuída não é possível realizar a extração diretamente sem utilizar algum artifício para retornar mais de um valor.

```
01 String contents;
02 try {
03     contents = readRemote ();
04 } catch( RemoteException e ){
05     contents = readLocal ();
06 }
07 return contents ;
```



```
01 public String ContentReadEvent{}
02
03 String contents ;
04 contents = announce ContetReadEvent () {
05     contents = readRemote ();
06     return contents;
07 }
08 return contents ;
```

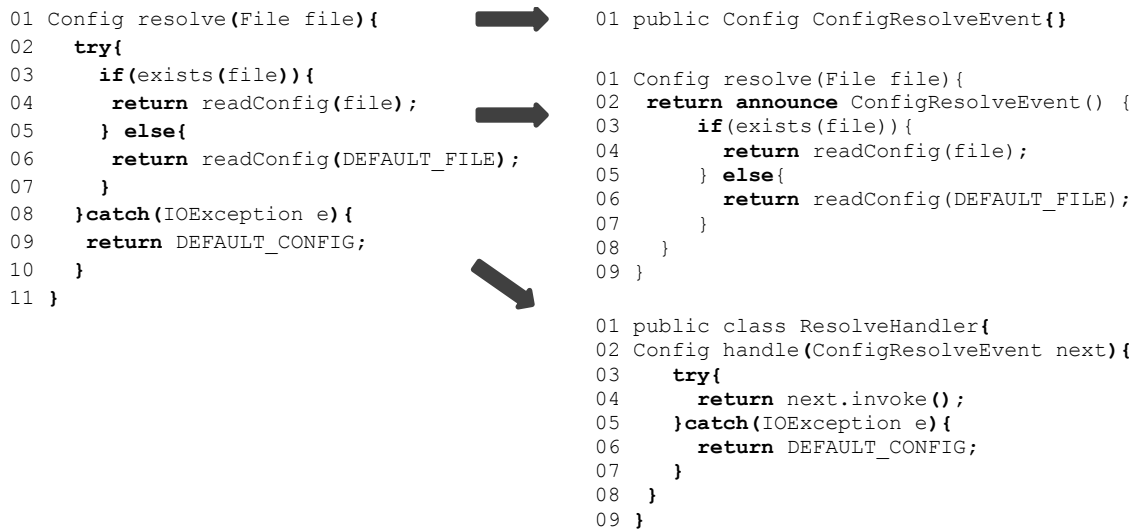
```
01 String handle(ContentReadEvent next){
02     String contents;
03     try {
04         contents = next.invoke();
05     } catch( RemoteException e ){
06         contents = readLocal ();
07     }
08     return contents;
09 }
```

Figura 19 - Exemplo de tratador modificador de contexto

### 4 - Região protegida totalmente de retorno com tratador totalmente de retorno

Uma variação da abordagem geral pode ser utilizada para extrair o tratamento neste cenário. A Figura 20 mostra uma instância do cenário e o resultado do processo de extração. É necessário definir um tipo de evento que retorne um valor do tipo definido pelo método alvo da extração. Além disso, o método tratador do evento deve retornar o resultado da execução do *invoke* ou o resultado definido pelo bloco tratador da exceção. Por fim, como o retorno dos tratadores de exceções está em outro método e o comando de retorno dentro do bloco *announce* se refere à *closure*

do evento, é necessário utilizar o resultado do anúncio<sup>5</sup> do evento como retorno do método extraído.



**Figura 20 - Exemplo de região protegida totalmente de retorno com tratador totalmente de retorno e sua extração**

É importante observar que caso haja mais de um tratador de exceções, todos devem ser de retorno ou de relançamento. Se pelo menos um dos tratadores for de mascaramento não será possível manter o fluxo de controle original já que o método tratador de eventos deve obrigatoriamente retornar um valor. Tratadores de continuação e terminação de laço não são extraíveis como será visto posteriormente.

### 5 - Região protegida ou tratador parcialmente de retorno

Cenários que contêm blocos *try* ou *catch* parcialmente de retorno não são extraíveis porque o novo contexto em que esses blocos seriam executados após uma extração hipotética não permitiria um retorno parcial ou manter o mesmo fluxo de controle do método original após a execução desses blocos. Para métodos que possuem um tipo de retorno, os blocos *try* ou *catch* parciais, depois de extraídos, possuiriam pontos de terminação sem um retorno explícito em seus respectivos contextos (*closure* do evento e método tratador), o que não é permitido pela linguagem. Para métodos que não retornam algum valor, os comandos de retorno nesses blocos fariam apenas com que o novo contexto em que eles se encontram retornasse (terminasse) e não o método original. Tomando como exemplo a Figura 21, podemos ver um bloco *try* parcialmente de retorno e o anúncio de seu evento após uma possível extração, nesse exemplo a *closure* do evento poderia terminar sem explicitamente retornar um valor.

```

01 Config resolve(File file){
02     try{
03         if(exists(file)){
04             return readConfig(file);
05         }
06     }catch(IOException e){
07         // ...
08     }
09     return DEFAULT_CONFIG;
10 }

```

```

01 public Config ConfigResolveEvent{}
02
03 public Config resolve(File file){
04     announce ConfigResolveEvent(){
05         if(exists(file)){
06             return readConfig(file);
07         }
08     }
09     return DEFAULT_CONFIG;
10 }

```

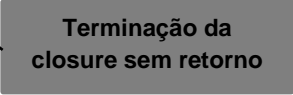


Figura 21 - Exemplo de região protegida parcialmente de retorno

**6 - Região protegida não de retorno com tratador totalmente de retorno ou região protegida totalmente de retorno com tratador de mascaramento.**

A justificativa da incapacidade de extrair estes cenários está ligada ao fato de que blocos *try..catch* podem, arbitrariamente, terminar ou não com a execução de um método através de um comando de retorno. No primeiro caso, a extração poderia resultar em uma *closure* de evento que não retorna um valor para o código base (para métodos não *void*) ou um tratador de exceções que não termina a execução do método original (para métodos *void*). No segundo caso, poderíamos ter a situação inversa, uma *closure* que não termina a execução do método original ou um tratador que não retorna um valor para o código base. Na Figura 22 podemos visualizar uma instância do primeiro cenário e quais as complicações caso tentássemos extraí-lo. Como pode ser observado, o corpo do tratador de eventos não retornaria um valor em todos os seus possíveis fluxos, o que não é permitido pela linguagem. Além disso, como o tratador de exceções foi movido do método *resolve* e o bloco *try* transformado em um bloco *announce*, foi necessário introduzir um comando de retorno a mais para que o método original pudesse retornar o resultado da execução da *closure* do evento. Isso cria uma situação ilegal em que um comando de retorno segue imediatamente outro.

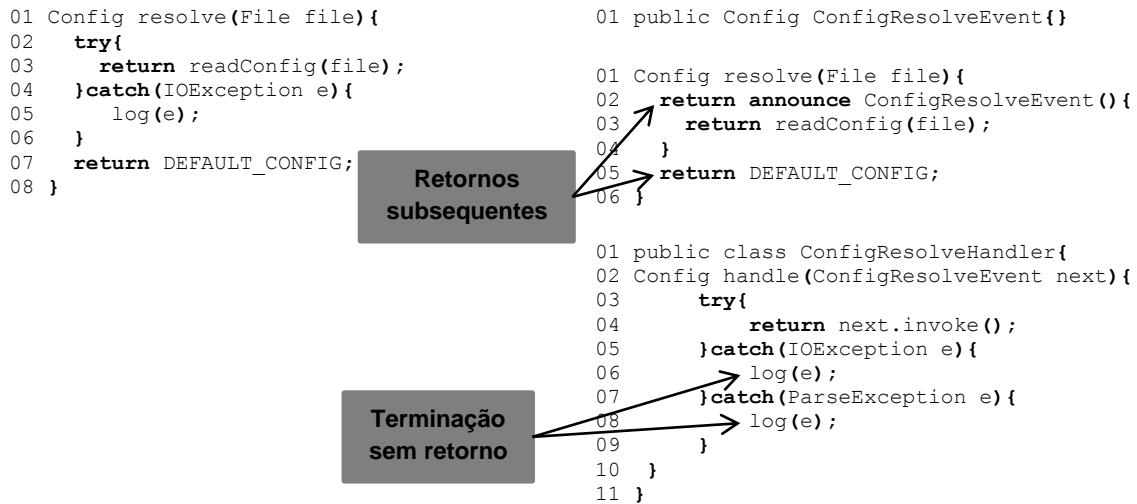


Figura 22 - Exemplo de região protegida totalmente de retorno com tratador de mascaramento

Na Figura 23 podemos ver um exemplo do segundo cenário e sua (im)possível extração. O complicador desse cenário está no fato de que o comando de retorno dentro do bloco tratador de exceções, quando extraído, não fará com o que método o *waitAll* termine, mas apenas o método tratador de eventos.

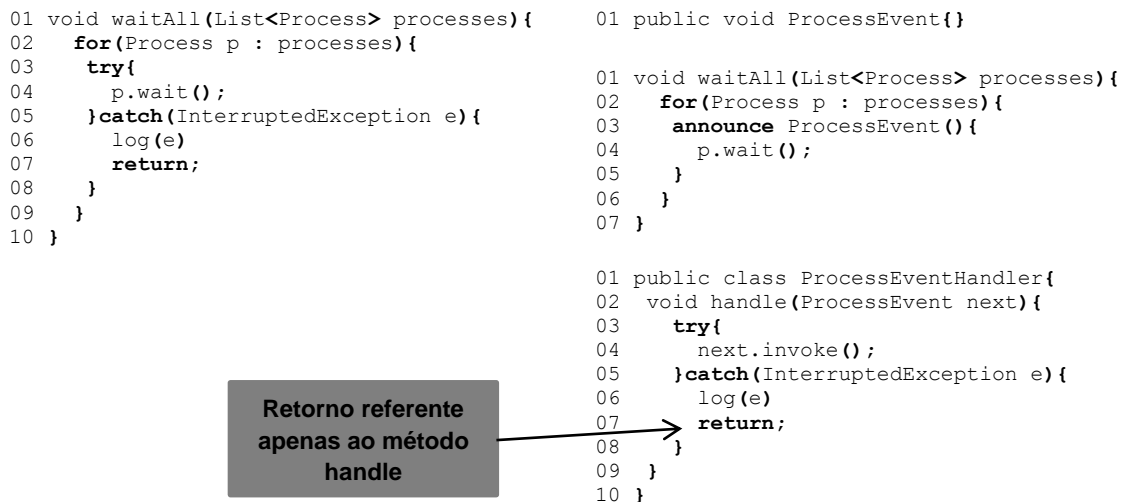


Figura 23 - Exemplo de região protegida não de retorno com tratador totalmente de retorno

### 7 - Tratador ou região protegida de terminação de laço ou continuação de laço

De maneira geral, instâncias desses cenários não são extraíveis, pois comandos para desvio de fluxo de controle em laços não podem se referir a laços em outros contextos. Como após a extração o bloco *try* e os blocos dos tratadores são executados em outro contexto, da *closure* do evento e do método tratador de eventos, respectivamente, não é possível realizar a extração quando esses blocos contêm um comando *break* ou um comando *continue*. A Figura 24 mostra instâncias de tratadores de terminação e continuação de laço e a Figura 25 mostra instâncias de regiões protegidas análogas.



```

01 int failThreshold = 0 ;
02 for ( int i = 0 ; i < 100 ; i++){
03     try{
04         update();
05     }catch (IOException e){
06         failThreshold++;
07         if( failThreshold > 15) {
08             break ;
09         }
10     }
11 }

01 while ( /* condição */) {
02     try {
03         writeFile(..);
04     } catch ( IOException e ) {
05         continue;
06     }catch(FileNotFoundException e) {
07         continue;
08     }
09     notifyObservers();
10 }

```

Figura 24 - Exemplos de tratador de terminação de laço (esquerda) e continuação de laço (direita)

```

01 int i = 0 ;
02 while ( i++ < 5) {
03     try {
04         writeFile();
05         break;
06     }catch ( IOException e ) {
07         // ...
08     }catch(FileNotFoundException e) {
09         // ...
10     }
11 }

01 while (/* ... */) {
02     try {
03         writeFile(..);
04         // sleep...
05         continue;
06     }catch(IOException e) {
07         // ...
08     }catch( FileNotFoundException e ) {
09         // ...
10     }
11     killProcesses();
12 }

```

Figura 25 - Exemplos de região protegida de terminação de laço (esquerda) e continuação de laço (direita)

### 3.5 Ptolemy e os mecanismos de tratamento de exceções


O mecanismo de anúncio e tratamento de eventos de *Ptolemy* possui uma característica bastante pertinente ao tratamento de exceções e que deve ser observada. Como visto anteriormente, a execução da *closure* de um evento ou de um tratador de eventos pode, arbitrariamente, lançar qualquer tipo de exceção (checadas ou não checadas). Uma das implicações disso é que um método que anuncia eventos também pode, arbitrariamente, lançar qualquer tipo de exceção, incluindo exceções checadas que não façam parte da sua interface. Isso pode resultar em situações em que clientes desses métodos não estão preparados para tratar essas exceções<sup>8</sup>. Problemas semelhantes a esse também existem em *AspectJ*, onde exceções podem ser introduzidas pelos aspectos. Coelho et al. [Coelho et al. 2011] propõem uma abordagem para amenizar esse problema que também poderia ser estendida para *Ptolemy*.

Outra observação que pode ser feita é sobre o comportamento de supressão da verificação de exceções de blocos *announce*. Por permitir que a *closure* do evento

<sup>8</sup> A situação é um pouco pior para métodos que lançam exceções checadas que não fazem parte da interface de exceções dos mesmos. Clientes desses métodos simplesmente **não** poderiam cobri-los com tratadores para essas exceções já que a linguagem não permite associar tratadores de exceções checadas a regiões protegidas que não possuam pelo menos uma instrução que indique a possibilidade de lançar uma exceção desse tipo.

lance qualquer tipo de exceção, o compilador suprime as verificações estáticas com relação às possíveis exceções checadas lançadas dentro de um bloco *announce*. Para um desenvolvedor desavisado, seria possível, por exemplo, implementar um trecho de código como o da Figura 26 sem tomar conhecimento que as linhas 3 e 4 poderiam levantar uma *IOException*. Nesse exemplo, o bloco *announce* faria com que o compilador suprimisse avisos relacionados ao potencial lançamento de exceções checadas no bloco. Pode-se notar também que o desenvolvedor não é nem mesmo obrigado a declarar *IOException* na interface do método *read*.

```
01 String read(File f) {
02   return announce FileReadEvent (f) {
03     FileReader reader = new FileReader (f);
04     return readToEnd (f);
05   }
06 }
```



**Figura 26 - Bloco *announce* suprimindo verificação de exceções**

Em *AspectJ* também é possível suprimir as verificações estáticas para exceções checadas através do relaxamento (do inglês: *softening*) de exceções. Esse relaxamento é feito explicitamente pelo desenvolvedor do aspecto que pretende suprimir essa verificação. A diferença com relação a *Ptolemy* está justamente no fato dessa supressão ser explícita ao invés de acontecer implicitamente, pelo menos uma das partes envolvidas (o desenvolvedor do aspecto) deve se comprometer ao realizar a supressão. Para amenizar esse problema de supressão de verificações, uma abordagem como a apresentada por Hoffman e Eugster [Hoffman e Eugster 2009] poderia ser utilizada. A declaração de eventos poderia conter também uma interface de exceções indicando que exceções o anúncio daquele evento poderia lançar ou que exceções poderiam ser tratadas caso fossem levantada durante o anúncio.

## 4. Trabalhos futuros

Seguindo o mesmo raciocínio dos estudos realizados anteriormente [Lippert e Lopes 2000], [Hoffman e Eugster 2009], [Cacho et al. 2002], [Castor et al. 2009], uma extensão natural para este trabalho seria avaliar quantitativamente e qualitativamente o sistema estudado após a extração. O estudo poderia focar em métricas de tamanho, coesão, acoplamento e separação de interesses já utilizadas nos outros trabalhos. Esse estudo poderia mostrar quão adequada *Ptolemy* é para o tratamento de exceções além dos problemas já encontrados neste trabalho.

Outro possível trabalho seria tentar estudar as interações entre tratamento de exceções modularizado e outros interesses transversais utilizando *Ptolemy*. Esse estudo complementaria o anterior e serviria também para mostrar como uma linguagem baseada em eventos explícitos se comporta na presença de interesses transversais que precisam cooperar para realizar o sistema.

Uma linha de pesquisa alternativa seria contribuir de alguma forma para a o desenvolvimento da linguagem tentando implementar soluções para amenizar ou resolver os problemas de extração encontrados neste trabalho.

## 5. Conclusão

Este trabalho propôs soluções para extrair o tratamento de exceções de um sistema existente utilizando uma linguagem orientada a aspectos com eventos explícitos. A linguagem utilizada foi *Ptolemy*, uma extensão de Java que contém mecanismos para definir, anunciar e tratar eventos explicitamente. Este trabalho também identificou as principais limitações da linguagem para modularizar o tratamento de exceções. Para ilustrar e facilitar o entendimento das soluções propostas e dos problemas da linguagem, este trabalho detalhou e estendeu a classificação do código de tratamento criada por [Castor et al. 2009] e utilizou essa classificação para definir cenários de extração.

*Ptolemy* se mostrou bastante eficiente para a modularização do tratamento de exceções em cenários que não envolviam blocos *try..catch* que possuíam comandos de controle de fluxo ou que escreviam em variáveis locais. Em contraste a *AspectJ*, a modularização de cenários envolvendo emaranhamento, aninhamento ou blocos *try* com instruções não terminais de levantamento de exceção se mostrou trivial devido à relação direta existente entre a região protegida de uma construção *try..catch* e bloco de anúncio correspondente após a extração.

Por outro lado, a extração de cenários contendo comandos de controle de fluxo, isto é, *return*, *break* ou *continue*, se mostrou difícil ou impossível. Para cenários contendo comandos de retorno, a extração só é viável caso todos os possíveis pontos de saída do fluxo não excepcional da construção *try..catch* terminem com um retorno explícito. Isto significa que a região protegida e todos os blocos tratadores devem terminar, em todos os seus fluxos, com um comando de retorno. Construções *try..catch* que possuem comandos do tipo *break* ou *continue*, tanto na região protegida quanto nos blocos tratadores, não são extraíveis em geral.

Outros cenários não extraíveis estão relacionados com a escrita em variáveis locais do método alvo. Mais uma vez, o contexto em que o método original, os tratadores extraídos e a *closure* do evento são executados impossibilita a extração. Cenários contendo uma região protegida que escreve em uma variável local que é lida por um tratador não são extraíveis.

## 5. Referências

1. <http://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html>. Último acesso em 28/05/2012 às 21h36min.
2. <http://docs.oracle.com/javase/specs/jvms/se7/html/index.html>. Último acesso em 28/05/2012 às 21h37min.
3. Alessandro Garcia, Cecília Rubira, Alexander Romanovsky, Jie Xu. 2001. A comprative study of exception handling mechanisms for building dependable object-oriented software.
4. <http://www.eclipse.org/aspectj/doc/next/progguide/index.html>. Último acesso em 07/06/2012.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. 1997. Aspect-Oriented Programming. In Proceeding of the European Conference on Object-Oriented Programming (ECOOP 1997).
6. Robert Filman e Daniel Friedman. 2000. Aspect-Oriented Programming is Quantification and Obliviousness. OOPSLA 2000.
7. Peri Tarr, Harold Ossher, William Harrison e Stanley M. Sutton, Jr.. 1999. N degrees of separation: multi-dimensional separation of concerns. In Proceedings of the 21st international conference on Software engineering (ICSE 1999). Páginas 107-119.
8. Robert Filman. 2001. What is Aspect-Oriented Programming, Revisited. In 15th European Conference on Object-Oriented Programming (ECOOP 2001).
9. <http://ptolemy.cs.iastate.edu/docs/lang.shtml>. Último acesso em 07/06/2012 às 15h24min.
10. Hridesh Rajan e Gary Leavens. 2008. Quantified, Type Events for Improved Separation of Concerns. In 22nd European Conference on Object-Oriented Programming (ECOOP 2008).
11. <http://www.aosd.net/wiki/index.php?title=Glossary>. Último acesso em 17/06/2012 às 21h59min.
12. Patrick Eugster, Pascal Felber, Rachid Guerraoui e Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. ACM Computing Survey, Volume 35, Edição 2, Páginas 114-131.
13. Kevin Hoffman e Patrick Eugster. 2008. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In Proceedings of the 30th international conference on Software engineering (ICSE 2008). Páginas 91-100.
14. Julio Cesar Taveira, Cristiane Queiroz, Romulo Lima, Juliana Saraiva, Emanuel Barreiros, Fernando Castor, Sergio Soares, Nathalia Temudo, Hitalo Oliveira, Amanda Araujo e Jefferson Amorim. 2009. Assessing Intra-Application

- Exception Handling Reuse with Aspects. In Proceedings of the 23rd Brazilian Symposium on Software Engineering.
15. Martin Lippert e Cristina Videira Lopes. 2000. A study on exception detection and handling using aspect-oriented programming. In Proceedings of the 22nd international conference on Software engineering (ICSE 2000). Páginas 418-427.
  16. Roberta Coelho, Arndt von Staa, Uirá Kulesza, Awais Rashid e Carlos Lucena. 2011. Unveiling and taming liabilities of aspects in the presence of exceptions: A static analysis based approach. Information Sciences, Volume 181, Edição 13, Páginas 2700-2720.
  17. Nélio Cacho, Francisco Dantas, Alessandro Garcia e Fernando Castor. 2009. Exception Flows Made Explicit: An Exploratory Study. In XXIII Brazilian Symposium (SBES 2009). Páginas 43-53.
  18. Kevin Hoffman e Patrick Eugster. 2009. Cooperative aspect-oriented programming. Science of Computer Programming, Volume 74, Edições 5–6, Páginas 333-354.
  19. Sérgio Soares, Eduardo Laureano e Paulo Borba. 2002. Implementing distribution and persistence aspects with AspectJ. In Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002). Páginas 174-190.
  20. Fernando Castor, Nélio Cacho, Eduardo Figueiredo, Alessandro Garcia, Cecília Rubira, Jefferson Amorim e Hítalo Silva. 2009. On the modularization and reuse of exception handling with aspects. Software Practice and Experience.
  21. Hridesh Rajan e Kevin Sullivan. 2005. Classpects: Unifying Aspect- and Object-Oriented Language Design. In Proceeding of the 27th International Conference on Software Engineering (ICES 2005).
  22. Nelio Cacho, Fernando Castor Filho, Alessandro Garcia e Eduardo Figueiredo. 2008. EJFlow: taming exceptional control flows in aspect-oriented programming. In Proceedings of the 7th international conference on Aspect-oriented software development (AOSD 2008). Páginas 72-83.
  23. Hidehiko Masuhara , Gregor Kiczales. 2003. Modeling Crosscutting in Aspect-Oriented Mechanisms. In Proceeding of the European Conference on Object-Oriented Programming (ECOOP 2003).
  24. Robert Dyer, Hridesh Rajan and Yuanfang Cai. 2012. An Exploratory Study of the Design Impact of Language Features for Aspect-oriented Interfaces. 11th International Conference on Aspect-Oriented Software Development (AOSD 2012).
  25. Martin Fowler 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.