Mestrado em Engenharia Informática
Dissertação
Relatório Final

Extensão à Linguagem Groovy

Carlos Cortinhas
ccort@student.dei.uc.pt

Orientador:
Fernando Barros
Data: 10 de Julho de 2012

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Resumo

O trabalho aqui reportado refere-se aos métodos e mecanismos estudados e implementados para extender funcionalidades do Groovy. A linguagem Groovy é uma linguagem de programação baseada em Java que possibilita o uso de sintaxes próprias bem como as usuais no Java, possibilitando a mistura de sintaxe Groovy com Java dentro do mesmo código. Estas funcionalidades vêm oferecer uma maior usabilidade ao utilizador, facilitando a programação e permitindo um código mais compacto.

Nesta tese começou-se por abordar os mecanismos de funcionamento/manipulação das Abstract Syntax Tree (AST). Com base no estudo das AST foram criados monitores de variáveis e métodos. Estes monitores permitem ao utilizador, por exemplo: monitorizar variáveis para saber quando estas são alteradas/acedidas e executar um método específico definido pelo utilizador (por exemplo para monitorização da utilização de recursos). Esta funcionalidade de monitor é oferecida aos programadores através de uma nova sintaxe que foi adicionada à linguagem Groovy.

O Groovy possui ainda limitações no que respeita às *closures*, tais como não realizar a verificação de tipos de dados. Essas limitações foram abordadas e analisadas de modo a encontrar novas formas de as solucionar. Isto é, foram feitas transformações globais às AST de forma a fazer a verificação do tipo de entrada e de saída das *closures*.

Outra limitação encontrada no Groovy e abordada nesta tese relaciona-se com o paradigma publish/subscribe para o Groovy. Esta funcionalidade não está comtemplada pelo Groovy na sua versão base e foi introduzida no decorrer deste trabalho. Esta funcionalidade permite maior reutilização do *software* por meio de subscrição de dados/eventos.

Palavras-Chave

Abstract Syntax Tree, AntlrParserPlugin, Groovy, Java, Transformations, Publish/Subscribe Paradigm, AspectJ, Closures

Índice

Índice de Figuras

Capítulo 1
Introdução

Este projecto enquadra-se no âmbito da disciplina anual de Estágio/Dissertação do Mestrado em Engenharia Informática do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

O objectivo principal desta dissertação é o de melhorar a linguagem Groovy e ajudar ao seu desenvolvimento implementando funcionalidades que auxiliem a programação em Groovy [1][2][3].

Trata-se de um projecto de investigação a tempo inteiro realizado nas instalações do Departamento de Engenharia Informática estando sob a orientação do professor Fernando Barros.

1.1. Contextualização e limitações actuais

O Groovy ainda não se trata de uma linguagem usada a grande escala nem por muitos programadores. Encontra-se em franco crescimento mas de momento não possui um número de utilizadores capaz de rivalizar por exemplo com os do Java[4]. Como consequência a existência de ferramentas de auxílio ou que complementem o Groovy não são em grande número como seria de esperar.

Surge então uma janela de oportunidade por onde a experiência Groovy pode ser melhorada, identificando aspectos da linguagem que careçam de alguma funcionalidade, ou identificando ferramentas que auxiliem o programador durante o desenvolvimento de aplicações. É neste ponto que esta dissertação se vai focar, melhorar a experiência do programador de Groovy.

1.2. Objectivos da dissertação

O primeiro objectivo centra-se em implementar algumas das características do *Aspect Programming*[5][6] no Groovy. Mais concretamente serão implementados monitores de variáveis e de métodos o que irá ajudar o programador a realizar as tarefas de *debugging*. Estes monitores vão denominar-se de *ReadMonitor*, *WriteMonitor* e *MethodMonitor*.

Um segundo objectivo trata-se de identificar lacunas da própria linguagem e tentar colmatá-las recorrendo à alteração do próprio Groovy. Uma das lacunas identificadas logo na fase inicial foi o facto de o Groovy ignorar a parametrização das *closures* e não realizar qualquer tipo de verificação de tipos nos parâmetros de entrada. Acredito que esta verificação seja útil de modo a garantir um funcionamento consistente da aplicação e como consequência tornou-se um objectivo a atingir. Uma outra funcionalidade a incluir nas *closures* será a verificação do tipo de retorno e garantir que a *closure* só pode devolver o tipo especificado.

O terceiro grande objectivo deste projecto trata de implementar o paradigma *publish/subscribe* [7] na linguagem de forma simples e eficaz. Ainda não existe uma implementação nativa para Groovy e como tal foi traçado o objectivo de desenvolver uma.

O resultado final será um novo Groovy que já tenha nele incluído todas estas novas funcionalidades.

1.3. Resumo dos Objectivos

Os objectivos práticos a atingir neste trabalho são os seguintes:

- Criação de monitores de métodos e variáveis:
  - ReadMonitor
    - Criar sintaxe alternativa
  - WriteMonitor
    - Criar sintaxe alternativa
  - MethodMonitor
- Closures
  - Compreender limitações actuais
  - Planear melhorias e implementá-las
    - Verificação de parâmetros
    - Verificação do tipo de retorno
  - Criação de novos métodos e campos da *closure*
    - Parametrização da *closure* e respectiva função de acesso
    - String que contém o bloco de código da *closure* e respectivo *getter*
- Publish/Subscribe
  - Implementar usando transformações AST locais
  - Criar sintaxe alternativa para o utilizador não precisar de usar as transformações de forma directa

1.4. Estrutura do documento

Uma explicação detalhada de cada capítulo segue de seguida.

Este primeiro capítulo descreve a contextualização do trabalho e também a estrutura do resto do documento.

No capítulo 2 é realizado o estado da arte. O que foi estudado nesta secção foram as características do *Aspect Programming* e tecnologias que usem o *Aspect Programming* como por exemplo o AspectJ [8] que está destinado para ser usado com Java. Foi também estudado o paradigma *publish/subscribe*. São também estudadas as *closures* e o seu funcionamento.

No terceiro capítulo é realizada uma introdução e descrição da linguagem Groovy, descrevendo as suas principais funcionalidades em comparação principalmente com o Java.

O capítulo 4 explica como transformar o Groovy a um detalhe elevado o suficiente de modo a não alongar demasiado este documento. Portanto aqui descreve-se as maneiras de criar uma transformação AST [9] e como proceder para a aplicar numa aplicação. É também explicado de forma breve a estruturação do código Groovy para entender como o alterar.

No capítulo 5, são descritas as transformações elaboradas durante esta dissertação, nomeadamente o *ReadMonitor*, *WriteMonitor* e o *MethodMonitor*.

No capítulo 6, são analisadas as limitações das *closures*. Posteriormente são explicados e aplicados os melhoramentos propostos.

No sétimo capítulo, é analisado e descrito o paradigma *publish/subscribe* e é explicado como foi implementado em Groovy.

No oitavo capítulo é detalhado todo o percurso realizado durante a dissertação, e é também feita uma comparação entre o plano de trabalhos inicial e o final, justificando as respectivas diferenças.

O capítulo 9 aborda o trabalho futuro a realizar. Após a realização deste trabalho surgem novas questões que podem ser investigadas, e como tal é nesta secção que se encontram.

O décimo e último capítulo contém apenas as conclusões e comentários finais sobre o trabalho realizado.

Por último encontram-se as referências e os anexos do documento.

Capítulo 2
Estado da Arte

Neste capítulo vou referir as tecnologias que existem e que tiveram de ser consultadas para a elaboração deste trabalho. Assim como serão referidas as ferramentas que de alguma forma possam realizar as mesmas ou semelhantes tarefas desenvolvidas nesta dissertação.

2.1. AspectJ

O AspectJ [10][11] é uma linguagem de *Aspect-Oriented Programming* (AOP) [12]. O AspectJ pode ser aplicado em qualquer aplicação escrita em Java e facilita a tarefa de *debugging* permitindo interceptar métodos e variáveis quando o programador assim o entender.

A sintaxe usada no AspectJ é em tudo semelhante a Java [13], salvo algumas novas palavras-chave próprias da linguagem.

O AspectJ adiciona alguns conceitos novos, tais como:

- *Pointcuts* [14] – Permitem ao programador especificar pontos concretos do programa com o objectivo de por exemplo executar algum bloco de código. Para tal os *pointcuts* têm de ser usados em conjunto com os *advices* que são descritos de seguida.
- *Advices* [15] – Contêm blocos de código inseridos pelo programador. Quando usados em conjunto com os *pointcuts* o código é executado antes ou depois do ponto do programa especificado pelos *pointcuts*. O antes ou depois é também definido pelo utilizador na declaração do *advice*, podendo para isso usar as palavras-chave *before* e *after*.
- *Inter-type declarations* [16] – Permite adicionar métodos e membros a classes, assim como alterar a sua herança. Nesta dissertação é possível realizar estas mesmas tarefas recorrendo às transformações AST que serão descritas mais à frente.

O AspectJ foi desenvolvido pela *Eclipse Foundation*, proprietária do Eclipse IDE [17]. E como tal a integração com o IDE facilita ainda mais a experiência com o programador. No entanto nada impede que a ferramenta seja usada na linha de comandos.

No momento de intercepção de métodos e variáveis o AspectJ permite executar código definido pelo utilizador. Além disso permite adicionar métodos a classes de forma transparente e também alterar métodos já existentes.

No que respeita a esta dissertação não existe uma ferramenta do género desenvolvida para Groovy, então o objectivo passa por desenvolver ferramentas que realizem funções semelhantes. É com este objectivo em mente que foram desenvolvidas as transformações *Read/WriteMonitor* e *MethodMonitor*.

O *Read* e *WriteMonitor* destinam-se a monitorizar variáveis anotadas, e o *MethodMonitor* a monitorizar os métodos. As três transformações permitiram executar código definido pelo utilizador.

2.2. Paradigma Publish/Subscribe

Algumas arquitecturas de *software* são constituídas por várias aplicações. Muito frequentemente estas aplicações necessitam de comunicar entre si e portanto é construída uma infraestrutura de comunicação que permita às aplicações enviar e receber mensagens. No entanto, uma aplicação só estará interessada num determinado tipo de mensagens. O problema em causa é então simplificar a troca de mensagens fazendo com que uma determinada mensagem seja enviada apenas para as aplicações que estiverem interessadas nele.

Surge assim o paradigma *publish/subscribe* [18][19], que se trata de um esquema de interacção e troca de mensagens entre as aplicações. Neste paradigma existem os *publishers* que são os objectos que publicam eventos. E existem os *subscribers* que são os objectos que subscrevem determinados eventos.

Os *publishers* podem publicar vários eventos, estes eventos em programação orientada a objectos correspondem normalmente a classes/objectos.

O *publish/subscribe* pode ter diferentes formas de aplicação:

- Baseado por listas de subscritores – Nesta implementação é identificado um tópico a ser publicado, e por cada tópico a ser publicado existe uma lista de subscritores. Este modelo baseia-se na *Observer Pattern* [20]. E tendo esta tecnologia em conta podemos dividir as classes em dois tipos: os *subjects* e *observers*. Um *subject* contém por norma 3 métodos, *attach(), detach()* e *notify()*. Já o *observer* contém apenas o método *update()*.
  Neste modelo os *observers* que estiverem interessados em determinados *subjects* têm de efectuar a chamada ao método *attach()* do *subject*. De seguida a partir do momento em que ocorrem alterações ao *subject* este usa o método *notify()* para notificar a lista de subscritores daquele *subject* em concreto.
- Baseado numa comunicação por *broadcast* – Neste modelo a classe que se encontra encarregue de publicar não guarda nenhuma lista de subscritores, ao invés envia o evento em *broadcast* e todos os objectos subscritores vão estar à escuta. O que acontece neste modelo é quem nem todos os objectos que estão à escuta estão interessados em todos os eventos publicados. E portanto estes no momento que recebem o evento verificam se estão interessados nele, e caso estejam aí sim é processada a mensagem.

De notar que para Groovy não existia nenhuma aplicação do *publish/subscribe* desenvolvida e pronta a usar. As vertentes existentes são para Java e não para Groovy pelo que teria de haver algumas adaptações. Além da adaptação do paradigma para o Groovy, vai ser criada uma nova sintaxe para o utilizar.

2.3. Closures

As *closures* [22][23] são uma característica que diferencia o Groovy do Java, já que estas não vêm incluídas de origem no Java, e só virão quando for lançada a versão 8 [24]. As *closures* tratam-se de blocos de código que podem ser inseridos em qualquer parte do código e têm o mesmo funcionamento que os métodos. Podemos atribuir este bloco de código a uma variável e usá-la pelo código como se usaria qualquer outra. Isto traz a vantagem de poder usar a *closure* como um método normal, ou poder passa-la como parâmetro para algum método em particular, e aqui se encontra uma das diferenças entre as *closures* e os métodos normais.

De seguida segue-se o exemplo de uma *closure* ilustrada pela Figura 1 - **Exemplo de *closure* simples**:

```
def closure = {
      println it
}
closure(5)

Output:
5
```

Figura 1 - Exemplo de *closure* simples

Neste exemplo como se pode observar a *closure* é criada e tudo o que faz é imprimir o valor que lhe for passado como parâmetro. Para a chamar basta proceder da mesma forma como se procederia com um método.

Na Figura 2 um exemplo de uma função que recebe uma *closure* como parâmetro:

```
def closure = {
      println it
}
[1,2,3,4].each(closure)

Output:
1
2
3
4
```

Figura 2 - Exemplo de *closure* usada como parâmetro

Isto já não seria possível de realizar com métodos normais e é realmente uma inovação na linguagem que se revela útil para o programador pois permite uma maior flexibilidade na forma de programar.

Existem também as chamadas *closures* anónimas, que são blocos de código ao qual não é atribuído qualquer nome, e como tal só são evocadas no local onde estão inseridas no código. Estas *closures* revelam-se bastante úteis para qualquer programador. Facilita o acto de programar em vários aspectos como por exemplo para imprimir todos os itens de uma lista:

```
[1,2,3,4].each{
      println it
}

Output:
1
2
3
4
```

Figura 3 - Exemplo de uma *closure* usada com uma lista

Como se pode observar na Figura 3 - **Exemplo de uma *closure* usada com uma lista** a quantidade de código necessário para imprimir no ecrã uma lista é menor do que no

exemplo anterior, e também menor do que se usarmos por exemplo um ciclo *for* para o mesmo efeito.

Um segundo exemplo simples sobre *closures* anónimas será o acto de retirar duma lista apenas o ou os elementos que pretendemos, veja-se o exemplo:

```
def newList = [1,2,3,4,5,6,7,8].findAll{
      it % 2 == 0
}
println newList

Output:
[2, 4, 6, 8]
```

Figura 4 - *Closure* para retirar elementos de uma lista

Neste exemplo presente na Figura 4 - ***Closure* para retirar elementos de uma lista** o objectivo é colectar todos os números par presentes na lista, e como se pode ver, é bastante simples de efectuar esta tarefa. Além disso no final temos um código muito mais minimalista do que se usássemos algum dos equivalentes em Java.

No entanto estas *closures* do Groovy são passiveis de melhoramentos, e é nisso que vai ser focado neste trabalho, toda a informação respectiva aos melhoramentos está disponível no **Capítulo 6**.

Capítulo 3
Groovy

O Groovy trata-se de uma linguagem de programação orientada a objectos, que tem como uma das principais características o facto de ser uma linguagem dinâmica. Esta linguagem é também fácil de aprender por programadores de Java [25], já que o Groovy permite grande parte da sintaxe do Java.

O código de um ficheiro Groovy ao contrário do Java pode ainda ser compilado directamente como um script não necessitando de gravar em disco as classes compiladas para a execução do script em causa.

Como prova do dinamismo da linguagem é por exemplo o facto de que esta permite declarar variáveis sem que estas tenham um tipo explícito. A atribuição de um tipo à variável é realizada em *runtime* e não é permanente, o que significa que ao longo do tempo de execução do programa o tipo da variável pode mudar.

Outra característica que não se encontra presente em muitas linguagens, é a possibilidade de efectuar chamadas a métodos de forma completamente dinâmica. No exemplo seguinte o método é invocado sem saber o nome do método de antemão.

```
class Carro {
      def acelera() { println "O "+this+" acelerou!" }
      def trava() { println "O "+this+" travou!" }
}
def doAction( carro, action ) {
      carro."$action"()                  //A acção é passada na invocação
}
def car = new Carro()
doAction( car, "acelera" )
doAction( car, "trava" )

Output:
O Carro@1f77df3 acelerou!
O Carro@1f77df3 travou!
```

Figura 5 - Exemplo da chamada dinâmica de um método em Groovy

No exemplo da Figura 5 o método *doAction()* não sabe qual o método que vai invocar e sobre que objecto é que o vai invocar. Só saberá esta informação quando o programa se encontrar a correr.

O Groovy possui ainda excelentes capacidades de *Meta-Programming* [26][27], que se trata de escrever aplicações que se reescrevem e manipulam a elas próprias ou a outras aplicações. Para realizar isto no Groovy uma possibilidade será trabalhar com a *MetaClass* existente em todos os objectos. Através desta propriedade podemos adicionar métodos, construtores e propriedades à classe de forma completamente dinâmica.

A característica que vai ser mais utilizada na realização deste projecto serão as transformações das *Abstract Syntax Tree's* [28], que nos permitem alterar em grande parte o próprio compilador e as classes geradas por este. O Java também permite este tipo de transformações, no entanto no Groovy o uso delas está muito mais *user-friendly* e permite um maior conjunto de funcionalidades. No capítulo seguinte serão abordadas mais a fundo este tipo de transformações.

Capítulo 4
Transformar o Groovy

Durante o processo de compilação do Groovy, a um dado ponto, o código-fonte é representado em memória através de uma *Concrete Syntax Tree* (CST), que posteriormente é transformada numa *Abstract Syntax Tree* (AST). Só depois de o compilador possuir as AST's definidas é que tudo é transformado em *bytecode* que o *Java Virtual Machine* (JVM) [29] vai usar para executar a aplicação. Para transformar o Groovy pode-se portanto usar as transformações das AST.

Estas transformações permitem aceder e consequentemente alterar as AST's formadas durante a compilação. Também é permitido adicionar código à aplicação, e da mesma forma é também permitido remover código. O código está estruturado na forma de *Statement's* e *Expression's*, isto será abordado e explicado em maior pormenor mais à frente.

O Groovy possui 9 fases de compilação [30] podendo as transformações ser aplicadas a cada uma delas a partir da fase de inicialização, estas diversas fases são apresentadas e descritas na Figura 6 - **Fases do processo de compilação do Groovy**:

Fase de Compilação	Descrição
Initialization:	O código-fonte é aberto e o ambiente é configurado
Parsing:	É usada a gramática do Groovy para procurar erros de sintaxe e gerar a árvore de *tokens*
Conversion:	É criada a AST a partir da árvore de *tokens*
Semantic Analysis:	Nesta fase são realizados testes de viabilidade e consistência que a gramática não consegue resolver
Canonicalization:	Termina a construção da AST
Instruction Selection:	Nesta fase é escolhido o conjunto de instruções, Java 5 ou antecedentes do Java 5
Class Generation:	Cria o *bytecode* e coloca-o em memória
Output:	Escreve o *bytecode* para o sistema de ficheiros
Finalization:	São realizados os últimos processos de limpeza e finalização

Figura 6 - Fases do processo de compilação do Groovy

Apesar de a documentação do Groovy declarar que podemos usar as primeiras 3 fases de compilação, tal não se revelou possível através de transformações das AST. Já que sempre que se seleccionava uma fase anterior à da análise semântica o comportamento era completamente idêntico à da fase semântica, pelo que se conclui que o compilador na prática apenas permita transformações que actuem na fase da análise semântica e posteriores.

4.1. AntlrParserPlugin

Com o objectivo de adicionar funcionalidades sintácticas à linguagem, que permitam a manipulação do código Groovy a nível do *parsing*, estudaram-se 2 métodos no primeiro semestre. O primeiro passava por alterar a própria gramática da linguagem usando o *Antlr* (*Another tool for Language Recognition*) [31], esta é a gramática que o Java e o Groovy usam. O segundo método passa por utilizar o *AntlrParserPlugin* [32][33] que é um objecto Groovy que permite realizar o *parsing* dos ficheiros de código antes de os enviar para o compilador, e como tal também permite a alteração sintáctica da linguagem.

Para escolher um dos métodos a utilizar tentou-se realizar a mesma tarefa num e noutro método. Esta tarefa consistia em alterar uma palavra-chave da linguagem que fosse reconhecida e aceite pelo compilador. No entanto usando o primeiro método não se conseguiram os resultados desejados e após algumas tentativas concluí que o trabalho e investigação necessários para conseguir realizar esta tarefa alterando a gramática poderia ser demasiado tendo em conta os outros objectivos da dissertação.

Já durante o estudo sobre a *AntlrParserPlugin* percebeu-se desde cedo que seria possível realizar a simples alteração de uma palavra-chave, e por isso este método ganhou prioridade para ser usado durante o projecto. Como já foi referido esta *plugin* trata-se de um objecto Groovy e como tal vejo isto como uma vantagem já que permite alterar a linguagem usando o próprio Groovy.

No entanto usando este ultimo método, a alteração da sintaxe não é realizada de forma nativa. Ou seja a alteração não é feita no próprio compilador. O que a *plugin* realiza é um pré-processamento de todos os *scripts* e objectos Groovy antes de os enviar para o compilador. Em suma, função da *plugin* é detectar a nova sintaxe e transformá-la em código Groovy normal que o compilador detecte e compile correctamente.

Assim a *AntlrParserPlugin* colmata a inabilidade das transformações AST em actuarem nos primeiros níveis de compilação.

A forma como esta *plugin* funciona está descrita com maior pormenor no **Capítulo 4.5**.

4.2. Transformações AST

O Groovy disponibiliza dois tipos de transformações, as transformações locais e as transformações globais.

Transformações globais [34]**:** São transformações que após a sua criação têm de ser compiladas para um ficheiro .jar. Sempre que se pretender usar estas transformações tem de se adicionar o ficheiro .jar ao *classpath* do compilador. As transformações são então aplicadas a todos os ficheiros de código que forem compilados. Segundo a documentação do Groovy, estas transformações podem ser aplicadas em qualquer fase de compilação.

Transformações locais [35]**:** São em quase tudo semelhantes às transformações globais, no entanto não é necessário compilá-las para um ficheiro .jar, ao invés disso o código fonte de uma transformação local pode-se encontrar em qualquer lugar no sistema de ficheiros, necessitam apenas de ser importadas para o projecto onde se pretende usá-las. Estas transformações no entanto não são aplicadas em todo o código. Para serem aplicadas tem de se utilizar anotações no código fonte, esta particularidade já era usada no Java. O compilador detecta estas anotações e aplica as transformações devidas.

As transformações AST permitem ao programador actuar a partir do nível da análise semântica.

Anotações

As anotações são necessárias para declarar as transformações locais, nas anotações é obrigatório definir desde logo 2 parâmetros, o *Target* para especificar sobre que tipo de código a transformação irá actuar, e também é necessário definir onde se encontra localizado o ficheiro que contém a transformação a implementar. Este último ficheiro poderá estar escrito em Groovy ou Java, portanto a escolha da linguagem fica ao critério do utilizador.

De seguida apresenta-se o exemplo de uma anotação na Figura 7:

```
@Target([ElementType.METHOD,ElementType.TYPE,ElementType.FIELD])
@GroovyASTTransformationClass(["monitor.transform.WriteMonitorAST"])
public @interface WriteMonitor {
	String codigo () default "";
}
```

Figura 7 - Exemplo de declaração de uma anotação

@Target

O *Target* é o local onde definimos quais os elementos do programa que podem ser anotados, estes elementos são membros da enumeração java.lang.annotation.ElementType e estão apresentados de seguida:

Annotation_type: Uma anotação com este elemento serve para anotar outras anotações.
Constructor: Usada para transformações que tenham como intuito trabalhar sobre o ou os construtores de uma classe.
Field: Tem como objectivo anotar as variáveis membro da classe.
Local_variable: Permite anotar variáveis locais presentes dentro de métodos.
Method: Possibilita a anotação de métodos da classe.
Package: Esta anotação permite anotar as declarações de *package* da aplicação.
Parameter: Possibilita anotar directamente os parâmetros, como por exemplo os parâmetros de um método.
Type: Esta anotação normalmente é apenas usada para anotar directamente classes ou interfaces.

@GroovyASTTransformationClass

Esta anotação também é obrigatória pois é ela que informa o compilador da localização do ficheiro que contém a transformação a ser implementada.

Transformação

O ficheiro que contém a transformação tem obrigatoriamente de conter uma classe que implemente a interface *ASTTransformation*, e esta classe tem por sua vez de estar anotada pela anotação *@GroovyASTTransformation*.

Esta anotação tem de conter a fase de compilação na qual se pretende que a transformação actue.

O código a ser executado pela transformação tem de ser colocado no método *visit()*, como tal este método é também ele obrigatório em qualquer transformação criada.

Um exemplo base da criação da uma transformação está na Figura 8:

```
@GroovyASTTransformation(phase=CompilePhase.CANONICALIZATION)
public class ReadMonitorAST implements ASTTransformation {
      public void visit(ASTNode[] nodes, SourceUnit sourceUnit) {
            (…)
      }
}
```

Figura 8 - Corpo da criação de uma transformação AST

Como podemos observar, esta transformação vai actuar na fase de *Canonicalization*, e implementa correctamente a *ASTTransformation,* além disso possui o já referido método *visit()* que será o método que vai ser chamado durante o processo de compilação do Groovy.

O método *visit()* contém como parâmetros os objectos que necessitamos para intervir no processo de compilação, já que a partir do *ASTNode* e da *SourceUnit* podemos aceder a toda a *Abstract Syntax Tree* da classe anotada e a partir desta realizar todas as operações desejadas.

O *ASTNode* é o objecto mais importante de uma transformação, pois é a partir deste que conseguimos obter toda a árvore de um programa Groovy. Uma explicação mais detalhada será dada no ponto seguinte deste documento.

A respeito do *ASTNode*, a primeira posição deste *array* vai conter a anotação, ou seja, em linguagem Groovy o *AnnotatedNode*, e a segunda posição vai conter o objecto anotado, quer este seja uma classe, um campo ou um método.

A titulo de exemplo, se o tipo da transformação for *Type*, a segunda posição do *array nodes* vai conter a classe anotada, então a partir do *nodes[1]* podemos obter todos os métodos da classe apenas invocando a seguinte linha de código presente na Figura 9:

```
List methods = nodes[1].getMethods()
```

Figura 9 - Exemplo de como retirar todos os métodos de uma classe

E de seguida para editar a *AST* de cada um destes métodos bastaria editar o código que se encontra em cada um deles.

Este código no entanto não se encontra no formato texto, e já se encontra em objectos, uma explicação mais aprofundada sobre a estruturação do código será dada de seguida.

4.3. Estruturação do código

A classe base de todos os *nodes* de uma AST é a:

ASTNode

É a partir desta que descendem as classes mais importantes representativas do código, sendo que apenas as principais subclasses serão apresentadas de seguida.

AnnotatedNode

Esta classe é a principal classe base de todos os *nodes* que podem ser anotados, sendo por isso uma classe importante para guardar as anotações presentes nestes nódulos. É a partir desta que podemos portanto ir buscar todas as anotações inseridas no código. As principais subclasses directas são as seguintes:

- ClassNode – É nesta classe que fica guardada toda a informação sobre uma classe Groovy, através desta podemos obter todos os métodos presentes na classe, assim como adicionar e remover. Pode-se ainda obter todos os campos presentes na classe, assim como saber quais as interfaces que a classe implementa, ou seja, em suma toda a informação sobre a classe é aqui guardada.

- MethodNode – Guarda toda a informação sobre os métodos de uma classe, podemos aceder ao tipo de método, aos parâmetros do método, e também ao código presente dentro do método. Todos estes campos referidos podem também ser alterados directamente num objecto deste tipo. Quando queremos aceder ao código o objecto devolvido é um *BlockStatement*, que será abordado mais à frente.

- FieldNode – Guarda a informação sobre os membros da classe. É possível aceder ao tipo da variável, à sua inicialização e nome, e caso se deseje alterá-los à vontade do utilizador.

- Expression – Esta é a classe base de todas as expressões mais básicas de Groovy, por expressões entendem-se declarações de variáveis (*DeclarationExpression*), chamadas de métodos (*MethodCallExpression*), atribuições (*BinaryExpression*), referências a variáveis (*VariableExpression*), entre várias outras.

AnnotationNode

Esta é a classe que guarda toda a informação sobre a anotação, é através deste objecto que podemos ir buscar os membros desta e o seu respectivo valor.

ModuleNode

Esta classe representa um módulo que pode ser constituído por várias classes e métodos misturados, como por exemplo acontece nos casos dos script's. Um objecto deste tipo permite que seja devolvida a lista de *ClassNode's* presente no script, e permite também que seja devolvida a lista de *MethodNode's* que se encontrar no script. Assim como é possível devolver cada um dos objectos referidos, também é por sua vez permitido adicionar classes e métodos ao módulo.

Statement

A *Statement* é outra das classes base que é bastante importante na estrutura do Groovy, pois existem muitas subclasses desta que são usadas extensivamente quando realizamos operações com as transformações das AST's. As principais subclasses são as seguintes:

- ***BlockStatement*** – Trata-se de um bloco que contém uma lista de *Statement's*, na maioria das transformações de AST's que forem realizadas é provável que a certo ponto tenhamos de lidar com um objecto deste tipo.

- ***ForStatement*** – Este *Statement* é criado para representar um ciclo *for*, o código devolvido deste *Statement* será um objecto *BlockStatement* caso contenha mais que uma linha de código, ou uma *ExpressionStatement* caso possua apenas uma linha de código.

- ***IfStatement*** – Este objecto é usado para representar um *if*, e é possível requisitar ao objecto a devolução do código presente no *if*, e também o código presente no *else*.

- ***ReturnStatement*** – Cada *return* no código fica representado na AST como um *ReturnStatement*, a partir deste podemos aceder ao que está a ser retornado.

- ***SwitchStatement*** – Um *switch/case* no código vai fazer com que seja criado este objecto internamente, e este objecto vai conter todos os *CaseStatement's* que podem posteriormente ser alterados.

- ***CaseStatement*** – Este objecto representa os *case's* do *switch*, é a partir deste objecto que podemos ir buscar o bloco de código correspondente a cada *case*, este bloco de código é devolvido na forma dum objecto *BlockStatement*.

- ***WhileStatement*** – Como o próprio nome indica é neste objecto que fica guardada a informação do ciclo *while*, do qual podemos ler e adicionar código tal e qual como nos *Statement's* anteriores.

4.4. Exemplo da AST de um código

Com o objectivo de dar a entender a forma como o código está estruturado, elaborei um pequeno código exemplo para depois o desmembrar nos vários objectos que o compõem.

O código é o seguinte:

```
class Foo{
    def boo(){
        int i = 10
        println i
        for(i=0;i<10;i++){
            if(i==5){
                println 10
                return
            }
            continue
        }
    }
}
```

Figura 10 - Exemplo de um código Groovy

Como se pode observar na Figura 10 é um bloco de código bastante simples que declara uma classe, um método, além de ter um ciclo e chamadas a métodos simples.

A árvore AST resultante do código acima mencionado é a seguinte da Figura 11:

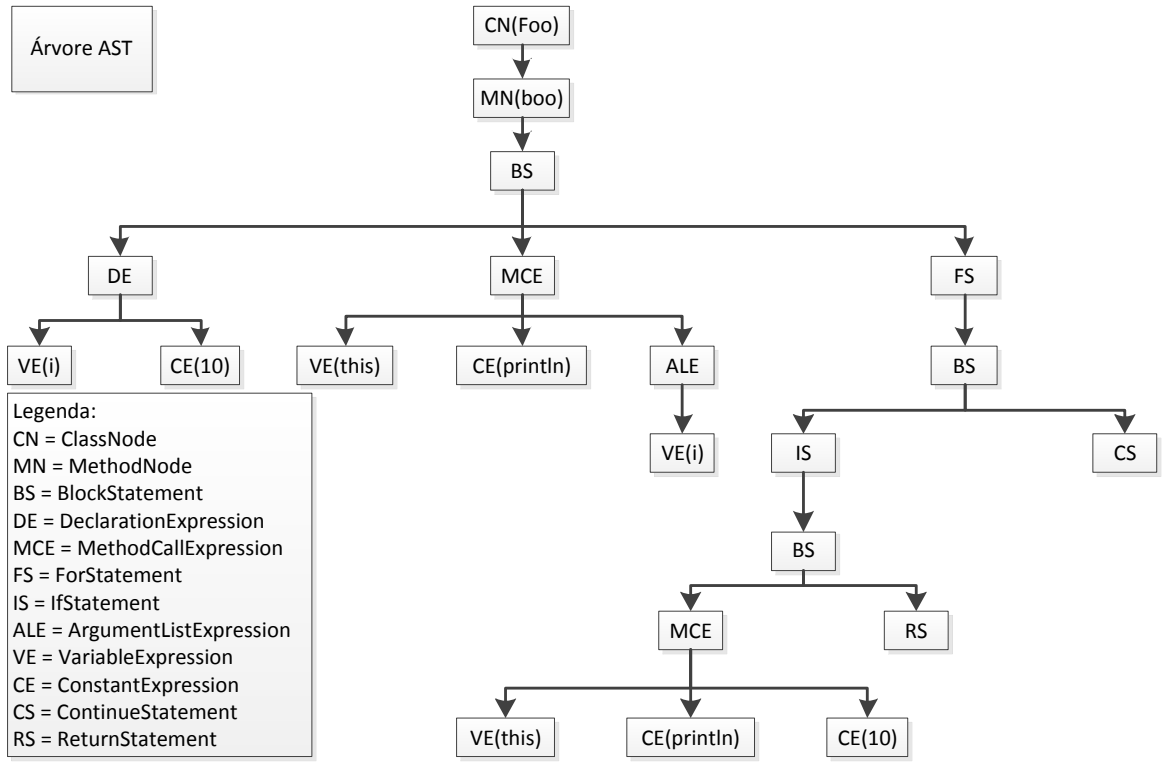

Figura 11 - Árvore gerada pelo código do exemplo

O objecto base neste código é o *ClassNode* que além de conter informação relativa à classe e suas propriedades contém um método representado por um *MethodNode*.

Cada *MethodNode* contém um *BlockStatement* que por sua vez contém uma lista de expressões e/ou *Statement's*. Neste caso a lista será composta por:

- *DeclarationExpression* – Que corresponde à declaração da variável *i* na linha:

  ```
  "int i = 10"
  ```

  A *DeclarationExpression* contém ainda dois objectos, uma *VariableExpression* que corresponde à variável 'i', e uma *ConstantExpression* que neste caso é o valor de inicialização da variável que corresponde a '10'.

- *MethodCallExpression* – Este objecto é usado sempre quando um método é chamado, e nesta ocasião é chamado o método 'println' na linha:

  ```
  "println i"
  ```

  Está ainda contido neste objecto 3 objectos adicionais, uma *VariableExpression* com o parâmetro 'this', uma *ConstantExpression* com o nome do método ('println') e uma *ArgumentListExpression* que contém uma lista de argumentos, mas neste caso apenas contém uma *VariableExpression* de 'i'.

- *ForStatement* – Como o nome indica este objecto refere-se a um ciclo *for* e contém nele todas as informações relativas ao ciclo:

```
"for(i=0;i<10;i++){ (...) }"
```

A partir do *ForStatement* obtém-se um *BlockStatement* que contém como já foi referido, uma lista de objectos/expressões/*Statement's*. Por isto creio que o resto da árvore é auto-explicativa, já que explicar o resto da árvore iria levar à repetição de explicações anteriormente dadas.

4.5. Transformar o Groovy

Para transformar o Groovy de forma permanente temos várias opções à escolha. Os passos a seguir para transformar a linguagem são diferentes consoante o que se pretende alterar. No entanto todas as alterações para ficarem permanentes exigem que o código-fonte da linguagem seja recompilado.

Compilar código-fonte do Groovy

Depois de obter o código fonte e de o colocarmos numa dada directoria, basta abrir uma linha de comandos nessa mesma directoria e executar o comando: *ant install* [36]. A primeira compilação do código fonte é a mais morosa, já que é quando são criadas as classes e verificadas todas as dependências entre elas, podendo atingir aproximadamente uma hora de compilação. As compilações seguintes do código apenas vão compilar ficheiros que tenham sido alterados ou criados, e deixa os inalterados intactos, como tal as compilações seguintes atingem por média um minuto de compilação.

Após a compilação obtém-se a directoria denominada de *target*, e dentro desta encontra-se a *install* que contém as novas bibliotecas Groovy geradas no processo de compilação. Para as instalar basta substituir as bibliotecas que se encontram no *classpath* do Groovy pelas novas.

Nas secções seguintes vou explicar que passos seguir para tornar permanentes as seguintes alterações:

- Adicionar transformações locais ao Groovy
- Adicionar transformações globais ao Groovy
- Alterar a sintaxe da linguagem usando o *AntlrParserPlugin*

Adicionar transformações locais ao Groovy

Caso o objectivo seja adicionar novas transformações locais à linguagem, temos de possuir a anotação e a classe da transformação. Após isso colocamos a anotação na mesma directoria das restantes anotações que na versão actual do Groovy está situada em: "{*sourcedir*}/groovy/lang/". Já a transformação tem de ser inserida aqui: "{*sourcedir*}/org/codehaus/groovy/transform/" que é onde se encontram as restantes transformações do Groovy. Após esta inserção para a transformação ficar implementada de forma permanente basta compilar novamente a linguagem e a anotação e transformação correspondente podem a partir desse momento ser usadas a qualquer altura no código.

Adicionar transformações globais ao Groovy

Para criar a transformação global o processo é ligeiramente diferente do de criar uma transformação local. A classe com a transformação até pode permanecer inalterada seja local, ou global em alguns dos casos. Já a forma de fazer com que o compilador execute a transformação global é bem diferente da forma de executar uma transformação local.

O primeiro passo após ter a classe construída é fazer o *wrapping* dela para um ficheiro *.jar*. Para tal, temos de compilar o ou os ficheiros Groovy que fazem parte da transformação para ficheiros *.class*. De seguida temos de criar o seguinte sistema de directorias ilustrado na Figura 12:

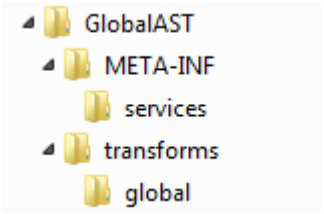

Figura 12 - Esquema de directorias para o ficheiro *jar*

Dentro da directoria *services* vamos criar um ficheiro sem qualquer tipo de extensão comum que ostente o nome de: *org.codehaus.groovy.transform.ASTTransformation*, e dentro deste coloca-se o nome da classe da transformação, como por exemplo: *transforms.global.ClasseName*.

Após as classes compiladas e o ficheiro criado e na directoria correcta apenas resta criar o ficheiro *jar* com a transformação, e para tal basta colocar a directoria *META-INF* e as classes dentro do *jar*. O Java traz um comando que nos facilita este trabalho, e como tal para colocar estas directorias no ficheiro *jar* bastaria executar a seguinte linha presente na Figura 13:

```
C:\workdir> jar cf transformation.jar META-INF classes
```

Figura 13 - Comando para criar o ficheiro *jar*

Por último e para esta transformação entrar em acção basta colocar o ficheiro *jar* no *classpath* do compilador e a transformação será executada.

Alterar a sintaxe da linguagem usando o *AntlrParserPlugin*

Para utilizar este processo já é exigido um nível de conhecimento superior sobre o código-fonte da linguagem, pois pra usufruir deste temos de modificar um/dois ficheiros do código-fonte, possivelmente inserir outro e por fim compilar a linguagem novamente.

Comecemos pelo principal, para modificar a sintaxe usando apenas recursos do próprio Groovy é primeiro necessário criar uma classe que seja uma subclasse da *AntlrParserPlugin*, e dentro desta classe temos de fazer o *override* ao método *parseCST()*, tal como demonstrado na Figura 14:

```
class SourceModifierParserPlugin extends AntlrParserPlugin {
Reduction parseCST(SourceUnit sU, Reader r) throws CompilationFailedException{
            (...)
      }
}
```

Figura 14 - Exemplo de uma subclasse da *AntlrParserPlugin*

Como parâmetro do *parseCST()* é passado um *Reader*, e através deste temos a possibilidade de aceder ao código-fonte original e inalterado do utilizador. Como tal podemos efectuar sobre ele as operações desejadas, essas operações podem passar por alterações sintácticas.

No entanto, a partir deste método temos de encontrar a nova sintaxe presente no código e transformá-la para Groovy já que a alteração sintáctica é no nível de pré-processamento que é tratada e não ao nível do compilador, é este o objectivo desta classe.

Se o objectivo for tornar permanentes na linguagem as alterações sintácticas presentes na classe que estende a *plugin*, então temos de alterar um ficheiro do código-fonte do Groovy. Este ficheiro é o *GroovyMain.java* e encontra-se no *package groovy.ui*.

A tarefa a realizar é informar o compilador que tem de fazer primeiro este pré-processamento do código. Para tal pode-se usufruir de uma das características que o Groovy nos possibilita, que é o facto de permitir enviar uma configuração personalizada para a *GroovyShell* que se trata do objecto responsável por correr todos os *scripts* compilados em Groovy.

Esta configuração é um objecto do tipo *CompilerConfiguration* e permite-nos entre outras coisas activar uma *PluginFactory* que nos permitirá que as alterações sintácticas sejam realizadas.

Como tal resta criar esta *PluginFactory* que é a classe que vai estabelecer a ligação entre o compilador e a *AntlrParserPlugin* desenvolvida. A criação é bastante simples como se poderá ver de seguida na Figura 15:

```
public class SourcePreProcessor extends ParserPluginFactory {
      public ParserPlugin createParserPlugin() {
            return new SourceModifierParserPlugin();
      }
}
```

Figura 15 - Criação de uma *ParserPluginFactory*

Como se pode ver acima esta *factory* apenas tem de devolver um objecto do tipo da *AntlrParserPlugin* desenvolvida.

Agora que temos todas as informações que necessitamos há que voltar ao *GroovyMain* de modo a implementar a configuração. O primeiro passo é criar uma variável membro do tipo *SourcePreProcessor* que é a *ParserPluginFactory* como ilustrado na Figura 16.

```
SourcePreProcessor parserPluginFactory = new SourcePreProcessor();
```

Figura 16 - Inicialização do pré-processador

De seguida pretende-se adicionar este objecto na configuração do compilador, e para tal temos de ir às instâncias onde a *GroovyShell* é inicializada e configurada para processar os *scripts* Groovy. Esta inicialização dá-se no método *processOnce()* da classe *GroovyMain* então tem de se efectuar esta configuração antes desta inicialização da seguinte forma ilustrada na Figura 17:

```
conf.setPluginFactory(parserPluginFactory);
GroovyShell groovy = new GroovyShell(conf);
```

Figura 17 - Inserção do pré-processador na *GroovyShell*

Com isto concluído resta apenas compilar a linguagem de novo para tornar esta alteração permanente.

Capítulo 5
Transformações Desenvolvidas

Nesta secção vou passar a descrever as transformações desenvolvidas ao longo do estágio, as duas primeiras são o *Read* e *WriteMonitor* que têm como objectivo monitorizar variáveis. A terceira é o *MethodMonitor* que permite monitorizar métodos.

5.1. ReadMonitor

O objectivo desta transformação é o de informar o utilizador quando uma variável é acedida. É uma funcionalidade importante pois permite ao utilizador ser informado sempre que uma variável em específico é lida em *runtime* e nesse momento executa um bloco de código customizado. É possível ainda aceder e alterar o valor da variável.

O utilizador dispõe de dois métodos de a utilizar, o primeiro consiste em usar uma anotação da forma normal, enviando como parâmetro o código que o utilizador pretende ver executado cada vez que a variável é lida. Um exemplo de uma anotação pode ser esta que se segue:

Anotação

```
@ReadMonitor(codigo = '{Object target, Object currValue -> println
"Target: "+target+ " Current Value: "+currValue}')
Integer k=0
```

Figura 18 - Anotação do *@ReadMonitor*

Esta anotação presente na Figura 18 vai imprimir no ecrã o objecto de onde provém, e o valor actual da variável no momento da sua leitura.

Por exemplo para o seguinte caso-teste:

```
@ReadMonitor(codigo = '{Object target, Object currValue -> println
"Target: "+target+ " Current Value: "+currValue}')
Integer k=0
k = 20
println (k--)
k.getClass()
println "Math.min(50,k): " + Math.min(50, k)
```

Figura 19 - Exemplo de utilização do *@ReadMonitor*

Irá ser produzido o seguinte output:

```
Target: monitor.PrototipoClass@135d392 Current Value: 20
20
Target: monitor.PrototipoClass@135d392 Current Value: 19
Target: monitor.PrototipoClass@135d392 Current Value: 19
Math.min(50,k): 19
Target: monitor.PrototipoClass@135d392 Current Value: 19
```

Figura 20 - Resultado do *output* do exemplo acima

Como se pode observar na Figura 20 a transformação comporta-se como estava previsto, informando o utilizador cada vez que a variável é lida, e como esta no *script* acima é lida em 4 linhas de código a transformação informa o utilizador 4 vezes.

Nova Sintaxe

Existe no entanto outra forma de utilizar esta transformação, através de uma sintaxe customizada que foi criada para esta transformação, para tal basta colocar o código do seguinte modo como exposto na Figura 21:

```
monitor public static float var=0
Read {
      println "Target: " + target
      println "CurrentValue: " + currentValue;
}
println "Antes de ser lida!"
var = 345
println "Depois de ser lida!"
```

Figura 21 - Exemplo do *@ReadMonitor* utilizando a sintaxe alterada

Esta sintaxe exige o uso da nova *keyword* 'monitor' e após a declaração da variável a anotar é necessário inserir um bloco de código à frente de outra nova *keyword* 'Read', este código tem ainda de estar delimitado por chavetas.

O exemplo acima irá produzir o seguinte resultado de saída que se encontra na Figura 22:

```
Antes de ser lida!
Target: class PrototipoClass
CurrentValue: 345,0
Depois de ser lida!
```

Figura 22 - *Output* do exemplo anterior

Esta modificação sintáctica foi conseguida através do uso da *AntlrParserPlugin* tal como foi descrito no **Cap. 4.5.**

Como funciona

Depois de ter sido descrita a estrutura do código em Groovy no **Cap. 4.3.** vai ser descrito o modo de funcionamento da transformação. Para começar, na anotação foi definido que se trata de uma transformação que actua sobre as variáveis membro, ou seja, os *FieldNode's*. E para a anotação receber argumentos foi criado o campo *codigo* dentro desta para receber o código na forma de *String*.

Aqui se encontra a anotação ilustrada na Figura 23:

```
@Target([ElementType.FIELD])
@GroovyASTTransformationClass(["org.codehaus.groovy.transform.ReadMonitorA
ST"])
public @interface ReadMonitor {
      String codigo ()
}
```

Figura 23 - Anotação do *@ReadMonitor* - Código

Já na classe da transformação a abordagem escolhida para cumprir o pretendido pode ser representada pelo seguinte pseudo-código ilustrado pela Figura 24:

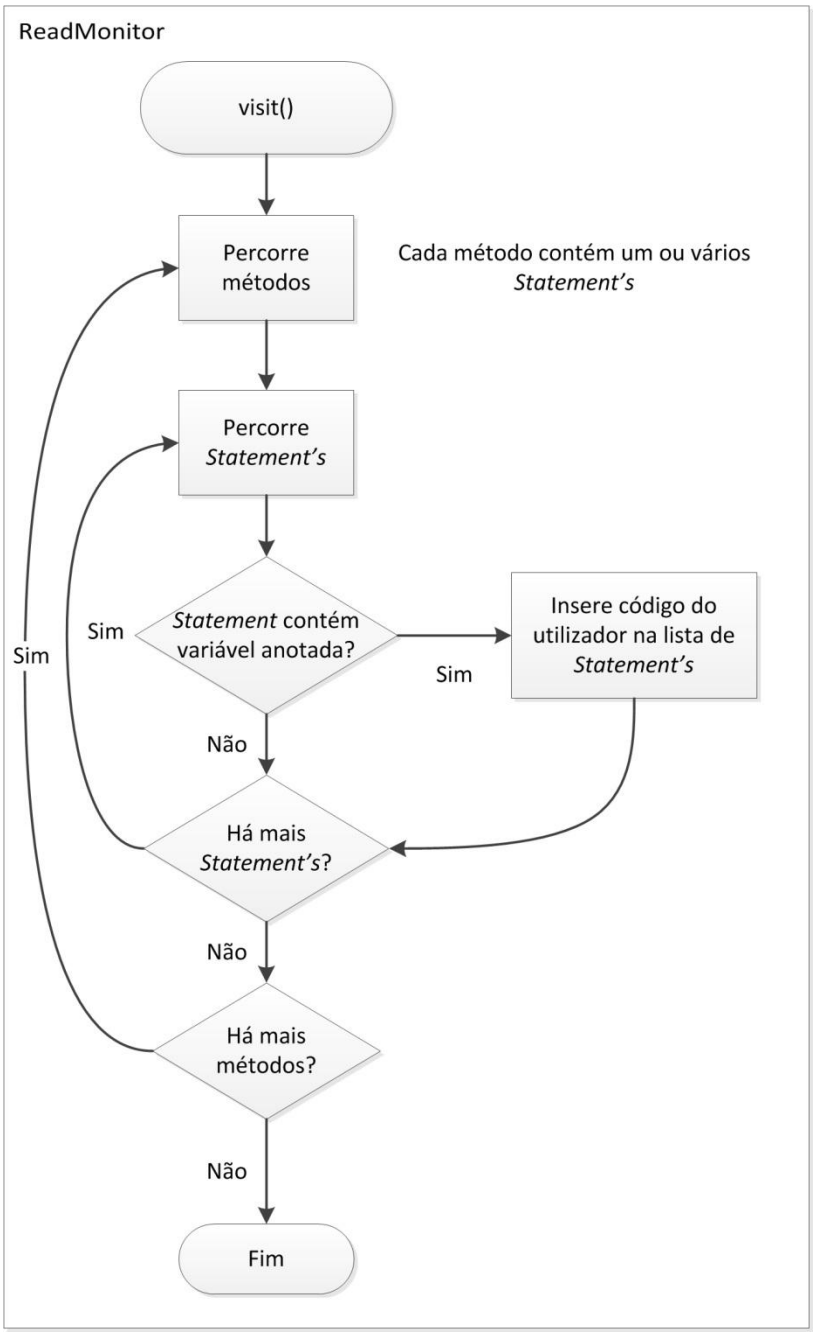

Figura 24 - Fluxograma do *@ReadMonitor*

Como se pode observar através desta simples transformação podemos alterar a estrutura do código presente na aplicação adicionando código customizado pelo utilizador, como tal o *ReadMonitor* pode assim ser usado para um vasto leque de possibilidades.

5.2. WriteMonitor

O *WriteMonitor* é muito semelhante à transformação acima descrita, mas tem como objectivo informar o utilizador quando a variável monitorizada é acedida e/ou escrita, podendo ainda armazenar o novo e antigo valor da variável antes de este ser alterado. Esta transformação assim como a anterior também é bastante útil pois permite algumas funcionalidades que a anterior não permite.

Anotação

```
@WriteMonitor(codigo = '{Object target, Object oldValue, Object newValue -
> println "OldValue: "+oldValue+" NewValue: "+newValue+" Target:
"+target;}')
Integer k=0
```

Figura 25 - Anotação do *@WriteMonitor*

Tal como na transformação anterior esta anotação ilustrada na Figura 25 necessita que seja passado como parâmetro o código a executar quando a variável é escrita. E neste caso a aplicação irá imprimir o antigo e o novo valor da variável sempre que este foi alterado.

Segue-se um exemplo:

```
@WriteMonitor(codigo = '{Object target, Object oldValue, Object newValue -
> println "OldValue: "+oldValue+" NewValue: "+newValue+" Target:
"+target;}')
Integer k=0
k = 20
println (k--)
k = Math.min(50, 25)
println "k + 10 = "+(k+=10)
```

Figura 26 - Exemplo do *@WriteMonitor*

Este exemplo vai gerar o seguinte resultado:

```
OldValue: 0 NewValue: 20 Target: monitor.PrototipoClass@6e63d9
20
OldValue: 20 NewValue: 19 Target: monitor.PrototipoClass@6e63d9
OldValue: 19 NewValue: 25 Target: monitor.PrototipoClass@6e63d9
35
OldValue: 25 NewValue: 35 Target: monitor.PrototipoClass@6e63d9
```

Figura 27 - *Output* do exemplo acima ilustrado

Como se pode observar na Figura 27, tudo funcionou como esperado, já que cada vez que a variável foi escrita foi também executado o código definido pelo utilizador.

Nova Sintaxe

Tal como para a transformação anterior optei por adaptar a *AntlrParserPlugin* também para esta transformação.

Para usar esta transformação usufruindo da nova sintaxe o método é em tudo semelhante ao método que é usado para o *ReadMonitor*, em que a única diferença é que o bloco de código é precedido por *Write* em vez de *Read*, como se pode observar de seguida na Figura 28:

```
monitor String var
Write {
      println "Target: " + target
      println "OldValue: "+oldValue+" NewValue: "+newValue
}
println "Primeira leitura!"
var ="Teste 1"
println "Segunda leitura!"
var ="Teste 2"
```

Figura 28 - Exemplo do *@WriteMonitor* usando a nova sintaxe

Este código produzirá o seguinte *output* presente na Figura 29:

```
Primeira leitura!
Target: class PrototipoClass
OldValue: null NewValue: Teste 1
Segunda leitura!
Target: class PrototipoClass
OldValue: Teste 1 NewValue: Teste 2
```

Figura 29 - *Output* do exemplo anterior

Também variáveis do tipo *String* podem ser monitorizadas com sucesso como o *output* acima exemplifica, a transformação captura com sucesso o antigo e novo valor da variável em questão.

Como funciona

Assim como no *ReadMonitor* o funcionamento desta transformação é muito simples. Para começar temos a anotação que é igual à anotação da transformação anterior

A anotação do *WriteMonitor* é a seguinte ilustrada na Figura 30:

```
@Target([ElementType.FIELD])
@GroovyASTTransformationClass(["org.codehaus.groovy.transform.WriteMonitor
AST"])
public @interface WriteMonitor {
      String codigo ();
}
```

Figura 30 - Anotação do *@WriteMonitor* - Código

De seguida encontra-se na Figura 31 o pseudo-código da transformação para facilitar a sua compreensão:

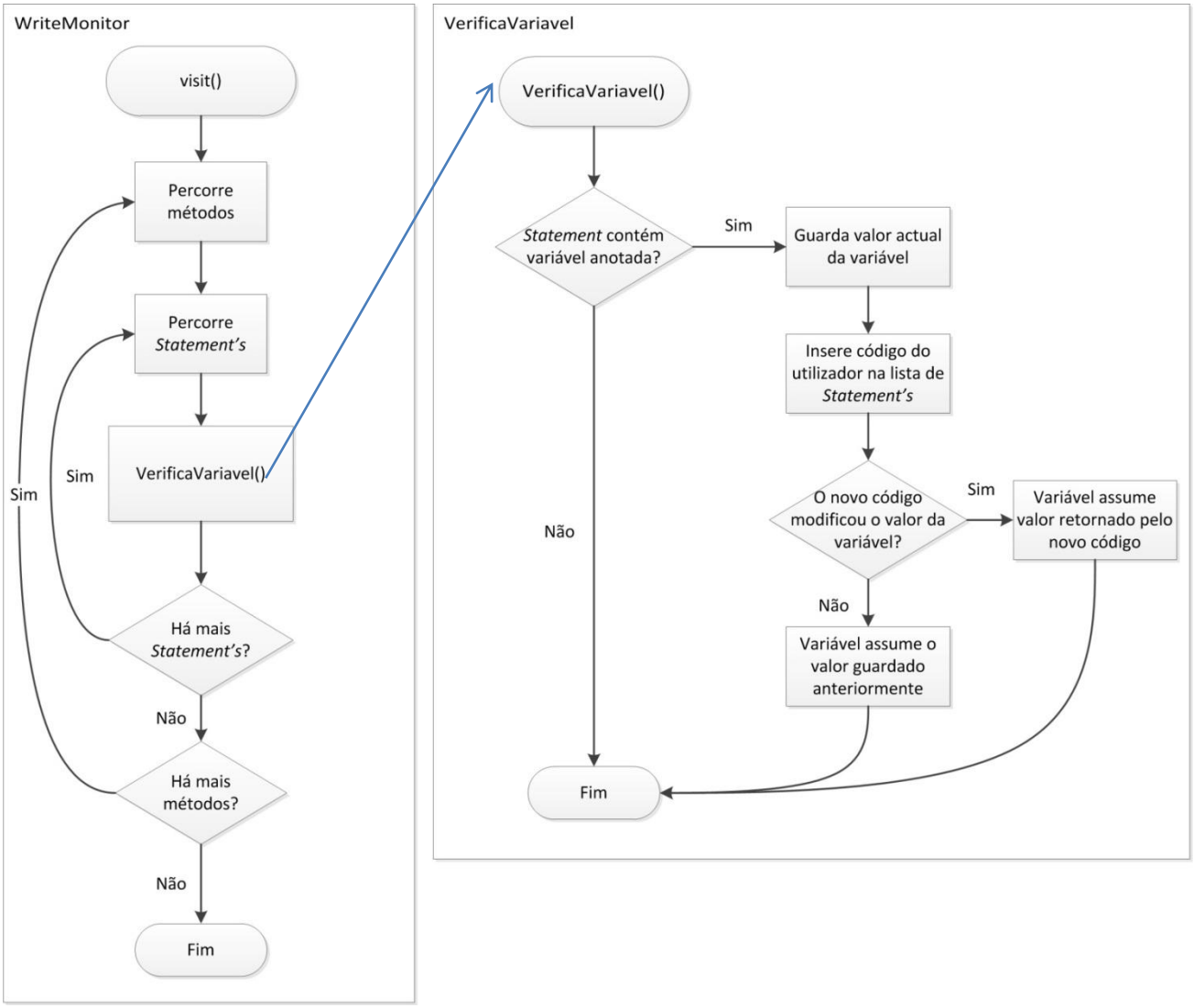

Figura 31 - Fluxograma do funcionamento do *@WriteMonitor*

Esta transformação é só um pouco mais complexa que a anterior, pois o *WriteMonitor* tem de lidar com o antigo valor da variável, e também com o novo valor da variável. Além deste facto a transformação permite ainda alterar o valor da variável através do código do utilizador, para tal basta que no código que o utilizador enviar como parâmetro na anotação contenha um ***return*** com o valor a atribuir à variável.

ReadMonitor + WriteMonitor

Graças à alteração sintáctica realizada para estas duas transformações podemos então usá-las em conjunto através da nova sintaxe, segue-se um breve exemplo:

```
monitor static String var
Read {
      println "CurrentValue: " + currentValue
}
Write {
      println "OldValue: "+oldValue+" NewValue: "+newValue
}
var ="Teste 1"
```

Figura 32 - Exemplo de utilização do *@Read/WriteMonitor* em conjunto

Como se pode observar na Figura 32 basta encadear os blocos de código do *Read* e do *Write* para as transformações funcionarem em conjunto.

O resultado do pequeno exemplo é apresentado na Figura 33:

```
OldValue: null NewValue: Teste 1
CurrentValue: Teste 1
```

Figura 33 - *Output* do exemplo do *@Read+WriteMonitor*

Estas transformações só permitem monitorizar variáveis membro de uma classe, e não variáveis locais.

5.3. MethodMonitor

Este monitor de métodos permite ao utilizador executar excertos de código antes e depois que o método a monitorizar seja executado. Esta transformação tem como objectivo implementar uma característica proveniente do *Aspect Programming*[5][6]. Neste caso refiro-me ao uso de *pointcuts* em conjunto com os *advices* que permitem executar código em pontos específicos do programa. Este ponto específico no *MethodMonitor* refere-se ao momento da chamada aos métodos, ou seja antes e depois da referida chamada, são executados os blocos de código respectivos.

Anotação

```
@MethodMonitor(before = '{Código do Before}',after= '{Código do After}')
int method(){
      (…)
}
```

Figura 34 - Anotação do *@MethodMonitor*

Esta anotação presente na Figura 34 difere um pouco das transformações anteriores, já que a anotação pode levar dois parâmetros, que se tratam do código a executar antes do método e do código a executar depois. Caso o utilizador só deseje correr um dos blocos de código da transformação, tudo o que tem de fazer é omitir o parâmetro que não pretende usar. Por exemplo se o utilizador apenas pretender executar código antes do método então só terá de preencher o parâmetro *before* da anotação, pode omitir por completo o parâmetro *after*.

Segue-se um exemplo na Figura 35 do uso desta transformação.

```
@MethodMonitor(before = '{Object target, List par -> println "Before - Class:
"+target+"; Argumentos: "+par;}',
after= '{Object target, List par -> println "After - Class: "+target+";
Argumentos: "+par; if(par[0]<10){ println "Custom Return: 50"; return 50;}}')
int test(int k, int g){
      k=23
      return k
}
println test(2,3456)
```

Figura 35 - Exemplo de utilização do *@MethodMonitor*

O primeiro bloco de código desta anotação é o que vai executar antes de o método ser invocado, e neste exemplo apenas vai ser impresso no ecrã a classe a que o método pertence, e os argumentos que estão a ser enviados para o método.

O bloco a executar depois do método vai também imprimir a classe e os argumentos, mas além disso vai controlar o valor retornado pelo método. Neste caso específico se o primeiro argumento enviado para o método for inferior a 10 o método irá devolver o valor de 50.

O exemplo descrito vai gerar o *output* ilustrado na Figura 36:

```
Before - Class: monitor.PrototipoClass@19ca40c; Argumentos: [2, 3456]
Unchanged Return of Method: 23
After - Class: monitor.PrototipoClass@19ca40c; Argumentos: [2, 3456]
Custom Return: 50
50
```

Figura 36 - *Output* gerado pelo exemplo do *@MethodMonitor*

Como o primeiro argumento enviado para a função foi inferior a 10 a transformação devolveu o valor 50 em vez de 23 que seria o valor retornado sem a transformação e como tal a transformação neste caso funciona na perfeição.

Usando agora o exemplo clássico do AspectJ está demonstrada na Figura 37 a implementação de uma aplicação que notifique o utilizador do início e fim do método anotado.

```
@MethodMonitor(before = '{Object target, List par -> println "Begin of
test()"}',after= '{Object target,Object ret, List par -> println "End of
test()";}')
int test(int i){
      println i
      i
}
test(25)
```

Figura 37 - Exemplo de utilização do *@MethodMonitor* que emula o exemplo do AspectJ

Resultará no seguinte *output* presente na Figura 38.

```
Begin of test()
25
End of test()
```

Figura 38 - *Output* do exemplo anterior

Como funciona

Ao contrário das transformações anteriores o *MethodMonitor* não tem outro método de ser utilizado sem ser através da anotação. Além disso esta transformação recebe dois parâmetros pela anotação. Um terceiro parâmetro é opcional, trata-se do parâmetro que decide se a anotação está activa ou não, se este for omisso encontra-se por defeito activa.

A anotação do *MethodMonitor* está ilustrada na Figura 39.

```
@Target({ElementType.METHOD})
@GroovyASTTransformationClass("org.codehaus.groovy.transform.MethodMonitor
AST")
public @interface MethodMonitor {
      String before () default "";
      String after () default "";
      boolean activated () default true;
}
```

Figura 39 - Anotação do *@MethodMonitor* - Código

O pseudo-código desta transformação é representado na Figura 40 - **Fluxograma do funcionamento do *@MethodMonitor***:

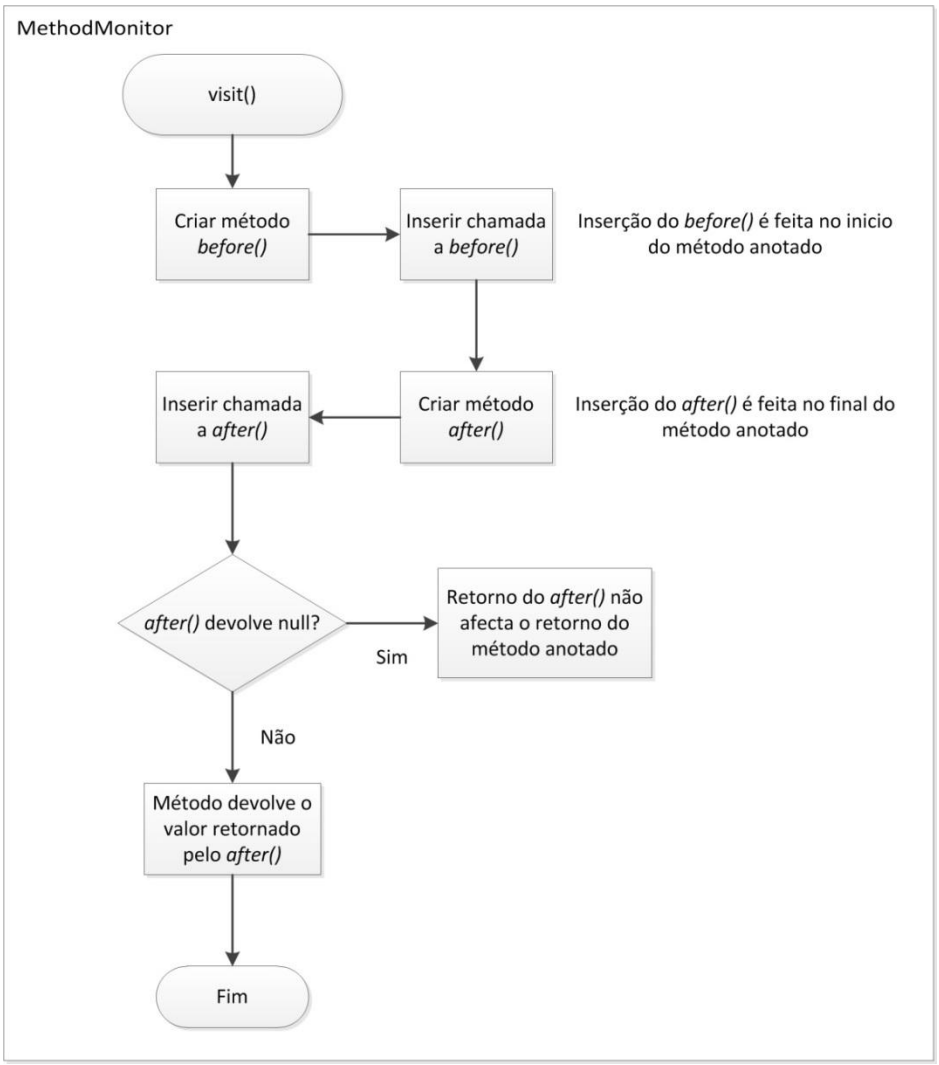

Figura 40 - Fluxograma do funcionamento do *@MethodMonitor*

Esta transformação é bastante mais complexa que as anteriores e faz uso de grande parte das ferramentas de Groovy para trabalhar com as AST's. Resumindo, esta transformação altera a lista de *Statement's* presentes no método anotado reescrevendo esta lista por completo. Tudo o que faz é inserir o código *before* antes do código já presente no método e inserir o código *after* no final do código original. No entanto e como o *after* permite alterar o valor a devolver pelo método é inserida uma verificação com o objectivo de verificar se o *after* alterou de facto o valor, caso tenha alterado devolve este novo valor, caso não tenha alterado devolve o valor inalterado.

Capítulo 6
Closures

Como já foi referido e explicado as *closures* do Groovy sofrem de algumas limitações, limitações essas que vão ser abordadas no **Capítulo 6.1** e serão discutidos os melhoramentos a implementar no **Capitulo 6.2**.

6.1. Limitações

As *closures* em Groovy têm o que por uns pode ser considerado uma limitação, e por outros pode ser considerado uma funcionalidade. Refiro-me à ausência da verificação dos tipos. Como o Groovy é uma linguagem dinâmica e tem como uma das principais características o facto de não necessitar explicitar os tipos então para alguns programadores faz sentido que as *closures* também não os verifiquem. Mas se surgir um erro e este tiver origem nos tipos então só será despoletado em *runtime*.

Os erros em *runtime* são um problema relevante, pois podem ser mais difíceis de realizar o seu *debug* e detectar a sua origem. Obrigando assim o utilizador a despender mais do seu tempo à procura de erros o que atrasa o desenvolvimento da aplicação. Portanto para alguns programadores o ideal é apanhar este erro o quanto antes, de preferência durante a fase de compilação. E para isso seria preciso que as *closures* verificassem os tipos de entrada e de saída.

Outra clara limitação do objecto *Closure* é o facto de em *runtime* não conseguirmos aceder ao código que se encontra na *closure*. Como tal foi pensada numa solução para este problema que passa por criar um campo no objecto *Closure* que guarde no formato *String* o código que se encontra na *closure* para um acesso posterior.

Outro campo útil a adicionar à *closure* será a sua assinatura que irá guardar a parametrização desta mesma, o objectivo é o de facilitar ao programador possíveis verificações em *runtime,* como por exemplo comparar os parâmetros de duas *closures*.

6.2. Melhoramentos

O objectivo dos melhoramentos a implementar será fazer com que o compilador passe a realizar uma verificação dos tipos dos parâmetros de entrada, e também verificar o tipo de retorno das *closures*. O ideal é realizar estas tarefas recorrendo ao uso de transformações, pois a sua integração com o Groovy é nativa e simplifica bastante o processo.

Verificação dos parâmetros

O Groovy de forma nativa permite a seguinte declaração de *closures* parametrizadas demonstrada na Figura 41.

```
Closure<Integer, String> clo = { int i, String s ->
      println i +" "+ s
}
```

Figura 41 - Declaração de uma *closure* parametrizada

Mas aqui acontece um acontecimento inesperado à partida. O compilador pura e simplesmente ignora todos os parâmetros inseridos, e não estabelece qualquer ligação entre

o `<Integer, String>` e o '`int i, String s`'. Para o compilador apenas irá importar tudo o que está depois da igualdade e entre as chavetas.

No entanto, o compilador guarda a parametrização da *closure*, apesar de não a usar. Então a ideia que surge aqui passa por criar uma transformação global que realize esta verificação entre a parametrização e os dados de entrada da *closure*. A transformação convém que seja global já que o objectivo será usá-la em todo o código Groovy desenvolvido.

A tarefa da transformação é então percorrer toda a árvore AST do código criado, encontrar todas as declarações de *closures* e verificar a respectiva compatibilidade entre a parametrização da *closure* e os parâmetros de entrada. Caso esta falhe deve ser despoletado um erro de compilação.

Existem agora alguns casos especiais tais como:

```
Closure<Integer,String> clo = { int i, String s -> (…) }
Closure<String,String> clo_2 = { String s1, String s2 -> (…) }
clo = clo_2
```

Figura 42 - Situação-exemplo de igualdade de *closures*

O bloco de código presente na Figura 42 no Groovy inalterado não gera erro nenhum. Mas o objectivo como é parametrizar as *closures* então esta atribuição não pode ser levada a cabo pois a declaração dos parâmetros das *closures* não é igual.

Isto significa que a transformação global tem também de verificar todas as atribuições realizadas a objectos do tipo *Closure*. Caso se verifique que a parametrização não é a correcta deve ser dado um erro ainda na fase de compilação.

Verificação do tipo de retorno

Com a verificação dos parâmetros concluída surge um novo desafio. Este desafio trata-se de definir um tipo de retorno da *Closure*. Ou seja definir o tipo de dados que pretendemos que as *Closures* que construímos vão retornar.

Esta já não é uma característica que o Groovy permita de forma nativa, ou seja, não há forma de guardar essa informação dentro do compilador. Como tal teve de se encontrar uma solução alternativa e esta solução passou por adicionar um parâmetro à parametrização das *closures*, sendo que este parâmetro seria o tipo de retorno. Decidiu-se colocar este novo parâmetro na primeira posição. Ficaria assim como ilustrado na Figura 43:

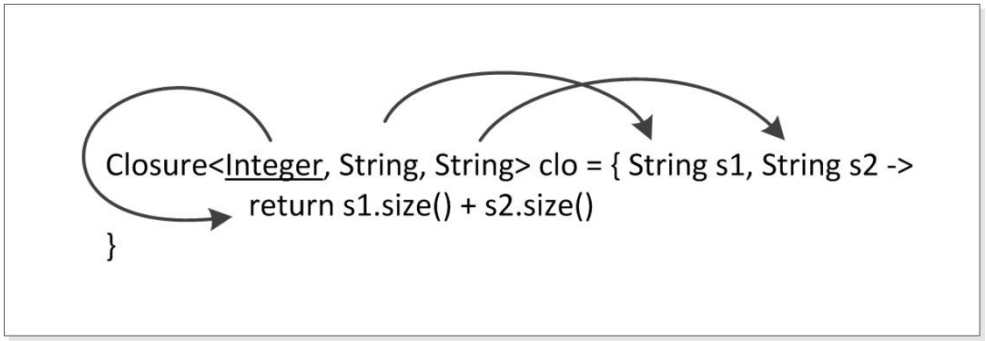

Figura 43 - Lógica da parametrização implementada

Como se pode observar, o primeiro parâmetro seria usado para especificar o tipo de retorno da *Closure*, e os restantes seriam usados para a parametrização normal que já se realizava antes.

Com isto tive então que alterar ligeiramente a transformação global que já tinha elaborado para o melhoramento anterior. Tive de adaptar para o facto dos parâmetros de entrada só aparecerem a partir do segundo parâmetro. Com essa adaptação realizada o passo seguinte foi verificar o tipo retornado pelas *closures*. Para tal o processo é o seguinte: sempre que se encontra uma *closure* a ser inicializada é percorrido o código que a inicializa e obter todos os *ReturnStatement's* e verificar o seu tipo. Caso um deles não seja do tipo definido ou de um tipo compatível com o tipo definido é logo despoletado um erro de compilação.

6.3. Dificuldades encontradas

As dificuldades encontradas na elaboração desta parte do projecto foram em número elevado. A primeira e uma das principais é o acto de percorrer toda a árvore AST gerada pelo código. A princípio pode parecer uma tarefa simples, no entanto no acto de procurar todas as *closures* dentro do código há que ter em conta algumas situações. Uma deles é a possibilidade de as *closures* poderem estar dentro de outras *closures*. Esta situação além de obrigar a verificar a parametrização e o tipo de retorno, obriga ainda a que se realize também uma procura da existência de mais *closures* dentro de outras *closures*.

Outro factor a ter em conta é a possibilidade das *closures* se encontrarem dentro de um ou vários dos ciclos disponíveis presentes na linguagem, tais como o *for*, *while*, ou dentro de condições como o *if* e o *switch/case*. O problema destes ciclos é que todos eles contêm a sua própria lista de *Statement's* que tem de ser acedida e analisada. O *if* por exemplo até contém duas listas, uma para o *if* e outra para o *else*. O *switch* contém tantas listas de *Statement's* quanto o número de *case's*.

O que dificulta a tarefa quando estas condições e ciclos estão todos encadeados entre si. Como por exemplo um *if* dentro de um *for* que por sua vez contém outro *for*, e por aí adiante. Então a solução foi criar funções que sejam completamente recursivas de modo a percorrer todo e qualquer objecto da árvore AST.

Foram necessários dois tipos de funções com características recursivas. Isto porque o modo de tratar as *closures* encadeadas é diferente do tratamento de ciclos. Ambas as funções deram ainda bastantes problemas menores de implementação e compatibilidades, já que o objectivo é também que a transformação não se torne demasiado pesada e portanto seja o mais flexível possível.

A ideia inicial era bastante simples, no entanto não fazia prever à primeira vista as dificuldades que iriam aparecer e que exigiram uma correcção constante dos *bugs* que surgiam pelo caminho.

6.4. Funcionamento da transformação

Para um melhor entendimento sobre esta transformação segue-se uma explicação de como se encontra estruturada na forma de fluxogramas.

A primeira função a ser executada é a *ClosuresVisitor()* e o seu funcionamento está ilustrado na Figura 44 - **Fluxograma do *ClosuresVisitor***:

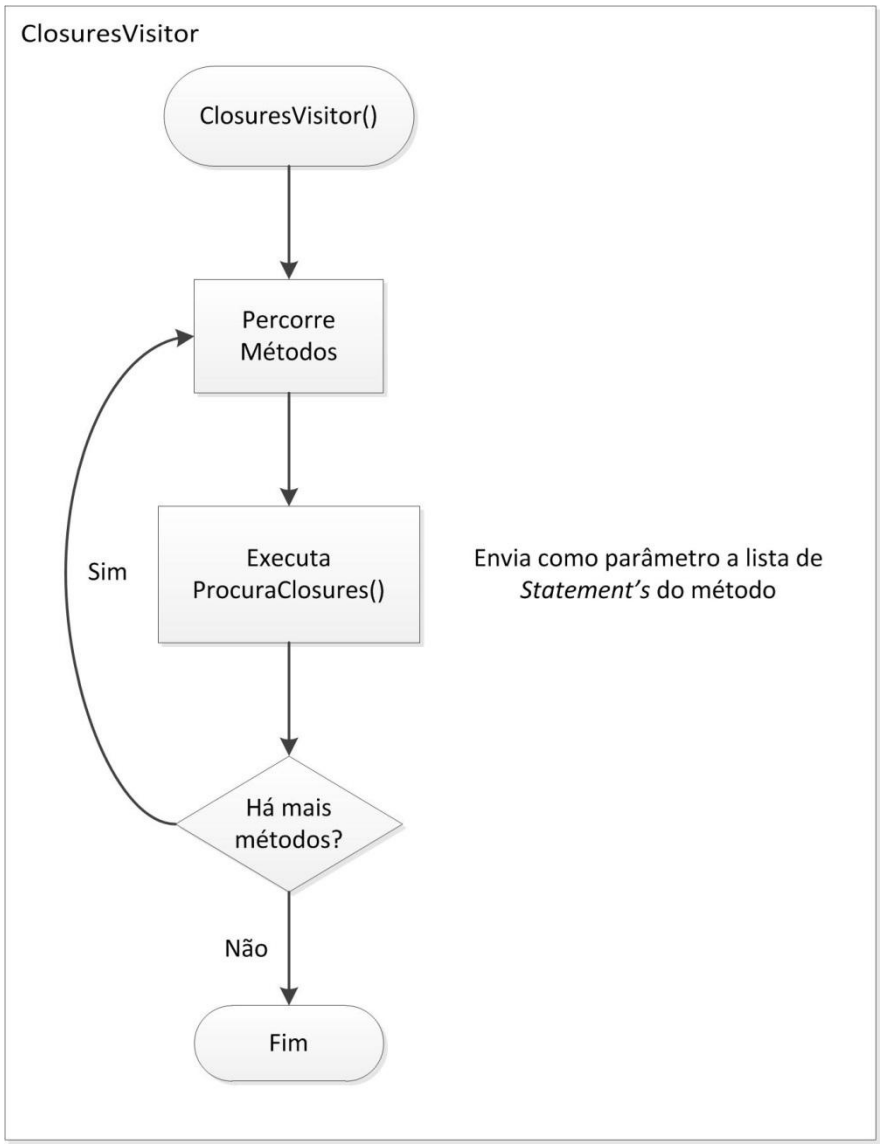

Figura 44 - Fluxograma do *ClosuresVisitor*

Esta função é muito simples, simplesmente percorre todos os métodos presentes no *script* ou classe e executa a função *ProcuraClosures()* em cada método. Para a *ProcuraClosures()* é enviado como parâmetro a lista de *Statement's* que o método possui.

A função termina a partir do momento que o ciclo já tenha percorrido todos os métodos existentes e tenha analisado todos os *Statement's* nestes inseridos.

De seguida apresenta-se um fluxograma da *ProcuraClosures()* ilustrado na Figura 45 - **Fluxograma do *ProcuraClosures***:

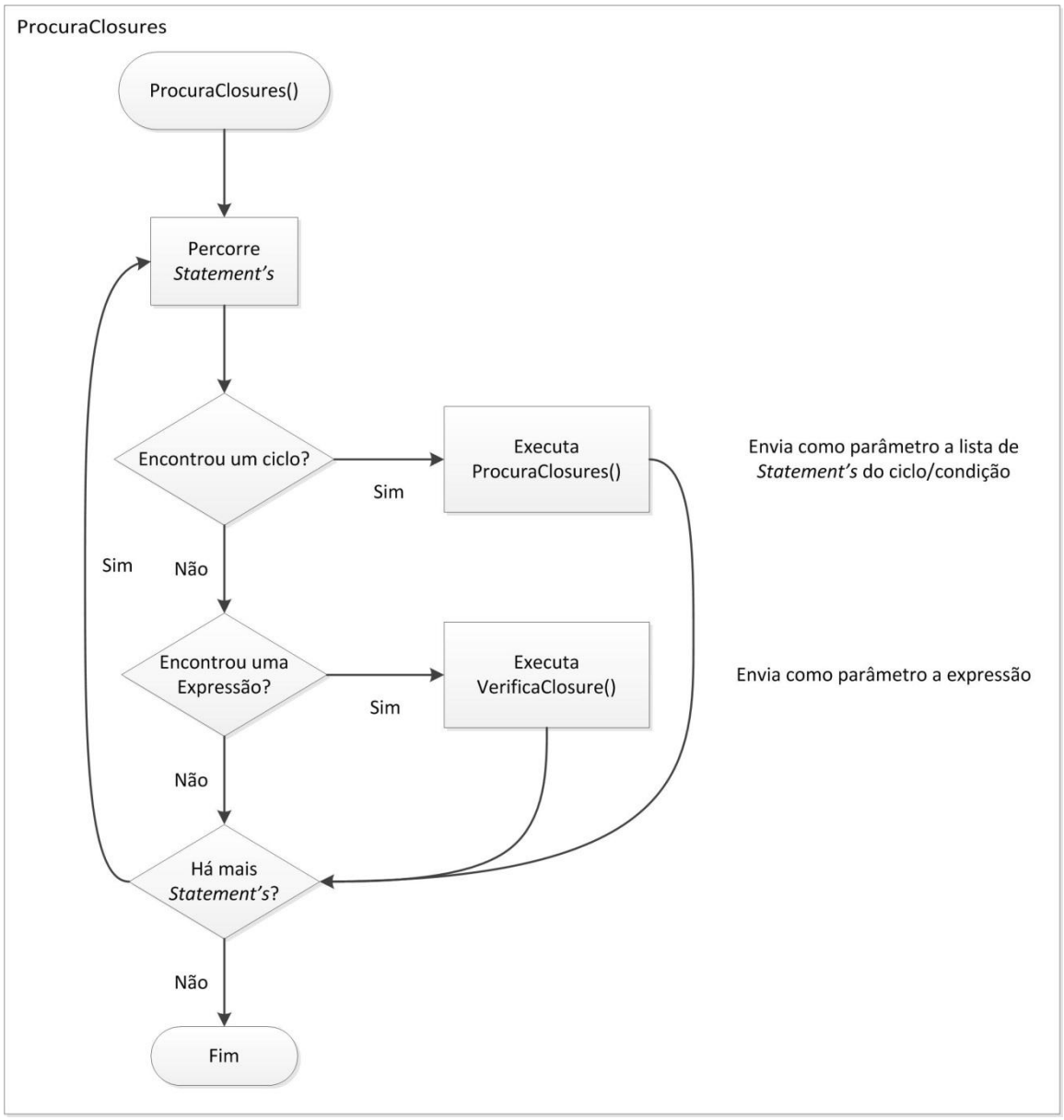

Figura 45 - Fluxograma do *ProcuraClosures*

Esta função é fundamental em todo o funcionamento da transformação, pois ela vai ser frequentemente chamada.

A função começa por percorrer todos os *Statement's* que recebe como parâmetro, e para cada um deles vai verificar se se trata de um ciclo ou de uma expressão. No caso de se tratar de um ciclo, ou de uma condição como o *if* ou o *switch*, a função vai-se chamar novamente a si própria enviando como parâmetro a lista de *Statement's* que se encontra presente nos ciclos e condições referidas.

Caso se trate de uma expressão, o objectivo é descobrir se esta se trata de uma *closure* e como tal se tem de ser tratada ou não. Este tratamento é realizado efectuando a chamada da função *VerificaClosure()* e enviando como parâmetro a expressão em causa.

Resta então explicar o funcionamento da *VerificaClosure()* ilustrado pela Figura 46 - **Fluxograma do *VerificaClosure***:

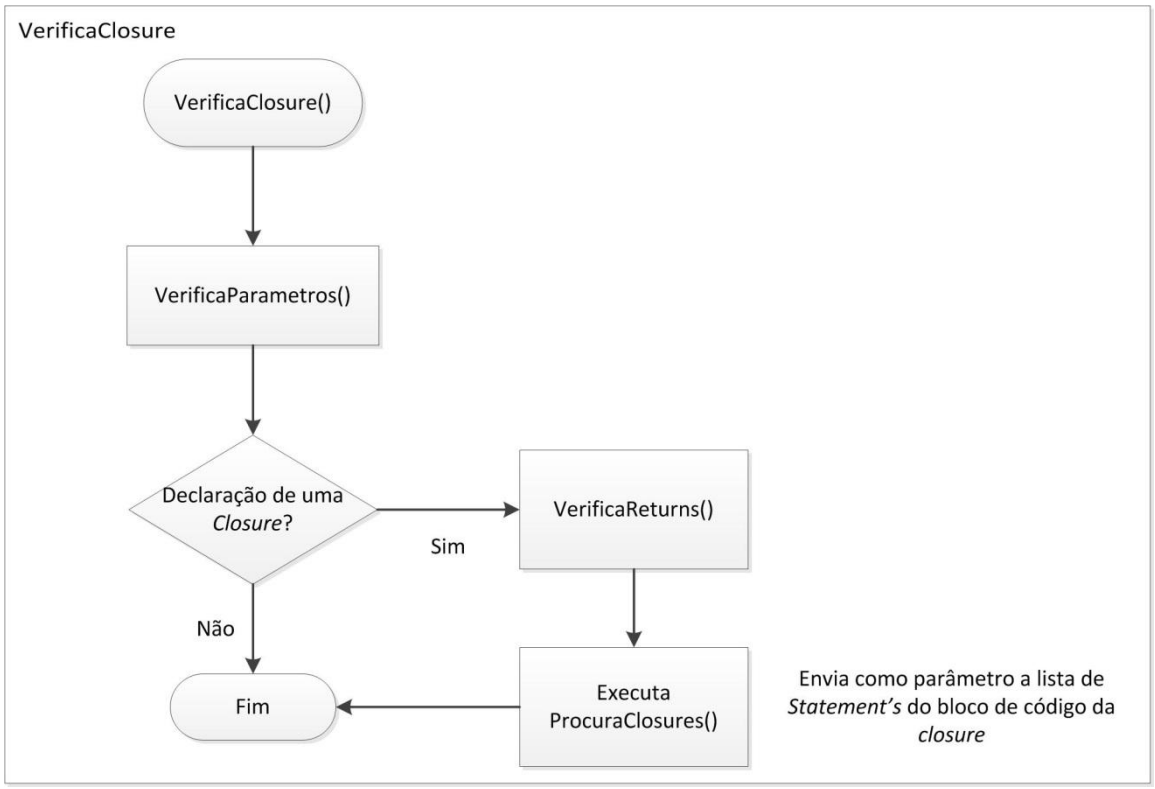

Figura 46 - Fluxograma do *VerificaClosure*

Esta função é bastante simples comparativamente à anterior. Tudo o que realiza é a verificação de parâmetros chamando a função *VerificaParametros()*, esta função faz exactamente o que o nome indica, verifica se a parametrização das *closures* confere com os tipos de entrada ou não, no caso de não conferir a função despoleta um erro.

De seguida a função *VerificaClosure()* confirma se a expressão se trata de uma declaração de uma *closure* ou não. Caso se trate de uma declaração será chamada a função *VerificaReturns()*, que vai assegurar que todos os *returns* presentes no código da *closure* correspondem com o tipo definido para o *return* na parametrização da *closure*.

Após a verificação do tipo de retorno temos que averiguar se dentro da *closure* não existem por sua vez outras *closures* declaradas, e para tal chamamos novamente a *ProcuraClosures()* para realizar esta tarefa.

Como se pode observar esta transformação tem muitas características recursivas, isto é necessário de modo a manter uma consistência de funcionamento para qualquer código Groovy.

6.5. Novos métodos e campos do objecto *Closure*

Com o objectivo de facilitar a visualização do código presente numa *closure* e de guardar uma assinatura com a parametrização da *closure* foram adicionados os seguintes campos:

- *String* codeClosure – Servirá para guardar o código presente na *closure* no formato de uma *String* para assim o programador ter fácil acesso a este campo, bastando invocar a função: *getCodeClosure()*

- *ArrayList<Object>* signatureClosure – Este campo vai guardar a parametrização pela qual a *closure* foi inicializada. O programador pode então aceder a esta invocando a nova função: *getSignatureClosure()*. Sendo que da lista devolvida o primeiro objecto é o do tipo de retorno da *closure()*.

A criação destes campos e métodos exige que se altere directamente o código fonte da linguagem, mais concretamente a classe *Closure.java* que se encontra no *package*: *groovy.lang*.

```
private String codeClosure;
private ArrayList<Object> signatureClosure = new ArrayList<Object>();

public void addToSignatureClosure(Object o){
      signatureClosure.add(o);
}
public ArrayList<Object> getSignatureClosure(){
      return signatureClosure;
}
public void _setCodeClosure(String s){
      this.codeClosure = s;
}
public String getCodeClosure(){
      return this.codeClosure;
}
```

Figura 47 - Código a inserir no *Closure.java*

Foi este o código (presente na Figura 47) adicionado ao ficheiro *Closure.java*. De seguida basta recompilar a linguagem para estes campos e métodos se encontrarem disponíveis.

No entanto para fazer uso destas funcionalidades é necessário realizar as chamadas às funções *addToSignatureClosure()* e *_setCodeClosure()* durante a execução da transformação global das *closures*. Esta chamada é realizada de forma transparente no momento em que existe uma declaração de uma *closure* para assim o utilizador poder aceder imediatamente aos novos campos já devidamente preenchidos. A função *_setCodeClosure()* é ainda chamada quando é atribuída uma *closure* a outra. Já que aqui o código que corresponde às *closures* vai alterar e como tal tem de ser actualizado.

Um exemplo da chamada das novas funções é o que se encontra ilustrado na Figura 48.

```
Closure<Integer,String> clo = { int i, String s -> (…) }
println clo.getCodeClosure()
println clo.getSignatureClosure()
Output:
{ int i, String s -> (…) }
[Integer, String]
```

Figura 48 - Exemplo de utilização dos novos métodos da *Closure*

Capítulo 7
Publish/Subscribe

O *publish/subscribe* implementado neste trabalho é baseado em listas de subscritores. Isto significa que para cada evento a ser publicado existe uma lista de subscritores interessados neste evento específico. Como tal, sempre que o *publisher* quer enviar determinado evento, basta aceder à lista de subscritores e notifica-los a todos do evento a ser enviado.

Os *subscribers* efectuam a subscrição junto dos *publishers* enviando a classe que pretendem subscrever. Quando um subscritor já não pretender receber actualizações de determinado evento basta efectuar o *unsubscribe* mais uma vez junto do *Publisher* e a partir desse momento o *publisher* não o irá notificar mais.

7.1. Publish/Subscribe no Groovy

No âmbito deste trabalho também foi criada uma arquitectura de comunicação baseada no *publish/subscribe*. Que funciona de forma em tudo semelhante à ideia original do paradigma[7][37].

A implementação em Groovy consistiu em usar transformações que de forma invisível ao utilizador criem os métodos necessários para a criação da infraestrutura de comunicação.

Foi então necessário proceder à criação de duas transformações, cujas respectivas anotações são: *@Publish* e *@Subscribe* que serão explicadas de seguida.

7.2. Publish

Esta transformação tem como objectivo ser usada nas classes que vão publicar, ou seja nos *publishers*. A respectiva anotação recebe uma lista das classes a publicar. A anotação deverá ser usada como demonstrado na Figura 49.

```
@Publish([A, B, C])
class A{
}
```

Figura 49 - Anotação do *@Publish*

Neste exemplo a classe A vai ser a publicadora dela própria, da classe B e da classe C.

A transformação vai entrar em acção, e irá primeiro criar uma lista de subscritores para as classe A, B e C. De seguida o objectivo será criar um conjunto de métodos que a classe publicadora tem de possuir. Neste caso os métodos serão os seguintes:

- ***publish(Class)*** **–** Este método recebe a classe a ser publicada (que tem obrigatoriamente de ser uma das classes definidas na anotação), e torna possível o envio de notificação aos subscritores dessa classe.
- ***unPublish(Class)*** **–** Este método funciona de forma semelhante ao de cima, mas com o objectivo de terminar as notificações.
- ***send(Object)*** **–** O método *send()* é o método usado para notificar os subscritores da classe do objecto enviado como parâmetro. Caso a classe já tenha sido publicada através do *publish()* o objecto será enviado a todos os seus subscritores.

7.3. Subscribe

O funcionamento desta anotação é semelhante ao funcionamento da anotação acima descrita, sendo que a anotação também recebe como parâmetro uma lista de classes que o objecto pretende subscrever.

```
@Subscribe([A, B, C])
class D{
}
```

Figura 50 - Anotação do *@Subscribe*

O *@Subscribe* tal como o *@Publish* vai criar métodos de forma imperceptível para o utilizador, esses métodos serão:

- ***subscribe(Object, Class)*** **–** Neste método é passado no primeiro parâmetro o objecto que está a publicar, e no segundo parâmetro é enviada a classe que se pretende subscrever. Esta classe tem de estar explicita como parâmetro na anotação.
- ***unsubscribe(Object, Class)*** **–** Este método permite cancelar a subscrição feita pelo método anterior, sendo que os parâmetros seguem a mesma lógica dos parâmetros usados no método anterior.

No entanto as classes que forem anotadas com *@Subscribe* necessitam de algum código escrito pelo próprio utilizador. Esse código será o método que vai receber as notificações dos *publishers*. Este método tem de se chamar *receive()* e levar como parâmetro um objecto do tipo da classe subscrita. Ou seja caso a classe subscreva várias outras tem de haver tantos métodos *receive()* quantas as classes subscritas.

Tendo em conta o exemplo dado na Figura 50 a classe '*D*' neste caso teria de ficar do modo ilustrado na Figura 51.

```
@Subscribe([A, B, C])
class D{
	void receive(A a){
	}
	void receive(B b){
	}
	void receive(C c){
	}
}
```

Figura 51 - Exemplo de uma classe anotada com *@Subscribe*

Como a classe do exemplo subscreve outras três classes, teve de possuir três métodos *receive()*, cada um com um parâmetro do tipo de cada classe subscrita. Caso isto não aconteça a transformação informará o utilizador da falta de algum destes métodos através de um erro de compilação.

7.4. Exemplo

Existem 2 objectos, *C1* e *C2*. Ambos os objectos possuem um conjunto de coordenadas que indicam a sua posição a um dado instante. Agora imagine-se que um outro conjunto de objectos pretende estar actualizado sobre a posição de *C1* e *C2* sempre que estas mudem. Estes objectos, que vamos chamar de *A*, *B* e *D*, vão ter de subscrever o objecto do qual pretendem receber as actualizações.

Isto pode ser ilustrado pelo seguinte cenário presente na Figura 52 - **Ilustração do funcionamento do *publish/subscribe***:

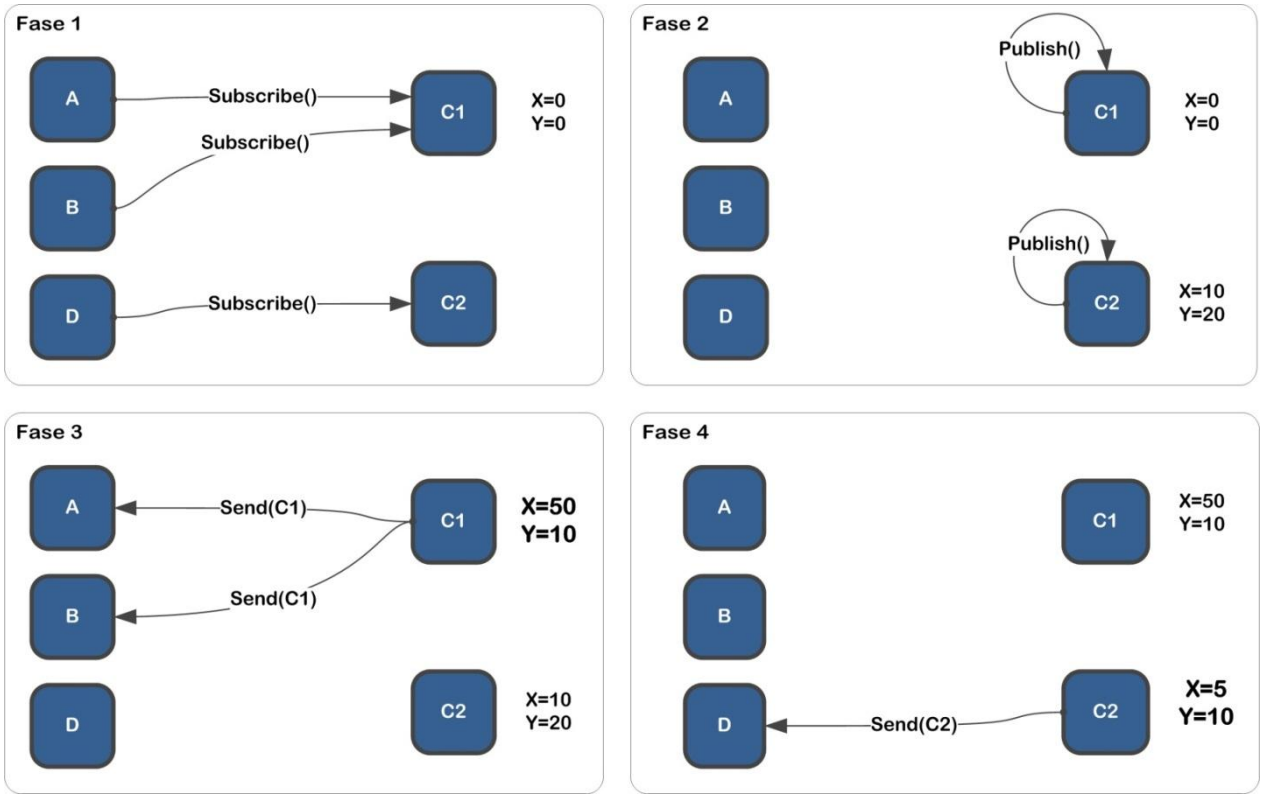

Figura 52 - Ilustração do funcionamento do *publish/subscribe*

Na fase inicial os objectos *A* e *B* subscrevem a classe *C1* e o objecto *D* subscreve a classe *C2*. Mas nesta altura mesmo que *C1* e *C2* mudem de posição ainda não notificam os seus subscritores. Isto porque ainda não realizaram o *publish()* de si próprios tal como sucede na fase 2 deste exemplo. Na terceira fase verificamos que *C1* obtém um novo conjunto de coordenadas, então o objecto trata de notificar os seus subscritores *A* e *B*. Na quarta e ultima fase deste exemplo é o objecto *C2* que obtém novas coordenadas e portanto notifica o objecto *D* que é o único que está subscrito a *C2*.

O que terá de acontecer é que a classe *C1* e *C2* terão de ser anotadas com *@Publish([Cx])* e os restantes objectos que pretenderem subscrever terão de ser anotados com por exemplo *@Subscribe([C1])* para subscreverem *C1*.

O exemplo ilustrado ficaria da seguinte forma ilustrado pela Figura 53:

```
@Publish([C1])
class C1{
	int x,y
}
@Publish([C2])
class C2{
	int x,y
}
@Subscribe([C1])
class A {
	void receive(C1 coord){ (...) }
}
@Subscribe([C1])
class B {
	void receive(C1 coord){ (...) }
}
@Subscribe([C2])
class D {
	void receive(C2 coord){ (...) }
}
```

Figura 53 - Exemplo de 2 classes publicadoras e 3 classes que vão subscrever

Como se pode observar iriam estar duas classes a publicarem-se a si mesmas, *C1* e *C2*. E iriamos ter três classes subscritoras. Duas delas, *A* e *B*, subscrevem *C1*. E a classe *D* subscreve *C2*. De relembrar que cada classe que subscreve tem o respectivo método *receive()* escrito pelo utilizador. Os restantes métodos são adicionados dinamicamente a cada classe durante a compilação e portanto não necessitam de ser explicitamente escritos no código.

7.5. Alteração Sintáctica

Tal como em transformações referidas anteriormente como o *@ReadMonitor* e o *@WriteMonitor*, viu-se aqui uma oportunidade de utilizar estas transformações omitindo as anotações. Para isso procedeu-se então a uma inovação sintáctica na linguagem de modo a simplificar o uso do paradigma.

A nova sintaxe para definir uma classe como *publisher* será a definida na Figura 54.

```
class Coord publishes Coord{
	int x,y
}
```

Figura 54 - Declaração de uma classe que publica *Coord*

Como se pode observar alteração sintáctica funciona de forma semelhante à sintaxe do *extends* e do *implements* já presentes no Java. Ou seja, será só usar a nova palavra-chave *publishes* seguida das classes que se pretende publicar.

Já para definir uma classe como *subscriber* será necessário usar a sintaxe ilustrada na Figura 55.

```
class A subscribes Coord{
	void receive(Coord coord){ (...) }
}
```

Figura 55 - Declaração de uma classe que subscreve a *Coord*

Ambas as sintaxes podem ainda ser usadas em simultâneo, não importando a ordem em que aparecem no código. O uso do *publishes* e do *subscribes* também não impede o uso normal do *extends* e do *implements* por parte do utilizador tendo uma compatibilidade total.

O exemplo anterior caso use a sintaxe alterada ficaria da forma ilustrada pela Figura 56.

```
class C1 publishes C1{
      int x,y
}
class C2 publishes C2{
      int x,y
}
class A subscribes C1{
      void receive(C1 coord){ (...) }
}
class B subscribes C1{
      void receive(C1 coord){ (...) }
}
class D subscribes C2{
      void receive(C2 coord){ (...) }
}
```

Figura 56 - Exemplo do uso da nova sintaxe para declarar classes que implementem o *publish/subscribe*

Acredito então que esta nova sintaxe traz uma maior legibilidade do código além de simplificar o uso do paradigma *publish/subscribe*.

Capítulo 8
Plano de Trabalho e Resultados Obtidos

Nesta secção vou descrever todas as tarefas realizadas e o tempo despendido a realizá-las. Além disso será feita uma comparação com o planeamento inicial realizado no 1º semestre. Serão identificados os casos em que as tarefas tenham demorado mais ou menos tempo que o esperado e serão explicadas as razões que justifiquem a alteração figurada no 2º semestre.

8.1. Planeamento e objectivos iniciais (1º Semestre)

Inicialmente o plano de trabalhos elaborado foi o que se encontra presente na Figura 57.

Id da Tarefa	Tarefa
T1	Estudo/compreensão e aplicação do Antlr
T2	Implementação dos Métodos referidos: Read/WriteMonitor e MethodMonitor
T3	Alteração das Closures
T4	Implementação do paradigma Publish/Subscribe
T5	Criação de novas transformações que tenham sido pensadas nas semanas anteriores e se revelem úteis
T6	Escrita do relatório

Figura 57 - Tabela relativa às tarefas planeadas durante o 1º Semestre

O gráfico simplificado de Gantt correspondente às tarefas está ilustrado na Figura 58.

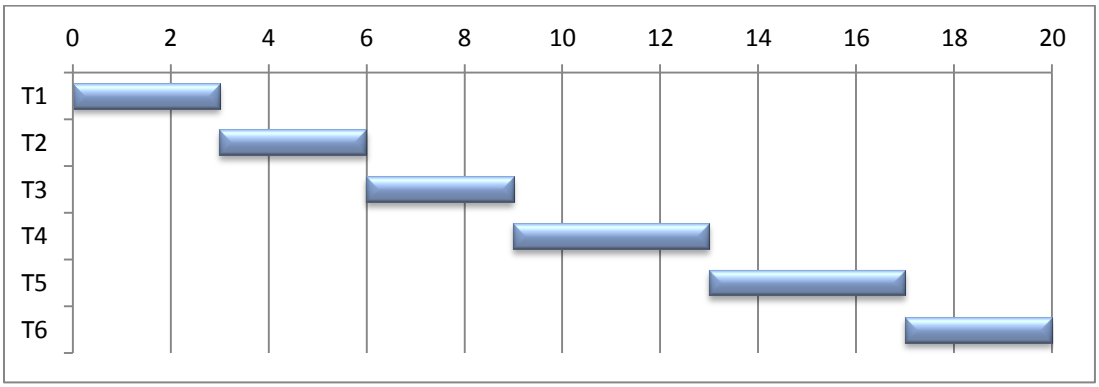

Figura 58 - Gráfico de Gantt construído com base nas tarefas planeadas para o 2º semestre

Este planeamento foi realizado com a estimativa de duração do estágio de 20 semanas. No entanto acabou por não se revelar completamente fiel ao trabalho efectivamente realizado e respectiva duração, pois teve de sofrer algumas alterações que são explicadas na secção seguinte.

8.2. Plano de trabalho efectivo (2º Semestre)

As tarefas realizadas durante este segundo semestre foram as ilustradas pela Figura 59.

Id da Tarefa	Tarefa
T1	Estudo/compreensão e aplicação do *Antlr* e da *AntlrParserPlugin*
T2	Implementação das transformações propostas: Read/WriteMonitor e MethodMonitor
T3	Adaptação das transformações a uma nova sintaxe
T4	Alteração das Closures
T5	Implementação do paradigma *publish/subscribe*
T6	Adaptar o *publish/subscribe* a uma nova sintaxe
T7	Compilação da linguagem com as novas funcionalidades
T8	Escrita do relatório

Figura 59 - Plano de trabalhos levado a cabo no 2º Semestre

O planeamento ao nível das tarefas a realizar não se alterou de forma radical, apenas se adicionou a tarefa que trata da adaptação sintáctica do paradigma *publish/subscribe* e também a tarefa relativa à compilação da nova linguagem com todas as novas funcionalidades incluídas.

Apenas uma ressalva relativamente ao facto de que no planeamento inicial estava prevista a criação de novas transformações mais perto do final do estágio, e isto não foi realizado devido ao facto de algumas tarefas terem demorado mais tempo que o esperado e portanto não ter sobrado tempo suficiente.

O tempo despendido para a realização de cada tarefa e subtarefa ao longo deste projecto está ilustrado no **Anexo nº 1** deste documento.

Após análise do gráfico em anexo observa-se que só a tarefa 5 referente à implementação do *publish/subscribe* demorou o tempo esperado inicial de 4 semanas. Todas as outras demoraram mais do que era previsto numa fase inicial com destaque para a criação das novas transformações e para as alterações das *closures*, que eram duas tarefas com um tempo previsto de 3 semanas e ambas demoraram 5 semanas a serem realizadas.

Esta alteração no tempo de execução das tarefas justifica-se com o surgimento de dificuldades imprevistas durante a elaboração das mesmas.

Capítulo 9
Trabalho Futuro

No que respeita ao trabalho a realizar no futuro acredito que existem imensas possibilidades. Tendo como base o trabalho realizado as *closures* podem sofrer mais melhoramentos, como por exemplo investigar como as alterar e melhorar directamente no código-fonte da linguagem para assim as alterações às *closures* serem incorporadas de forma nativa. Ou seja, o resultado final pretendido seria que sem recorrer a transformações locais ou globais se pudesse realizar a verificação de tipos em cada *Closure*.

Outra característica bastante útil a implementar seria possibilitar o retorno de mais do que uma variável por parte dos métodos e *closures*. Da mesma forma que por exemplo o *Python* permite. Isto iria revelar-se uma alteração bastante útil para a comunidade programadora de Groovy.

Neste momento o *publish/subscribe* encontra-se a funcionar apenas numa só máquina e com trocas de informação directamente entre objectos. Para trabalho futuro uma proposta seria melhorar esta implementação do *publish/subscribe* de modo a que funcionasse numa rede local entre vários computadores, isto traria uma flexibilidade bastante útil à utilização do paradigma por parte da linguagem.

Durante o projecto fui desenvolvendo uma ferramenta que automatiza a compilação da linguagem com as novas funcionalidades desenvolvidas. Para implementar as novas funcionalidades é necessário modificar o código-fonte da linguagem em várias localizações e inserir todos os ficheiros das transformações num local específico, como tal a ferramenta torna-se bastante útil. No entanto ela está apenas preparada para a versão 1.8.6 do Groovy que era a versão mais actualizada na data de início, e à data de hoje já existe a versão 2.0 com algumas novas funcionalidades. Portanto o objectivo seria continuar a evoluir esta ferramenta para a adaptar de modo a que permita migrações para as novas versões do Groovy.

Capítulo 10
Conclusões

Esta investigação teve como objectivo inicial o desenvolvimento de uma versão alterada do compilador de Groovy. Este novo compilador tinha de ter certas características que acredito que sejam úteis e melhorem a experiencia do programador de Groovy. Deste modo a criação desta versão alterada do compilador foi um sucesso.

Na fase inicial da investigação foram investigadas as formas possíveis de realizar as alterações pretendidas. Esta foi uma fase muito importante pois definiu quais as características passíveis de implementar no compilador. Toda esta investigação foi realizada durante o primeiro semestre.

A fase seguinte da investigação consistiu em implementar as funcionalidades cuja implementação se revelou exequível. Toda esta fase foi realizada durante o segundo semestre.

Concluindo, este estágio contribuiu bastante a nível profissional com a obtenção de conhecimentos numa área da qual eu tinha curiosidade e interesse. Obrigou-me a desenvolver a capacidade de auto-aprendizagem e de pesquisa/investigação e sempre com o apoio do orientador que se mostrou sempre disponível. Ao longo da investigação tive de compreender parte dos algoritmos e blocos de código que constroem o compilador de Groovy, isto por si só é uma tarefa difícil devido à complexidade de um compilador. Por fim, acredito que com os conhecimentos e experiência que ganhei estou mais preparado para encarar o futuro e o mundo profissional e portanto considero a realização desta investigação um sucesso.

Referências

[1] Groovy. Retrieved July 5, 2012, from http://groovy.codehaus.org/

[2] Groovy (programming language). Retrieved July 5, 2012, from http://en.wikipedia.org/wiki/Groovy_%28programming_language%29

[3] König, D. (2011). *Groovy in Action, Second Edition*. Manning. Retrieved from http://en.wikipedia.org/wiki/Groovy_(programming_language)

[4] TIOBE Software: Tiobe Index. Retrieved July 5, 2012, from http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[5] A look at aspect-oriented programming. (2004, February 17). Retrieved from http://www.ibm.com/developerworks/rational/library/2782.html

[6] Aspect-oriented programming. Retrieved July 5, 2012, from http://en.wikipedia.org/wiki/Aspect-oriented_programming

[7] Publish–subscribe pattern. Retrieved July 5, 2012, from http://en.wikipedia.org/wiki/Publish–subscribe_pattern

[8] The AspectJ Project. Retrieved July 5, 2012, from http://www.eclipse.org/aspectj/

[9] Groovy - Compile-time Metaprogramming - AST Transformations. Retrieved July 5, 2012, from http://groovy.codehaus.org/Compile-time+Metaprogramming+-+AST+Transformations

[10] Colyer, A., Clement, A., Harley, G., & Webster, M. (2004). *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools* (1st ed.). [[Addison-Wesley Professional]].

[11] Laddad, R. (2009). *AspectJ in Action: Enterprise AOP with Spring* (2nd ed.). [[Manning Publications]].

[12] Kiczales, G., Lamping, J., & Mendhekar, A. (1997). Aspect-oriented programming. - *Oriented Programming*, (June). Retrieved from http://www.springerlink.com/index/X535M642082K783R.pdf

[13] Learn AspectJ Using Eclipse AJDT | 2 | WebReference. Retrieved July 5, 2012, from http://www.webreference.com/programming/aspectj/2.html

[14] Pointcuts. Retrieved July 5, 2012, from http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html

[15] Advice. Retrieved July 5, 2012, from http://www.eclipse.org/aspectj/doc/released/progguide/semantics-advice.html

[16] Inter-type declarations. Retrieved July 5, 2012, from http://www.eclipse.org/aspectj/doc/released/progguide/language-interType.html

[17] Eclipse - The Eclipse Foundation open source community website. Retrieved July 5, 2012, from http://www.eclipse.org/

[18] Publish–subscribe pattern - Wikipedia, the free encyclopedia. Retrieved July 5, 2012, from http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

[19] Publish/Subscribe. Retrieved July 5, 2012, from http://msdn.microsoft.com/en-us/library/ff649664.aspx

[20] Gamma, E., Helm, R., Johnson, R., & Vissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (p. 395). Addison-Wesley.

[21] Java Swing | Free Java Guide & Tutorials. Retrieved July 5, 2012, from http://www.freejavaguide.com/java_swing.html

[22] Closures. Retrieved July 5, 2012, from http://en.wikipedia.org/wiki/Closure_%28computer_science%29

[23] Groovy - Closures. Retrieved July 5, 2012, from http://groovy.codehaus.org/Closures

[24] Geeky Articles: What's Cooking in Java 8 - Project Lambda. Retrieved July 5, 2012, from http://www.geekyarticles.com/2012/01/whats-cooking-in-java-8-project-lambda.html

[25] From Java to Groovy in a few easy steps | Groovy Zone. Retrieved July 5, 2012, from http://groovy.dzone.com/news/java-groovy-few-easy-steps

[26] Groovy - Dynamic Groovy. Retrieved July 5, 2012, from http://groovy.codehaus.org/Dynamic+Groovy

[27] MetaProgramming with Groovy I | Groovy Zone. Retrieved July 5, 2012, from http://groovy.dzone.com/articles/metaprogramming-groovy-i

[28] Abstract Syntax Tree. Retrieved January 22, 2012, from http://c2.com/cgi/wiki?AbstractSyntaxTree

[29] Java Virtual Machine. Retrieved July 5, 2012, from http://en.wikipedia.org/wiki/Java_virtual_machine

[30] Groovy - Compiler Phase Guide. Retrieved July 5, 2012, from http://groovy.codehaus.org/Compiler+Phase+Guide

[31] ANTLR Parser Generator. Retrieved January 22, 2012, from http://www.antlr.org/

[32] AntlrParserPlugin. Retrieved July 5, 2012, from http://groovy.codehaus.org/api/org/codehaus/groovy/antlr/AntlrParserPlugin.html

[33] Groovy Web Console - custom antlr parser plugin. Retrieved July 5, 2012, from http://groovyconsole.appspot.com/view.groovy?id=3

[34] Groovy - Global AST Transformations. Retrieved January 22, 2012, from http://groovy.codehaus.org/Global+AST+Transformations

[35] Groovy - Local AST Transformations. Retrieved January 22, 2012, from http://groovy.codehaus.org/Local+AST+Transformations

[36] Apache Ant. Retrieved July 5, 2012, from http://ant.apache.org/

[37] Eugster, P. T., Felber, P. a., Guerraoui, R., & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, *35*(2), 114-131. doi:10.1145/857076.857078

Anexos

Anexo nº 1

Gráfico de Gantt do Segundo Semestre