Universidade Federal de Pernambuco – UFPE

Centro de Informática

Suporte a Interfaces Bidimensionais para Exceções em Java na Plataforma de Desenvolvimento Eclipse

Filipe Marques Chaves de Arruda

Recife, Brasil

Fevereiro de 2015

Filipe Marques Chaves de Arruda

Suporte a Interfaces Bidimensionais para Exceções em Java na Plataforma de Desenvolvimento Eclipse

Trabalho apresentado ao Programa de GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO do CENTRO DE INFORMÁTICA da UNIVERSIDADE FEDERAL DE PERNAMBUCO.

Universidade Federal de Pernambuco – UFPE

Centro de Informática

Orientador: Prof. Dr. Fernando José Castor de Lima Filho

Recife, Brasil

Fevereiro de 2015

Agradecimentos

Primeiramente, agradeço a Deus por ter permitido e dado forças para a realização desse sonho.

À instituição e todos seus funcionários que prezam por um ensino de qualidade e proporcionam um ambiente de constante aprendizado.

Ao Prof. Dr. Fernando José Castor de Lima Filho por ter oferecido valiosas recomendações no pouco tempo que lhe coube.

Agradeço aos meus pais pela paciência, compreensão e por sempre terem investido na minha educação, mesmo em situações difíceis. Agradeço também à minha namorada, que me acompanhou por praticamente toda a graduação incentivando e me apoiando em todos os momentos. E também a todos meus amigos, principalmente aos que fiz durante a graduação, pelo companheirismo, pelo aprendizado conjunto e, sempre que possível, por transformar toda situação em diversão.

E, por fim, um muito obrigado a todos que diretamente ou indiretamente contribuíram para a minha formação.

Resumo

O conceito de exceções para identificar e tratar separadamente o desvio no fluxo lógico não é novidade, porém existem problemas inerentes às interfaces usadas em algumas linguagens. Em Java, por exemplo, a cláusula *throws* é suficiente para analisar o fluxo de exceções numa visão local e restrita, porém é limitada no contexto global tanto para controle quanto para manutenção. Para contornar esse problema existe uma extensão à linguagem chamada EPiC-Java, que provê uma sintaxe especial para especificar o fluxo da exceções desde os locais de lançamento até os de tratamento, mas não existe ferramenta que a suporte em *IDE*s. Para torná-la viável na prática, este trabalho visa adaptar a sintaxe proposta através da implementação de um *plug-in* para o ambiente de desenvolvimento Eclipse.

Palavras-chaves: exceções. java. eclipse. epic. interfaces bidimensionais.

Abstract

There isn't anything new about using the concept of exception to identify and handle logical flow disruptions, but there are a few problems associated to the exception interfaces in some languages. As an example, we can say the *throws* clause used in Java is enough to analyze the exception flow locally, but it is limited in a global context for both control and manutenability. To overcome this issue, an extension to the Java language (called EPiC-Java) was created, providing a special syntax to specify exceptions flows from throw sites to handle sites in just one sentence, but no tools were developed to provide IDE support. That being so, this work aims to adapt EPiC-Java syntax and make it portable to Eclipse environment as a plug-in.

Keywords: exceptions. java. eclipse. epic. bidimensional interfaces.

Sumário

Lista de ilustrações

Lista de tabelas

1 Introdução

Apesar de não exclusivamente relacionado às linguagens orientadas a objeto, a maioria das linguagens modernas deste paradigma utilizam o conceito de exceções para identificar e tratar o desvio no fluxo lógico esperado pela especificação do software (ROBILLARD; MURPHY, 2000). Em linguagens tradicionais, condições de erros são geralmente sinalizados por retorno de funções não-usuais ou absurdos no contexto como, por exemplo, -1 (ARNOLD et al., 1996) o que leva ao desenvolvedor a potencialmente ignorar ou sequer descobrir que alguma falha ocorreu. Este problema pode ser mitigado com o uso de exceções, pois a detecção, comunicação e tratamento de situações excepcionais são realizados separadamente do fluxo lógico principal de um programa, podendo ainda serem propagadas pela pilha de chamada de métodos para serem tratadas em outro local apropriado (CAMPIONE, 2001), constituindo-se, por exemplo, como um importante meio no *design* de *API*s para prover informações aos clientes sobre falhas e como tratá-las (BLOCH, 2006). Dito isto, a fim de terminologia, podemos definir métodos onde são originadas as exceções como locais de lançamento, os que possuem ações para tratá-las como locais de tratamento, e aqueles que se situam entre eles no fluxo de propagação como locais intermediários (SILVA; CASTOR, 2013).

1.1 Exceções em Java

De acordo com sua documentação oficial [1], a linguagem de programação Java possui três tipos de exceções: as de execução, os erros e as checadas. As de execução representam eventos excepcionais internos à aplicação, os quais não são esperados, interferindo no fluxo lógico de forma que seja improvável sua recuperação, representados em Java pela classe *RuntimeException*. Similarmente temos os erros, que são caracterizados pela classe Error, com a diferença que os erros se referem ao ambiente externo à aplicação. Já as checadas, como o próprio nome sugere, tem seus fluxos verificáveis pela análise estática realizada pelo compilador, sendo identificadas na linguagem pela classe *Exception* (Cód. 1, linha 7) ou quaisquer subclasses decorrentes exceto *RuntimeException*, que é reservada para exceções de execução.

Código 1: Código Java sem o devido tratamento da exceção checada

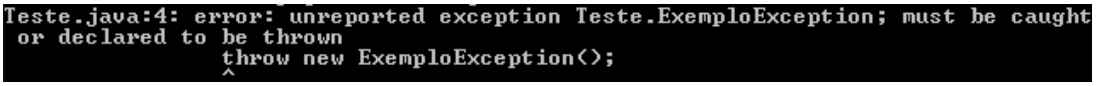
```
1 class Teste{
2
3   public void teste(){
4     throw new ExemploException();
5   }
6
7   class ExemploException extends Exception{}
```

[1] Disponível em: <http://docs.oracle.com/javase/tutorial/essential/exceptions/>. Acesso em 18/02/2015.

```
8 }
```

Quando uma exceção checada é lançada na implementação de qualquer método, este é obrigado a estabelecer uma política de captura: tratar localmente ou propagá-la para o próximo método da pilha. Desse modo, é possível antecipar problemas em tempo de compilação (Figura 1) e, consequentemente, tornar a aplicação mais robusta , pois obriga o estabelecimento de medidas para recuperação (ROBILLARD; MURPHY, 2003). Em Java, essa obrigação é manifestada através de blocos *try-catch*, que definem a política de recuperação, ou pela declaração de cláusulas *throws* na assinatura do método para transferir a responsabilidade de tratar as exceções identificadas a quem invocá-lo.

Figura 1: Erro de compilação: método deve declarar lançamento da exceção ou tratá-la.



Fonte: Screenshot do autor

1.2 Limitações

Como citado anteriormente, *Java* utiliza um sistema de declaração explícita acerca das exceções checadas que o método lança em sua assinatura, observável no Cód. 2. Apesar dessa abordagem deixar claro quais exceções o invocador deve tratar, podemos enumerar diversos problemas em relação ao fluxo de propagação.

Código 2: Lançamento de exceção declarado na assinatura do método

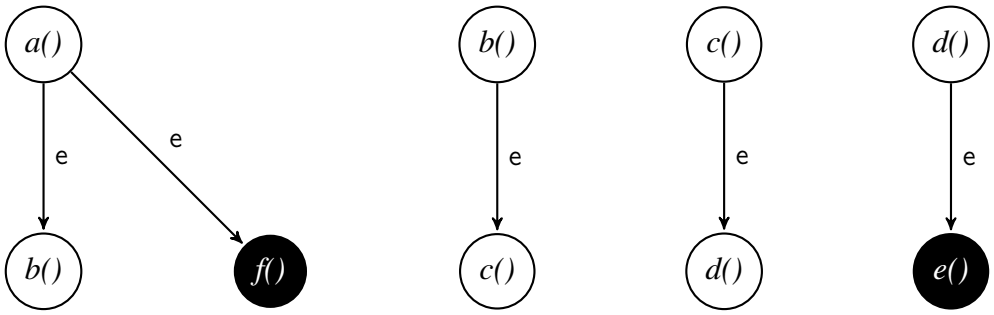
```
1  public void teste() throws ExemploException{
2    throw new ExemploException();
3  }
```

1.2.1 Análise e Controle

Há uma imensa divergência entre o modo em que a propagação é declarada e a natureza intríseca da propagação de exceções. Enquanto que o mecanismo de declaração na assinatura fornece uma visão estritamente local, a análise do fluxo entre os locais de lançamento e de tratamento é, na verdade, um problema a nível global da aplicação, especialmente em sistemas complexos. Como ilustração, podemos considerar uma situação trivial representada na Figura 2, onde fica evidente a falta de visão global do fluxo de uma exceção.

A Figura 2 é uma sequência de grafos direcionados que ilustra a busca de um desenvolvedor pelos locais de tratamento de uma dada exceção *e* lançada pelo método *a()*, onde métodos são representados como nós (nós preenchidos são locais de tratamento) e chamadas como arestas rotuladas pela exceção que o nó de origem lança. Para notar que o método *e()* é um local de

Figura 2: Representação da visão local consequente do uso de cláusulas *throws*



tratamento da exceção lançada por *a()*, por exemplo, é preciso analisar exaustivamente quem são seus invocadores e verificar se tratam ou relançam a exceção. Percebemos, então, que até num exemplo trivial a determinação do fluxo de exceções não é objetiva, como consequência do mecanismo limitado, requerindo várias etapas de análise.

Como destacado por CACHO et al., para mitigar este problema podem ser utilizadas ferramentas de análise estática como abordagem de obtenção da visão global sobre o fluxo de exceções através da exploração do grafo de chamadas geradas pela construção da árvore sintática do código. Dessa maneira, desenvolvedore são poupados da verificação manual. Porém, ainda de acordo com CACHO et al., estas ferramentas possuem limitações significativas: 1) Elas não são tão úteis enquanto o *software* ainda está em fase de desenvolvimento, pois os caminhos ainda não estão bem definidos; 2) Enquanto estas ferramentas são convenientes para analisar o fluxo, porém ineficientes para impor ou garantir determinado fluxo.

1.2.2 Manutenção

Atualmente há uma carência de ferramentas e/ou práticas que ajudem o desenvolvedor de software a estruturar e programar o mecanismo de tratamento de exceções em sistemas mais complexos (ROBILLARD; MURPHY, 2000), prejudicando a eficiência nas atividades de manutenção, teste e depuração (SHAH; GÖRG; HARROLD, 2008).

Observando novamente a Figura 2, consideremos a seguinte situação: suponhamos que o método *a()*, após modificações na implementação, a exceção *e()* é tratada locamente, enquanto que uma segunda exceção identificada como *e2* é lançada, devendo ser tratada nos métodos *d()* e *f()*. Dessa maneira, o método *a()* deve sinalizar em sua assinatura que lança nova exceção e, por consequência, todos os métodos entre *a()* e os locais de tratamento também devem atualizar suas assinaturas para indicar que também propagam a exceção *e2*. Além disso, surgem alguns problemas semânticos mais sérios, pois o desenvolvedor pode adicionar a nova exceção corretamente às assinaturas mas esquecer de remover a exceção *e* agora tratada localmente. Como o compilador não sinalizará nenhum erro, temos como consequência blocos de código para tratamento de exceção que nunca serão alcançados.

Como podemos avaliar, a atividade de alterar os fluxos editando manualmente todas

as assinaturas, além de consumir tempo de desenvolvimento, é essencialmente problemática principalmente no contexto da evolução do software, onde modificações devem ter uma apurada análise do impacto, pois se caracteriza por uma atividade propensa a erros (CACHO et al., 2008).

1.3 Interfaces Bidimensionais para Exceções

Para mitigar estas limitações, tanto de análise como de manutenção, o trabalho de SILVA; CASTOR propôs a adoção de interfaces bidimensionais através da especificação, adicionalmente, de interfaces verticais (*propagate*). Estas seriam responsáveis por expor a visão global do fluxo de exceções por toda a propagação através de cláusulas que expressam explicitamente o local de lançamento, os intermediários e o de tratamento. O uso conjunto das interfaces horizontais e verticais foi concretizada através de uma extensão à linguagem Java (EPiC-Java), a qual possui sintaxe própria para as cláusulas e um compilador protótipo. Porém, a utilização de uma sintaxe especial para reconhecimento e processamento das cláusulas, aliada à imposição de uso do compilador do *OpenJDK* [2] alterado e a ausência de suporte por qualquer ambiente de desenvolvimento, impele sua aplicação em projetos de desenvolvimento.

1.4 Objetivo

Portanto, para contornar a falta de suporte ao EPiC-Java em ambientes integrados de desenvolvimento e, assim, disponibilizar aos desenvolvedores uma maneira prática de utilizar o conceito de interfaces bidimensionais para exceções, este trabalho tem como objetivo apresentar a implementação do *EPiC4Eclipse*: uma ferramenta que funciona de forma integrada como um *plug-in*, na *IDE Eclipse*, através da qual o desenvolvedor poderá utilizar a sintaxe EPiC para estabelecer o fluxo das exceções por meio de anotações. As cláusulas *propagate*, então, são automaticamente convertidas em cláusulas *throws* correspondentes de forma dinâmica e transparente durante a codificação.

Neste contexto, o trabalho está dividido da seguinte forma: no Capítulo 2 discutiremos os conceitos relacionados aos canais de propagação de exceção, a extensão EPiC-Java e suas vantagens em relação à interface *throws*. Já no Capítulo 3 será abordada a ferramenta proposta, o EPiC4Eclipse, enunciando os conceitos relacionados à implementação como processamento de anotações, modificação da árvore sintática durante a compilação etc., além de demonstrar um cenário de funcionamento. Por fim, no Capítulo 4, serão debatidos os resultados, as limitações e os trabalhos futuros.

[2] Plataforma *opensource Java* disponível em <http://openjdk.java.net/>. Acesso em 18/02/2015.

2 EPCs

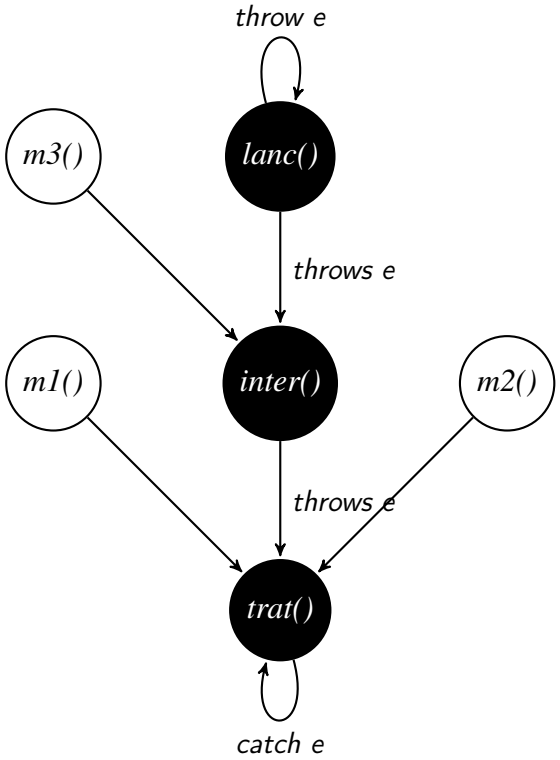
Neste capítulo iremos apresentar o conceito de EPC (Exception Propagation Channel) ou, em português, Canal de Propagação de Exceção, como descrito por SILVA; CASTOR. Esta definição é fundamental para estabelecer o sifnificado de fluxo de uma exceção, como ele será manipulado e quais são os componentes que fazem parte deste fluxo. Além disso, será apresentada a extensão à linguagem *Java*, *EPiC-Java*, a qual é a base da implementação deste trabalho, pois seus conceitos, objetivos de *design* originais (fácil manutenção, retrocompatibilidade etc.) e sintaxe para definição de EPCs serão implementados e preservados em sua essência. É importante salientar que, apesar deste conceito se aplicar também às não-checadas, iremos considerar nas próximas seções apenas as exceções checadas.

2.1 Definições

Podemos definir um canal de propagação de exceção como o o caminho na pilha de chamadas de métodos, desde o local de lançamento até o local de tratamento, se houver. Todos os locais, ou métodos, entre estes dois são chamados de locais intermediários. Locais de lançamento são, na prática, o ponto de origem: aqueles locais que propagam a exceção através da cláusula *throw*, onde é lançada a instancia de uma exceção, além de declará-la na assinatura. Já os locais intermediários são aqueles que apenas transferem a responsabilidade através da palavra-chave *throws* na assinatura do próprio método. Por fim, os locais de tratamento são aqueles que atribuem a responsabilidade para si, *capturando* a exceção em questão através do bloco *try-catch* e aplicando as regras necessárias para recuperar o sistema da situação excepcional. Em resumo, todo EPC é uma tupla envolvendo os conjuntos de: exceções que pertencem ao fluxo; locais de lançamento; locais intermediários; e locais de tratamento.

Como ilustração, podemos observar a Figura 3 que destaca um EPC trivial, assinalando todos seus componentes. O caminho em destaque possui todas as características de um EPC, considerando uma dada exceção *e*: 1) O método *lanc()* é o local de lançamento pois é a origem da propagação da exceção *e*, lançada em *Java* pela cláusula *throw*; 2) O método *inter()*, como é possível verificar, não realiza nenhum tipo de ação de tratamento da exceção *e*, apenas transferindo a responsabilidade para o método acima na pilha de chamadas, caracterizando-se apenas como local intermediário; 3) Por fim, temos o método *trat()* que é responsável por capturar *e* e definir uma lógica de tratamento.

Figura 3: Exemplo: Cadeia de chamadas de métodos com um EPC em destaque



2.2 EPiC-Java

Como vimos anteriormente, a maneira em que a propagação das exceções é definida em *Java* (cláusulas *throw*, *throws* e blocos *try-catch*) possui diversas limitações, tanto em relação à análise do código quanto, principalmente, à sua manutenção. Discernirmos, também, a natureza de um canal de propagação de exceção, composto por locais de lançamento, intermediação e tratamento. O trabalho de SILVA; CASTOR, então, definiu uma extensão à linguagem (*EPiC-Java*) com o objetivo de superar as limitações citadas implementando o conceito de EPCs através de expressões de propagação (cláusulas *propagate*). Essas cláusulas são responsáveis por definir claramente o fluxo/canal de propagação da exceção apenas explicitando a tupla correspondente, substituindo a utilização de cláusulas *throws*. Por exemplo, uma cláusula *propagate* possível para definir o EPC destacado na Figura 3 é a seguinte:

```
propagate e: lanc() -> inter() -| trat();
```

O primeiro elemento se refere a qual exceção está sendo propagada, no caso *e*. Logo após é definido qual é o fluxo da exceção referida, desde o local de lançamento *lanc()* até o de tratamento *trat()*. Ao analisar por partes, no trecho *lanc()* -> *inter()*, o operador **->** estabelece que a exceção em questão flui do método descrito no primeiro argumento *lanc()* até o descrito no segundo *inter()* independentemente se existe outros métodos entre eles na chamada. Já no trecho *inter()* -| *trat()*, o operador **-|** descreve que a exceção flui entre os métodos descritos nos argumentos mas ela é capturada e tratada no método descrito no segundo. A Tabela 1 descreve a

sintaxe EPiC-Java em detalhes.

Tabela 1: Descrição da sintaxe *EPiC-Java*

Sintaxe	**Descrição**
m1 **->** m2	Sendo m1 e m2 métodos descritos pelos seus nomes qualificados, o termo estabelece que há um fluxo da exceção dada entre *m1* e *m2* de forma direta ou indireta. Ou seja, basta existir um caminho na pilha de chamadas entre eles e que a exceção seja propagada até m2.
m1 **=>** m2	O mesmo que o operador **->**, com a restrição de que o método *m2* deve invocar *m1* diretamente.
m1 **-|** m2	Define a propagação de uma dada exceção entre *m1* até *m2*, diretamente ou indiretamente, sendo *m2* sempre um local de tratamento. Ou seja, *m2* trata a exceção localmente e não a propaga externamente ao escopo do método.
m1 **=|** m2	Mesmo critério do operador **-|**, com a restrição de que *m2* deve invocar *m1* diretamente.
{conj < I}.m	*conj* representa um conjunto de classes que implementam uma interface *I*, enquanto *m* representa o método herdado de *I*. Por questões de polimórficas e de herança, este termo tem como objetivo explicitar quais métodos fazem parte do canal de propagação. Se considerarmos apenas a interface *I*, devido à vinculação dinâmica em tempo de execução, qualquer implementação pode fazer parte do EPC. Ao usarmos essa sintaxe, torna-se possível estabelecer e garantir o EPC de antemão.
(m1 | m2)	Usado para determinar que o fluxo cruza tanto o método *m1* quanto *m2*. Por exemplo, o fluxo m0 –> (m1 | m2) –| m3 pode ser definido pelos dois fluxos a seguir: m0 –> m1 –| m3 e m0 –> m2 –| m3

Utilizando a sintaxe descrita na Tabela 1 agora podemos definir o fluxo da Figura 2, onde é ilustrada a visão restrita do fluxo de exceções usando *throws*, apenas em termos de cláusulas *propagate*:

1. `propagate e: a() => b() => c() => d() =| e();`

2. `propagate e: a() =| f();`

Podendo ainda serem simplificadas em:

```
propagate e: a() -| (e() | f());
```

Como podemos constatar, as cláusulas *propagate* explicitam o fluxo da exceção, habilitando o desenvolvedor a identificar em apenas um relance: qual a origem da exceção; onde ela é tratada; e por quais métodos ela cruza. Sendo assim, as limitações descritas na seção 1.2 são mitigadas:

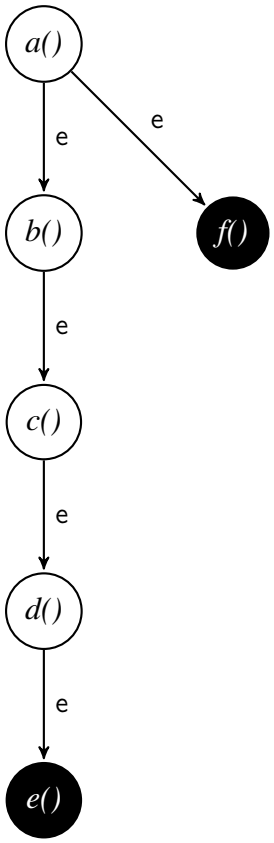
1. Análise

   Ao substituir a utilização de cláusulas *throws* na assinatura dos métodos pelas cláusulas *propagate*, observamos que o desenvolvedor adquire uma visão global (Figura 4) do fluxo ao invés de se ater à visão restritamente local.

2. Manutenção

   Durante a implementação ou evolução do *software*, é provável que algum dos fluxos de exceções seja modificado ou novos sejam adicionados. Também discutimos que editar ou adicionar exceções manualmente na assinatura dos métodos é uma atividade bastante suscetível a erros e que consome bastante tempo. Com a utilização da extensão EPiC-Java torna-se possível automatizar esta atividade apenas editando a declaração da cláusula *propagate* e todas as interfaces/assinaturas envolvidas serão atualizadas. Além disso, as cláusulas *propagate* podem ser declaradas num escopo mais geral, agregando por fluxos relacionados ou qualquer outro critério, facilitando a manutenção.

Figura 4: Representação da visão global consequente do uso de cláusulas *propagate*.



Para examinar a extensão, o SILVA; CASTOR também implementou um compilador protótipo[1], baseado no *javac* do *OpenJDK* que, ao acionado sobre um projeto *Java*, analisava a árvore sintática afim de analisar os fluxos das exceções e transformar as cláusulas *throws* correspondentes em *propagate*. Ainda foi realizado um diagnóstico sobre a utilização de EPiC-Java em projetos pré-existentes como o Rhino [2], onde foi possível observar que era possível substituir 60 (sessenta) cláusulas *throws* em apenas 4 (quatro) *propagate*, evidenciando seu impacto potencial no desenvolvimento e manutenção de *software*.

Porém, a utilização da extensão EPiC-Java por desenvolvedores em projetos reais é impraticável, pois: 1) a solução é baseada numa sintaxe que não faz parte da especificão de qualquer versão oficial da linguagem *Java*; 2) necessita um compilador modificado, forçando desenvolvedores a utilizá-lo; 3) não oferece suporte a ambientes de desenvolvimento integrados (*IDE*s), desperdiçando vantagens durante a codificação como *auto-completion*, análise estática *on-the-fly*, *build* automática etc. (CHEN; MARX, 2005). Portanto, fez-se necessário a criação de uma ferramenta que superasse essas limitações, habilitando o uso da extensão durante o desenvolvimento: o *plug-in* para *Eclipse*: o *EPiC4Eclipse*, que será descrito no capítulo a seguir.

[1] Disponível em: <https://github.com/thiago-silva/epic-java>. Acesso em 18/02/2015.
[2] Disponível em: <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>. Acesso em 18/02/2015.

3 EPiC4Eclipse

O *EPiC4Eclipse* surgiu da necessidade de verificar a utilização da extensão *EPiC-Java*, proposta no trabalho de SILVA; CASTOR, dentro de um ambiente de desenvolvimento integrado, de maneira a possibilitar seu uso durante o processo de construção de *software* de forma transparente, sem necessidade de etapas adicionais de refatoramento por ferramentas externas. *Eclipse*, desenvolvida pela *Eclipse Foundation*[1], foi a *IDE* escolhida para implementação da ferramenta, motivado por diversas vantagens, como:

1. APIs de acesso à árvore sintática abstrata que permitem o desenvolvimento de *plug-ins* para manipular o código (HOU, 2007);
2. Ampla utilização na indústria, marcada pela história por uma das plataformas de desenvolvimento dominantes para *Java* (GEER, 2005);
3. Possui código aberto.

Já no que se refere ao projeto de implementação do *plug-in*, vale salientar que foram levados em consideração todos os objetivos de projeto originais de *EPiC-Java*, preservando-os como pilares do *EPiC4Eclipse*, sendo eles:

1. Facilidade de Manutenção

   Deve ser possível modificar, criar ou eliminar objetivamente fluxos de exceções sem o retrabalho manual sobre os métodos ao longo da propagação. Declarações devem ser compactas, legíveis e representativas tanto no âmbito horizontal como no vertical afim de permitir essa propriedade. Por fim, qualquer alteração deve refletir *on-the-fly* no código.

2. Retrocompatibilidade com cláusulas *throws*

   A extensão não foi concebida com a finalidade de substituir em totalidade cláusulas *throws* Java-nativas, e sim para complementá-las. Assim, mantendo a compatibilidade com cláusulas *throws*, a refatoração de um código já existente pode ser feita gradativamente, por exemplo. Além disso, é importante evidenciar que durante a pré-compilação, todas as cláusulas *throws* deverão ser convertidas em *throws* para o código ser compilado nativamente.

3. Analisabilidade estática

   Uma das principais vantagens da adoção de exceções checadas é a viabilidade da análise estática do código, pois considerar também exceções de execução tornaria a análise

[1] Homepage: <https://eclipse.org/>

demasiadamente complexa (CHANG; JO; HER, 2002). Com isso, é possível prever situações excepcionais de antemão, sem necessidade de execução do código. Portanto, esta vantagem essencial não pode ser prejudicada com a adaptação das cláusulas *propagate*.

4. Visão global

   Como observado anteriormente, cláusulas *throws* fornecem apenas uma visão local da propagação, enquanto que na maioria dos casos, a exceção deve ser tratada num contexto distante do local de lançamento, se consideramos a pilha de chamadas de métodos. Logo, a bidimensionalidade proporcionalizada pela utilização de *propagate* deve ser garantida após adaptação.

3.1 Execução

Para demonstrar como o *plug-in* funciona, o Cód. 3 e Cód. 4 representam o desenvolvimento de um *software* sem e com o Epic4Eclipse, respectivamente. Os trechos de código a seguir simulam uma estrutura MVC[2] para ilustrar a porção de um *software* que imprime na tela todos os nomes dos clientes presentes no banco de dados.

Código 3: Trecho de código utilizando cláusulas *throws*

```
1  public class Application extends UserInterface{
2
3    public class GUI{
4        public void showNames(){
5            try{
6                for(String name : Controller.searchAllNames()){
7                    print(name);
8                }
9            }catch(TimeOutException | SecurityException | DBException e){
10               showError(e.getMessage());
11           }
12       }
13   }
14
15   public static class Controller{
16       public static List<String> searchAllNames() throws TimeOutException,
       SecurityException, DBException{
17           return Facade.getAllNames();
18       }
19   }
20
21   public static class Facade{
22       public static List<String> getAllNames() throws TimeOutException,
       SecurityException, DBException{
```

[2] Model-View-Controller

```
23            return NamesCollection.getAll();
24        }
25    }
26
27    public static class NamesCollection{
28        public static List<String> getAll() throws TimeOutException,
      SecurityException, DBException{
29            return DB.query("SELECT name FROM clients");
30        }
31    }
32
33    public static class DB{
34        public static List<String> query(String query) throws
      TimeOutException, SecurityException, DBException{
35            //...
36        }
37    }
38
39 }
```

Como podemos observar no Cód. 3, uma representação trivial de um software já revela alguns dos problemas em relação a adoção das cláusulas *throws* como citado na seção 1.2. Por exemplo, se considerarmos uma possível refatoração do código para adicionar uma exceção ao método DB.query(String), as assinaturas de NamesCollection.getAll(), Facade.getAllNames() e Controller.searchAllNames() também teriam de ser modificadas manualmente. Dada a devida proporção, como esta atividade também envolve a análise do fluxo das exceções pelas diversas partes do software, torna-se custoso realizar essa tarefa em um código razoavelmente extenso.

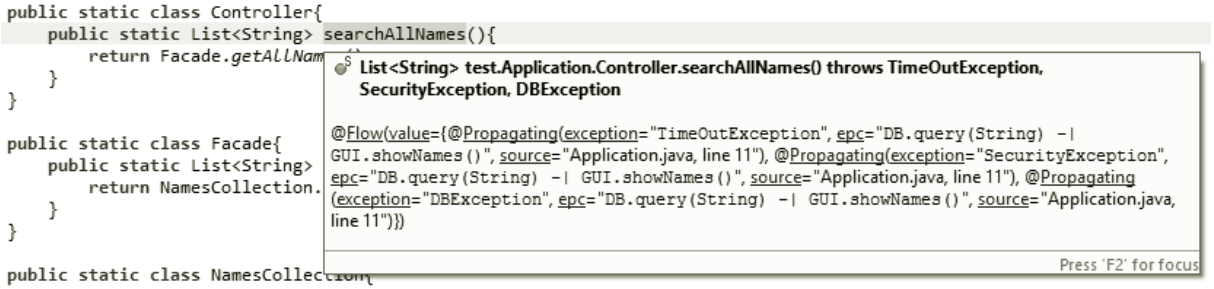
Código 4: Desenvolvimento com cláusulas @Propagate

```
1  @Epic
2  public class Application extends UserInterface{
3
4    public class GUI{
5      @Propagate("TimeOutException, SecurityException, DBException: DB.query(
       String) -| GUI.showNames()")
6        public void showNames(){
7            try{
8                for(String name : Controller.searchAllNames()){
9                    print(name);
10               }
11           }catch(TimeOutException | SecurityException | DBException e){
12               showError(e.getMessage());
13           }
14       }
15   }
16
```

```
17    public static class Controller{
18        public static List<String> searchAllNames(){
19            return Facade.getAllNames();
20        }
21    }
22
23    public static class Facade{
24        public static List<String> getAllNames(){
25            return NamesCollection.getAll();
26        }
27    }
28
29    public static class NamesCollection{
30        public static List<String> getAll(){
31            return DB.query("SELECT name FROM clients");
32        }
33    }
34
35    public static class DB{
36        public static List<String> query(String query){
37            //...
38        }
39    }
40
41 }
```

Utilizando as anotações suportadas pelo EPiC4Eclipse, como exposto no Cód. 4, e com o suporte oferecido pela IDE, todas as cláusulas *throws* foram omitidas no código-fonte, sendo inseridas automaticamente durante o processamento das anotações *@Propagate* de forma transparente ao desenvolvedor. Dessa forma, uma possível refatoração para adicionar uma exceção ao fluxo seria realizada imediatamente apenas inserindo-a na cláusula *propagate*. Além disso, para suprir a falta de informações sobre o fluxo, podemos verificar quais exceções um método propaga apenas selecionando o método em questão para que o Eclipse exponha as propagações envolvidas, como na Figura 5.

Figura 5: *Helper* do Eclipse contendo o fluxo das exceções no método



3.2 Implementação

O primeiro objetivo-base do trabalho foi encontrar uma forma de adaptar a sintaxe *EPiC-Java* para que o compilador reconheça cláusulas *propagate* sem necessidade de alterá-lo, e ainda evitar alterações significativas na estrutura da cláusula. Assim, será possível ultrapassar as limitações do protótipo apresentado no trabalho de SILVA; CASTOR. Portanto, para alcançar este objetivo, o conceito de anotações (aderido desde a versão 1.5 de *Java*) foi adotado, pois provê dados sobre o próprio código onde é inserido. Pode-se destacar que a principal justificativa para utilização de anotações baseia-se num dos objetivos de *EPiC-Java*: a analisabilidade estática, visto que são amplamente utilizadas por ferramentas deste contexto para descrever propriedades adicionais acerca do software que não podem ser facilmente inferidas (BESSEY et al., 2010), o que se adequa perfeitamente ao contexto do projeto. Além disso, anotações tornam as cláusulas *propagate* mais "*Java-like*", fazendo com que sejam mais legíveis aos programadores, encorajando o uso (FLANAGAN et al., 2002).

3.2.1 Anotações

3.2.1.1 Conceitos

Como descrito na especificação oficial[3] de *Java*, anotações são capazes de prover informações adicionais para o compilador, além de permitir pré-processamento através de APIs para gerar código, arquivos XML etc., além de possibilitar a análise durante a execução do programa. Como identificação, deve-se utilizar o caracter @ para indicar que o próximo elemento se refere a uma anotação, como por exemplo:

```
@Override
```

Anotações podem ser inseridas em qualquer lugar, desde que se associe a quaisquer tipos de declarações, porém é convenção que ela tenha sua própria linha no código. Podemos enumerar, então, algumas situações comuns em relação ao posicionamento e tipos de declarações:

1. Declarações de Classe

```
1  @Author("Filipe")
2  class Test{
3      //...
4  }
```

2. Declarações de Método

```
1  @Test(timeout=1000)
2  public void testEpic(){
3      //...
4  }
```

[3] Disponível em: <http://docs.oracle.com/javase/specs/>. Acesso em 18/02/2015.

3. Declaração de Variáveis

```
1     @XmlValue String attribute;
```

Além da classificação em relação a onde podem ser declaradas, também podemos categorizá-las quanto aos parâmetros que podem receber:

1. *NormalAnnotation*

   Caracteriza-se pela definição mais geral, permitindo a declaração de diversos pares identificador-valor como parâmetros, tal como no exemplo @Test(timeout=1000). Podemos classificar @Test como *NormalAnnotation* já que ela recebe ao menos um par identificador (timeout) / valor (1000ms).

2. *MarkerAnnotation*

   Anotações que não recebem nenhum parâmetro, constituindo um *syntax-sugar* para *NormalAnnotation* com um conjunto de pares identificador-valor vazio. No caso do @XmlValue, também poderia ser usado @XmlValue().

3. *SingleElementAnnotation*

   Ao utilizar o identificador value numa anotação, esta pode ser omitida nos parâmetros caso não haja outros identificadores ou se todos os outros possuem um valor *default*. Por exemplo, @Author: é uma anotação que possui um identificador value que recebe um valor do tipo String, mas que pode ser facilmente omitido como mm @Author("Filipe") que poderia ser substituído por @Author(value="Filipe").

Para criar uma anotação, segue-se um padrão semelhante à criação de classes, mas ao contrário da *keyword* class, utiliza-se uma anotação própria para declaração: @Interface. Anotações podem ainda serem incrementadas por outras anotações, como @Target, que são responsáveis por determinar a quais tipos elas podem ser associadas (classes, pacotes, métodos, variáveis etc.), e @Retention que delimita se a anotação estará presente somente no código-fonte ou se também estará presente no código binário após compilação, com a clara vantagem de permitir acesso via reflexão [4]. Conscientes desses conceitos, podemos exemplificar possíveis estruturas das anotações citadas:

1. @Author

```
1     @interface Author{
2         String value();
3     }
```

[4] Disponível em: <http://docs.oracle.com/javase/tutorial/reflect/>. Acesso em 18/02/2015.

2. @Test

```
@Retention(value=RUNTIME)
@Target(value=METHOD)
@interface Test{
    Class<? extends Throwable> expected default None.class;
    int timeout() default 0;
}
```

Aqui verificamos um típico caso de uso das anotações auxiliares @Retention e @Target, as quais definem que @Test que podem ser acessadas em tempo de execução e devem ser associadas somente à assinaturas de métodos, respectivamente.

3. @XmlValue

```
@interface XmlValue{
}
```

3.2.1.2 @Propagate e @Epic

Para o EPiC4Eclipse, foram criadas duas anotações básicas: @Propagate e @Epic, definidas a seguir:

Código 5: Implementação da anotação @Propagate

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
public @interface Propagate {
    String value() default "";
}
```

Código 6: Implementação da anotação @Epic

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Epic {
  Propagate[] value() default {};
}
```

A primeira foi planejada com a intenção de concretizar as cláusulas *propagate*, recebendo o fluxo como parâmetro do tipo String. Já a segunda torna-se necessária em versões de *Java* anteriores à 1.8, pois em nenhuma delas é possível adicionar uma mesma anotação múltiplas vezes no mesmo trecho de código. Para contornar esse problema, @Epic recebe como parâmetro um conjunto de anotações @Propagate, permitindo, dessa forma, múltiplas declarações sem a preocupação com compatibilidade. Assim, as cláusulas:

```
propagate SampleException: lanc() -> inter() -| trat();
propagate OtherException: lanc2() -| trat2();
```

Usando EPiC4Eclipse, podem ser adaptadas para:

```
@Epic({
    @Propagate("SampleException: lanc() -> inter() -| trat()"),
    @Propagate("OtherException: lanc2() -| trat2()")
})
```

3.2.2 Processamento

Processadores de anotações tem a função de analisar anotações pré-especificadas que estejam presentes no código, através dos quais é possível validar código, gerar arquivos adicionais etc. ainda no processo de construção/compilação do software. Como especificado oficialmente [5], estes processadores, previamente implementados, são adicionados à configuração do compilador através de: referência direta; procura numa lista processadores aptos à tratar a anotação em questão etc. Quando executados na infraestrutura fornecida pelo ambiente de desenvolvimento, processam as anotações através de *rounds* aninhados, nos quais um subconjunto de anotações encontradas no código podem ser analisadas. Os *rounds*, apesar de serem consecutivos também podem tratar anotações já processadas em *rounds* anteriores.

Para implementar o processamento de anotações, deve-se criar uma classe que implemente a interface Processor[5] ou estender a classe AbstractProcessor[6] (opção comumente utilizada por desenvolvedores). Para exemplificar, segue no Cód. 7 uma implementação simples de um processador de anotações *@Author*, que salva num arquivo *log* todos os autores que tiveram algum de seus códigos compilado e a data correspondente para posterior auditoria.

Código 7: Exemplo de processador de anotações @Author

```
@SupportedAnnotationTypes("br.ufpe.cin.Author")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class AuthorProcessor extends AbstractProcessor {
    //..
    public boolean process(Set<? extends TypeElement> elements,
                           RoundEnvironment roundEnv) {
        for (TypeElement te : elements) {
            for (Element e : roundEnv.getElementsAnnotatedWith(te)) {
                Author author = e.getAnnotation(Author.class);
                Date today = new Date();
                LogWriter.appendToFile(author.value() + ", on:" + today);
            }
        }
        return false;
    }
}
```

[5] Disponível em: <http://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html>. Acesso em 18/02/2015.

[6] Disponível em: <http://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/AbstractProcessor.html>. Acesso em 18/02/2015.

Avaliando o Cód. 7 na linha 1, observamos que a anotação *@SupportedAnnotationTypes* indica quais anotações serão suportadas pelo processador anotado, podendo ainda remover essa limitação através do *wildcard* *. Já na linha 2, *@SupportedSourceVersion* estabelece o limite mínimo da versão de *Java* que o processador deve ser executado para evitar problemas de compatibilidade na execução, já que algumas funções podem ser dependentes da versão. Adicionalmente, o valor inserido pode ser consultado através da função `getSupportedSourceVersion()` de *AbstractProcessor*.

Apesar de consideramos apenas o processador *AuthorProcessor*, vários processadores podem executar durante a pré-compilação. Processadores executados simultaneamente podem gerar ou tratar ãnotações já manipuladas, o que acarretaria por exemplo geração de código duplicado ou contraditório. Para evitar essa situação, podemos informar ao ambiente que as anotações processadas no último *round* por um determinado processador foram reinvicadas e não devem ser propagadas adiante. Esta reinvidicação é representada pelo retorno do método `process(Set<? extends TypeElement>, RoundEnvironment)`: caso *false*, as anotações serão propagadas; caso contrário, as anotações processadas não serão visíveis por outros processadores.

Para que o processador de anotações seja executado, é preciso informar ao compilador qual o caminho de cada um. Há duas formas de especificar essa informação:

1. Como argumento explícito na compilação

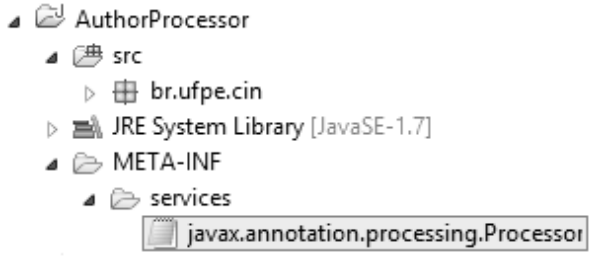
   Em muitas situações, principalmente quando IDEs não são utilizadas, torna-se vantajoso especificar os processadores a serem executados via argumento do comando de compilação. Segue um exemplo utilizando o *javac*:

```
1 javac -processor br.ufpe.cin.AuthorProcessor <source-file>
```

2. Registro em META-INF

   Esta opção consiste em registar o processador de anotações como serviço para que o compilador e até outras ferramentas/IDEs possam encontrá-los automaticamente. Primeiramente deve ser criado um arquivo no diretório `<RAIZ_DO_PROJETO>/META-INF/services` nomeado *javax.annotation.processing.Processor*, como apresentado na Figura 6 a seguir.

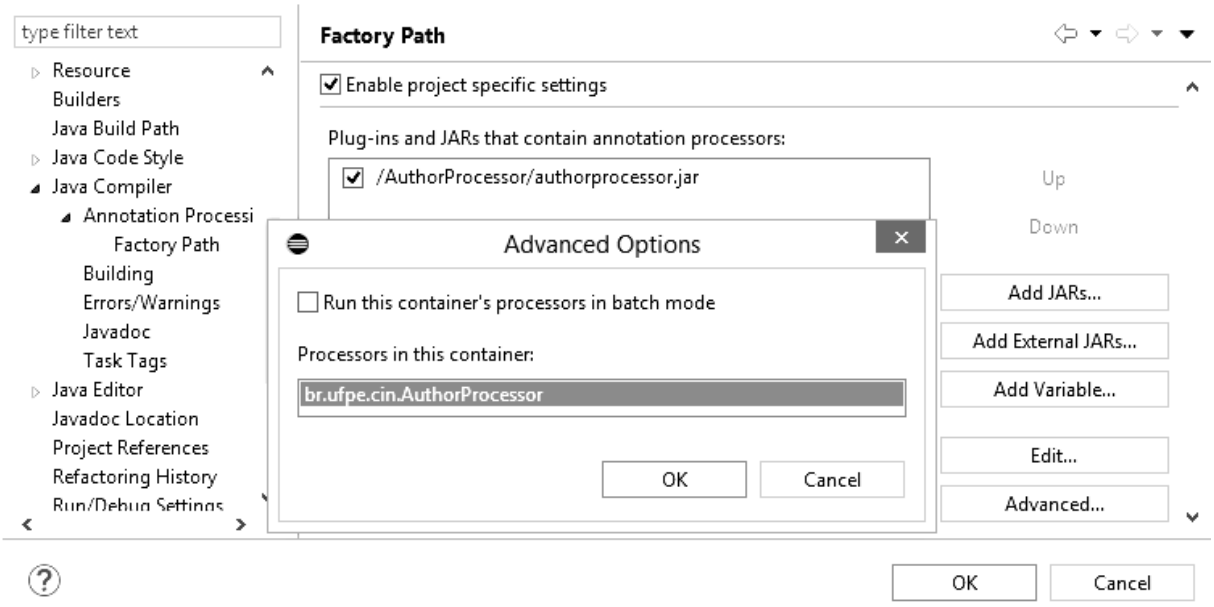
Figura 6: Registro do processador de anotações



Fonte: Screenshot do autor

No conteúdo do arquivo recém-criado, deve ser escrito o nome plenamente qualificado de quais processadores devem ser executados, separados por quebra de linha. Nesse caso, apenas br.ufpe.cin.AuthorProcessor. Após esses passos, usando *javac*, basta adicionar o projeto ao *classpath*. Porém, para utilizar o ambiente de processamento do *Eclipse*, o projeto exportado deve ser adicionado explicitamente às configurações do projeto como ilustrado na Figura 7 a seguir.

Figura 7: Configuração do processador de anotações no Eclipse



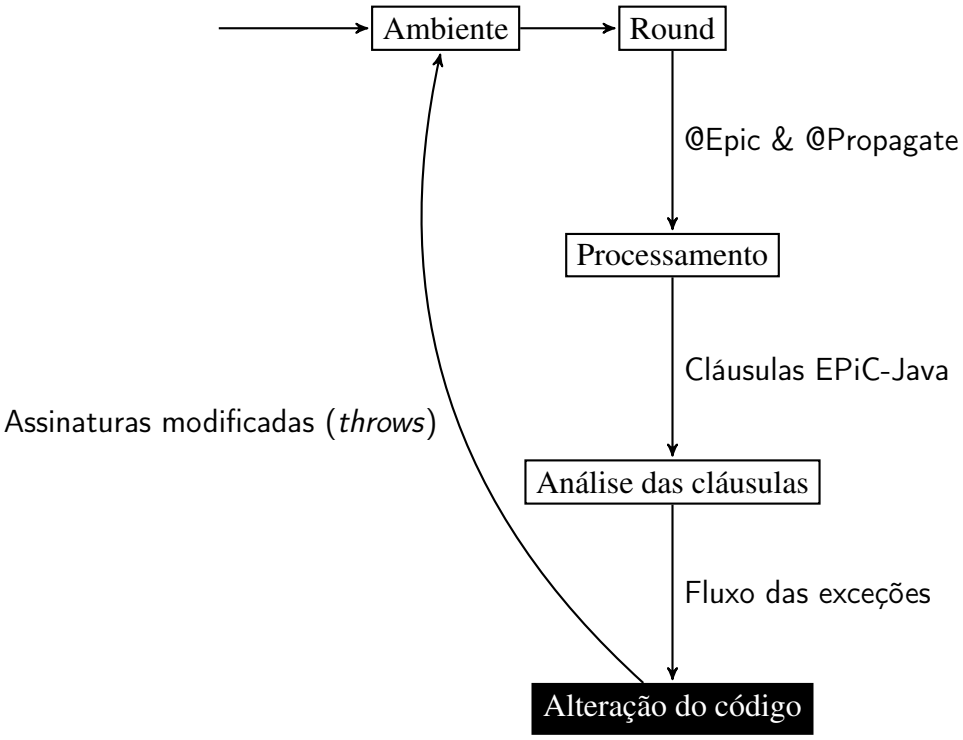
Fonte: Screenshot do autor

3.2.2.1 EpicProcessor

A ideia por trás do processador de anotações *@Epic* e *@Propagate* é reconhecer os fluxos de exceções para automatizar a alteração das cláusulas *throws* na assinatura dos métodos envolvidos. Seguindo a Figura 8, observamos que o para atingir o objetivo deve-se alterar o código durante o processamento das anotações.

Porém, o conceito de anotações foi originalmente introduzido para adicionar somente informações sobre o software em desenvolvimento, podendo até gerar código extra, mas não alterando a semântica do código em si (ARNOLD et al., 1996). Por esse motivo, processadores de anotações disponíveis pela plataforma não disponibilizam APIs diretas para alteração do código *Java*, limitando o processamento descrito na Figura 8 pois a fase de modificação do código é restringida. Contudo, para contonar essa situação, desenvolvedores criaram o Projeto Lombok descrito a seguir.

Figura 8: Diagrama do processamento de anotações esperado para o EPiC4Eclipse



3.2.3 Projeto Lombok

O Projeto *Lombok* [7] tem o objetivo de estender a linguagem *Java* através de anotações para remover código *boilerplate* (trechos de código que são usualmente distribuídos em diversas partes do software pelo mesmo propósito, comum em linguagens verbosas como *Java*) por meio da injeção de código. Para tanto, *Lombok* toma a frente durante a fase de compilação acessando e modificando a árvore sintática, usando interfaces internas descritas pela *Pluggable Annotation Processing API* (JSR269[8]) (KLEPININ; MELENTYEV, 2011).

Lombok disponibiliza um mecanismo semelhante ao *AbstractProcessor*: cada anotação deve possuir um manipulador, ou *Handler*, associado para processá-la. Como o projeto utiliza interfaces internas não-públicas do compilador, deve ser implementado um manipulador tanto para o compilador *Eclipse* (ecj) quanto para o *javac*. Segue abaixo um exemplo de um manipulador para a anotação *@Epic* no Eclipse:

Código 8: Estrutura do *Handler* para @Epic usando o projeto *Lombok*

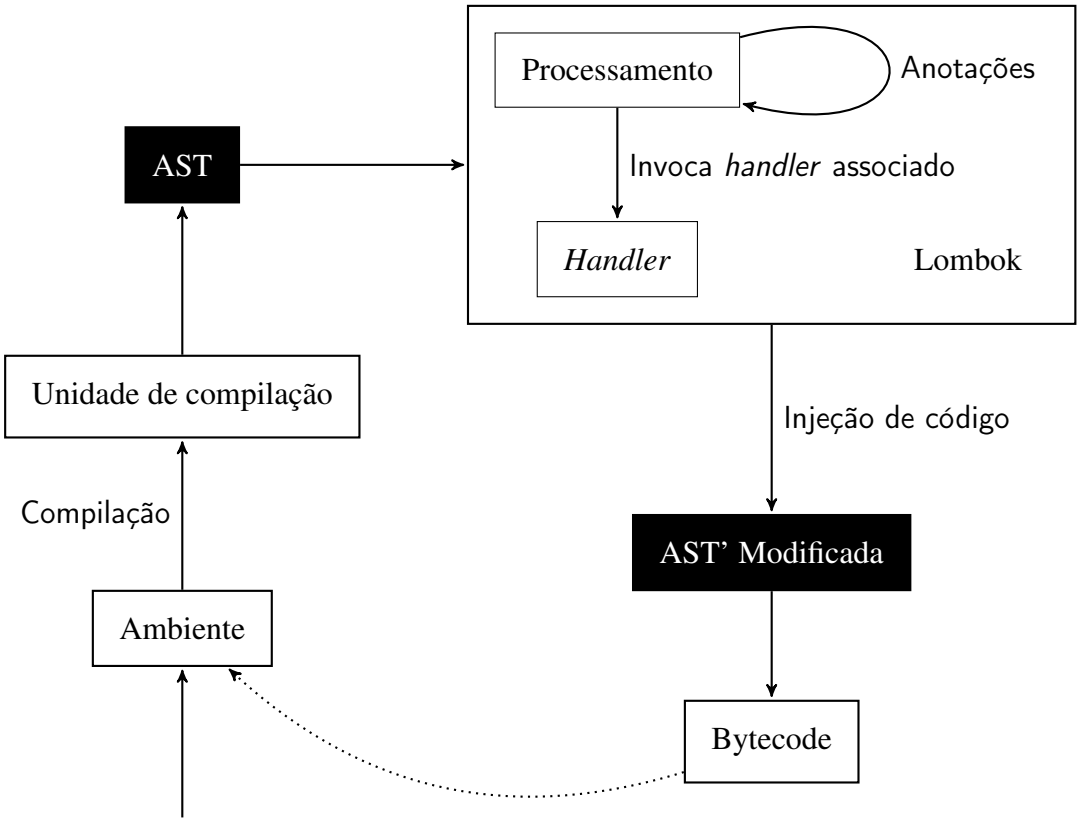
```
1 @ProviderFor(EclipseAnnotationHandler.class)
2 public class HandleEpic extends EclipseAnnotationHandler<Epic> {
3
4   @Override public void handle(AnnotationValues<Epic> annotation,
      Annotation ast, final EclipseNode annotationNode) {
5     //...
6   }
```

[7] Disponível em <projectlombok.org>. Acesso em 17/01/2015
[8] Disponível em <https://www.jcp.org/en/jsr/detail?id=269>. Acesso em: 08/02/2015

```
7 }
```

Cada *Handler* é invocado pelo ambiente especificamente para as anotações que ele é associado, semelhante ao mecanismos de *rounds* do processador de anotações que identifica quais anotações trata através de *@SupportedAnnotationTypes*.

Figura 9: Diagrama simplificado do processamento de anotações usando *Lombok*



De acordo com a Figura 9, assim que é realizada uma ordem de compilação, cada código fonte (unidade de compilação) é transformado em uma estrutura de árvore, a árvore sintática abstrata, a qual é usada como entrada para o processamento *Lombok*, que procura pelos nós correspondentes às anotações e invoca o *Handler* associado. É no *handler* que podemos utilizar a *API* disponibilizada pela biblioteca para injeção de código, modificando a AST original e retornando-a para o processo de análise e geração de código *bytecode*. Esse *bytecode*, enfim, pode ser utilizado pelo ambiente integrado de desenvolvimento para apurar as assinaturas dos métodos: exceções que lança, parâmetros, retorno etc.

3.2.4 AST

A análise da árvore sintática abstrata (AST, ou em inglês, Abstract Syntax Tree) é o ponto crucial para EPiC4Eclipse pois é através dela que se torna possível analisar e alterar o fluxo das exceções. A AST é nada mais que a representação de um código-fonte *Java* numa estrutura de árvore, permitindo análise apurada do código e é base de boa parte das *features* do

Eclipse (HOU, 2007), pois os nós da árvore também são objetos e representam tanto hierarquia de classe como de composição (EKMAN; HEDIN, 2007). A plataforma *Eclipse* implementou sua própria representação da *AST*, disponibilizando sua manipulação através da *framework* JDT (*Java Developer Tools*). Para percorrer a estrutura da AST, podemos estender a classe *ASTVisitor*[9]. Segue abaixo um exemplo:

Código 9: Estendendo *ASTVisitor*

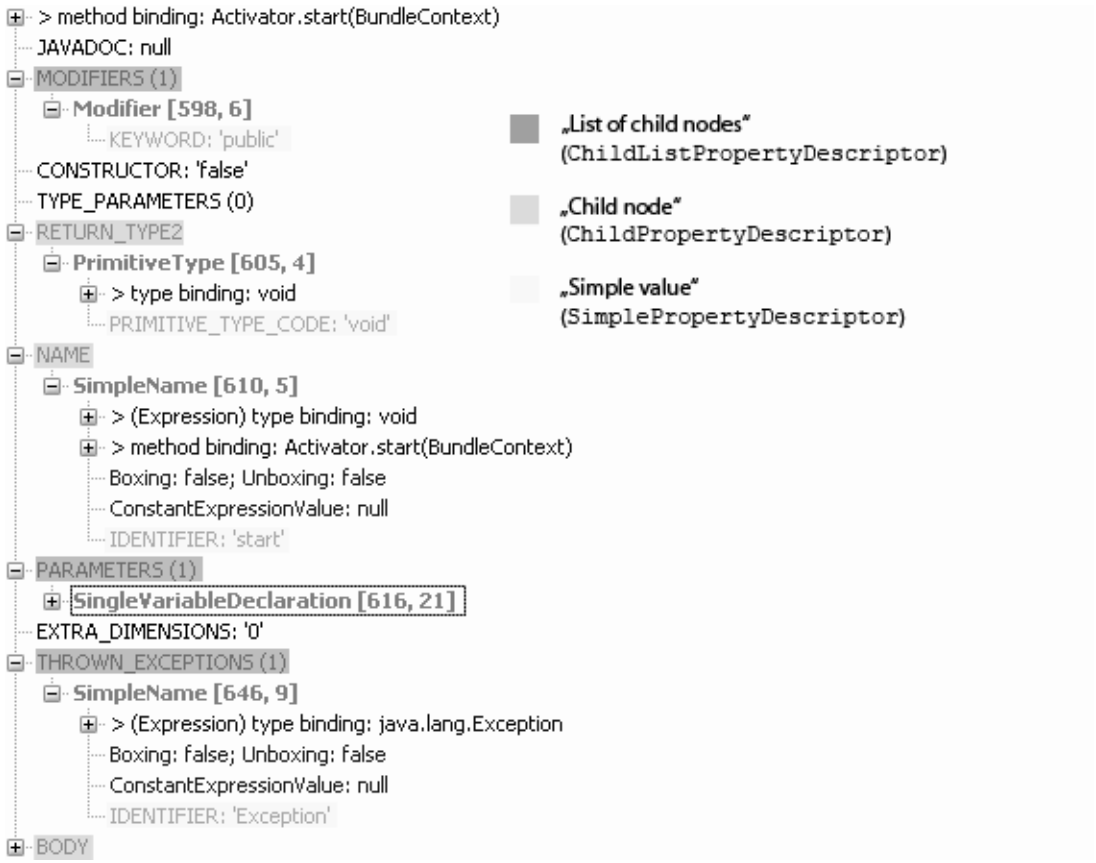
```
1  public class AnalyzerEclipse extends ASTVisitor {
2
3    public AnalyzerEclipse() {}
4
5
6    @Override public boolean visit(MethodDeclaration node) {
7        //...
8    }
9
10   @Override public boolean visit(SingleMemberAnnotation node) {
11       //...
12   }
13
14   @Override public boolean visit(MethodInvocation node) {
15       //...
16   }
17
18 }
```

Antes da alteração da AST, primeiramente é necessário analisar a chamada de métodos envolvidos para investigar se o fluxo descrito por determinada cláusula *propagate* efetivamente existe. Conforme o Cód. 9, o *EPiC4Eclipse* armazena dados sobre as declarações de método (estruturada na Figura 10 e cruza essas informações com as invocações correspondentes. Consequentemente, quando uma cláusula *propagate* é analisada, o primeiro método da propagação é procurado. Assim que encontrado, como foi realizada uma pré-análise na qual foram armazenados os dados sobre invocações de métodos, é possível rastrear as chamadas e verificar se os métodos seguintes da cláusula de fato formam um fluxo correto. Verificada essa condição, todas as referências são mantidas para posterior modificação da AST.

Após a análise da AST e com as propagações checadas, agora é necessário reescrever os nós de acordo com o fluxo proveniente das cláusulas *propagate*. Essencialmente, são alteradas as assinaturas dos métodos envolvidos adicionando as cláusulas *throws* correspondentes (através da API fornecida pelo projeto *Lombok*). Complementamente, para auxiliar o desenvolvedor, torna-se necessário indicar quais exceções um método propaga e a quais cláusulas *propagate* ele está relacionado. Como a ferramenta é baseada na alteração da AST, apenas nós pertencentes

[9] Disponível em <http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.isv/reference/api/org/eclipse/cdt/core/dom/ast/ASTVisitor.html>. Acesso em: 08/02/2015

Figura 10: Estrutura de um nó *MethodDeclaration*



Fonte: Artigo Eclipse

à estrutura da árvore sintática podem ser adicionados. Por isso, comentários e documentação *javadoc* não são alternativas. Portanto, como as anotações são representados por nós da AST, *EPiC4Eclipse* adiciona anotações *@Propagating* em cada método envolvido, permitindo que essa informação seja encontrada por ferramentas nativas da IDE, indicando: 1) exceção envolvida 2) EPC correspondente 3) Local no código-fonte onde foi declarada a cláusula *propagate*.

4 Considerações finais

O EPiC4Eclipse conseguiu concretizar seus objetivos através da transformação das cláusulas *propagate*, adaptadas por anotações, em cláusulas *throws* por meio da manipulação da árvore sintática durante a compilação, de forma transparente ao desenvolvedor e integrada à IDE *Eclipse* com alterações realizadas *on-the-fly*. Ao contrário de trabalhos como o *EJFlow* (CACHO et al., 2008; CASTOR et al., 2009) que são focados na separação de interesses e associados à orientação a aspectos, EPiC4Eclipse implementa uma abordagem totalmente independente da "aspectização", podendo ser utilizada sobre qualquer código Java, pois é baseada apenas em anotações. Já em relação às ferramentas de análise estática, como a *Jex* do trabalho de ROBILLARD; MURPHY ou a implementada no trabalho de CHANG; JO; HER, elas apenas provêm informações sobre os fluxos das exceções mas não garantem a conformidade com especificação e nem oferecem uma maneira direta para manutenção, constituindo-se apenas como ferramentas de auxílio informativo ao desenvolvedor.

Quanto às limitações do EPiC4Eclipse, podemos observar que apesar do ambiente Eclipse, como já citado, ser amplamente utilizado por desenvolvedores, a solução é atualmente restrita à plataforma, o que pode reduzir sua adoção. Em relação à performance, como as cláusulas *throws* envolvem métodos que frequentemente são de diferentes unidades de compilação (arquivo código-fonte), há um certo *overhead* pois é preciso transformar a AST de todas as classes envolvidas, além de afetar a sincronização dessas. Adicionalmente, apesar das anotações *@Propagating* serem suficientes para apresentar o fluxo sobre determinado método, a solução não oferece boa legibilidade quando envolve diversas cláusulas. Para contornar esse problema, considerou-se utilizar comentários, mas estes não são representados como nós na árvore sintática, constituindo-se, então, como uma solução inviável.

Como trabalho futuro podemos destacar a portabilidade do EPiC-Java para outras IDEs, pois é possível adaptar a ferramenta para outros ambientes que utilizam o *javac* como compilador padrão ajustando apenas as fases que manipulam a AST, pois os compiladores *ecj* e *javac* as representam de formas distintas. Fora essa adaptação, todas as etapas restantes de reconhecimento das cláusulas e verificação da propagação continuam as mesmas. Também podemos desenvolver uma análise do fluxo mais sofisticada, mas que deve ser ponderada para não prejudicar consideravelmente a performance. Por fim, após termos verificado a correspondência entre os requisitos iniciais do projeto e as características atuais da ferramenta, torna-se necessário validá-la junto a desenvolvedores para designar suas vantagens/desvantagens na prática. Para isso, pretende-se realizar um experimento controlado com alunos do Centro de Informática da UFPE.

Referências

ARNOLD, K. et al. *The Java programming language*. [S.l.]: Addison-wesley Reading, 1996. Citado 2 vezes nas páginas 7 e 25.

BESSEY, A. et al. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, ACM, v. 53, n. 2, p. 66–75, 2010. Citado na página 20.

BLOCH, J. How to design a good api and why it matters. In: ACM. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. [S.l.], 2006. p. 506–507. Citado na página 7.

CACHO, N. et al. Ejflow: taming exceptional control flows in aspect-oriented programming. In: ACM. *Proceedings of the 7th international conference on Aspect-oriented software development*. [S.l.], 2008. p. 72–83. Citado 3 vezes nas páginas 9, 10 e 30.

CAMPIONE, M. *The Java tutorial: a short course on the basics*. [S.l.]: Addison-Wesley Professional, 2001. Citado na página 7.

CASTOR, F. et al. On the modularization and reuse of exception handling with aspects. *Software: Practice and Experience*, Wiley Online Library, v. 39, n. 17, p. 1377–1417, 2009. Citado na página 30.

CHANG, B.-M.; JO, J.-W.; HER, S. H. Visualization of exception propagation for java using static analysis. In: IEEE. *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*. [S.l.], 2002. p. 173–182. Citado 2 vezes nas páginas 17 e 30.

CHEN, Z.; MARX, D. Experiences with eclipse ide in programming courses. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 21, n. 2, p. 104–112, 2005. Citado na página 15.

EKMAN, T.; HEDIN, G. The jastadd extensible java compiler. *ACM Sigplan Notices*, ACM, v. 42, n. 10, p. 1–18, 2007. Citado na página 28.

FLANAGAN, C. et al. Extended static checking for java. v. 37, n. 5, p. 234–245, 2002. Citado na página 20.

GEER, D. Eclipse becomes the dominant java ide. *Computer*, IEEE, v. 38, n. 7, p. 16–18, 2005. Citado na página 16.

HOU, D. Studying the evolution of the eclipse java editor. In: ACM. *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. [S.l.], 2007. p. 65–69. Citado 2 vezes nas páginas 16 e 28.

KLEPININ, A.; MELENTYEV, A. Integration of semantic verifiers into java language compilers. *Automatic Control and Computer Sciences*, Springer, v. 45, n. 7, p. 408–412, 2011. Citado na página 26.

ROBILLARD, M. P.; MURPHY, G. C. Designing robust java programs with exceptions. v. 25, n. 6, p. 2–10, 2000. Citado 2 vezes nas páginas 7 e 9.

ROBILLARD, M. P.; MURPHY, G. C. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 12, n. 2, p. 191–221, 2003. Citado 2 vezes nas páginas 8 e 30.

SHAH, H.; GÖRG, C.; HARROLD, M. J. Visualization of exception handling constructs to support program understanding. In: ACM. *Proceedings of the 4th ACM symposium on Software visualization*. [S.l.], 2008. p. 19–28. Citado na página 9.

SILVA, T. B.; CASTOR, F. New exception interfaces for java-like languages. In: ACM. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. [S.l.], 2013. p. 1661–1666. Citado 7 vezes nas páginas 7, 10, 11, 12, 15, 16 e 20.