PROGRAMAÇÃO ORIENTADA A CARACTERÍSTICAS EM GROOVY

GUILHERME HENRIQUE DE ASSIS

PROGRAMAÇÃO ORIENTADA A CARACTERÍSTICAS EM GROOVY

Dissertação apresentada ao Programa de Pós-Graduação em Ciências da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciências da Computação.

ORIENTADOR: EDUARDO FIGUEIREDO

Belo Horizonte

Dezembro de 2016

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Assis, Guilherme Henrique de.

A848p Programação orientada a características em Groovy. / Guilherme Henrique de Assis. – Belo Horizonte, 2016.
xx, 74 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Eduardo Magno Lages Figueiredo.

1. Computação - Teses. 2. Groovy ((Linguagem de programação de computador). 3. Linha de produtos de software. I. Orientador. II. Título.

CDU 519.6*32 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Programação orientada a características em Groovy

GUILHERME HENRIQUE DE ASSIS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Eduardo M Figueiredo
PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCELO DE ALMEIDA MAIA
Departamento de Ciência da Computação - UFU

Bigonha
PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 14 de dezembro de 2016.

Dedico este trabalho a minha esposa Eduarda, pelo apoio e incentivo dado durante todo este processo.

Agradecimentos

Agradeço ao meu orientador, meus colegas e professores pelo aprendizado adquirido e convivência. Agradeço aos meus pais, irmão e a minha esposa, pelo constante apoio e incentivo.

"Uma jornada de mil milhas começa com um único passo."
(Lao-Tzu)

Resumo

Linhas de Produtos de Software (LPS) estão cada vez mais presentes no mercado de software, por se tratar de um paradigma de desenvolvimento de software que tem como objetivo reutilizar como forma de reduzir custos e ganhar agilidade. LPS é um conjunto de sistemas de software que compartilham funcionalidades em comum que satisfazem um segmento de mercado. Há diversas formas de implementar uma LPS, sendo a Programação Orientada a Características (POC) uma técnica proposta para melhorar a modularidade e flexibilidade de uma LPS. A ideia básica do POC é decompor o software em partes menores, as características, de forma que possam ser compostas de acordo com as necessidades de cada cliente. Para a linguagem de programação Groovy, não há ferramentas e *frameworks* que apoiam a implementação de uma LPS utilizando POC. Groovy é uma linguagem de programação que vem crescendo em popularidade nos últimos anos. Dado este cenário, neste trabalho é proposto e desenvolvido o `G4FOP`, uma extensão de um *framework* chamado `Feature House` para a linguagem de programação Groovy. `Feature House` é um *framework* para composição de software suportado por um conjunto de ferramentas. Além de apresentar `G4FOP`, este trabalho demonstra a utilização desta extensão através da composição de características de uma LPS de exemplo. Também é apresentado como foi verificado que as estruturas da linguagem Groovy estavam sendo suportadas por `G4FOP`.

Palavras-chave: Groovy, Característica, Feature House, Linha de Produto de Software.

Abstract

Software Product Lines (SPL) are increasingly present in the software market, because they are a software development paradigm that aims to reuse in order to reduce costs and gain agility. There are several ways to implement a SPL, and Feature Oriented Programming (FOP) is one technique that aims to improve modularity and flexibility of SPL. The basic idea of FOP is to decompose software into smaller pieces, called features, so they can be composed according to the needs of each customer. For the programming language Groovy, there are no tool and framework that supports the implementation of a SPL using FOP. Groovy is a progrraming language that has been growing in popularity in recent years. Given this scenario, this work proposes G4FOP, which is an extension of a framework called `Feature House` for the Groovy programming language. Feature House is a framework for software composition supported by a set of tools. In addition to present the extension to `Feature House`, this work demonstrates the use of this extension for composing features of a SPL sample. It is also presented how the structures of Groovy have been supported by `G4FOP`.

Keywords: Groovy, Feature, Feature House, Software Product Line.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Linha de Produtos de Software (LPS) é um conjunto de sistemas de software que compartilham funcionalidades em comum que satisfazem um segmento de mercado [Clements & Northrop, 2002]. Ao invés de desenvolver o produto do zero, LPS propõe que sejam desenvolvidas somente as suas particularidades. Dessa forma LPS provê economia de escopo, uma vez que a maioria dos produtos são similares em uma família de sistemas [Clements & Northrop, 2002]. Existem diversas técnicas e paradigmas que suportam a implementação de LPS. Uma delas é a chamada Programação Orientada a Características [Batory et al., 2004].

Programação Orientada a Características (POC) ou Desenvolvimento de Software Orientado a Características é um paradigma para construção e customização de sistemas de software [Apel et al., 2008]. O principal conceito deste paradigma é uma característica, que representa uma unidade de funcionalidade de um sistema que satisfaz um requisito e provê uma opção de configuração em potencial [Batory et al., 2004]. A ideia básica do POC é decompor o software em partes menores, as características, que possam ser compostas de acordo com as necessidades do cliente. A partir deste conjunto de características é possível construir diversas combinações de sistemas de uma LPS. [Apel et al., 2008].

Existem várias ferramentas e abordagens para as mais diversas linguagens de programação que oferecem suporte a POC para LPS. Uma delas é o `Feature House`, que é um *framework* para composição de artefatos de software [Apel et al., 2009]. Ele é um *framework* genérico que pode ser estendido para diversas linguagens de programação, utilizando para isso o paradigma de composição chamado superposição[1]. `Feature House` possui extensões para linguagens de programação como Java e Haskell, por exemplo [Apel et al., 2009].

[1] do inglês *superimposition*

Uma linguagem de programação que vem sendo cada vez mais utilizada nos últimos anos é Groovy. Groovy é uma linguagem dinâmica, opcionalmente tipada e com tipagem estática para a plataforma Java [Apache, 2016b; Souza & Figueiredo, 2014]. Ele visa melhorar a produtividade do desenvolvedor devido a uma sintaxe concisa e familiar para os desenvolvedores Java, o que torna a sua curva de aprendizagem pequena para estes desenvolvedores. Groovy se integra facilmente com programas Java e entrega imediatamente para a aplicação poderosas funcionalidades, como recursos de *script* e programação funcional [Apache, 2016b].

1.1 Descrição do Problema e Objetivo

O objetivo deste trabalho é criar uma extensão do `Feature House` para a linguagem de programação Groovy, que será chamada de `G4FOP`[2], ou Groovy para POC. Groovy foi criada em 2003 por Guillaume Laforge e é uma linguagem opcionalmente tipada, dinâmica, que foi desenvolvida para a plataforma Java. Possui muitas características que foram inspiradas nas linguagens Python[3], Ruby[4] e Smalltalk[5]. Ela é compilada em *bytecode*, o que a faz executar naturalmente em uma Máquina Virtual Java [Koenig et al., 2007].

A linguagem Groovy vem sendo cada vez mais utilizada nos últimos anos. Segundo o ranking do TIOBE [TIOBE, 2016], entre agosto de 2015 e agosto de 2016 Groovy subiu 21 posições no ranking das linguagens de programação mais utilizadas, passando da 37ª posição para a 16ª. Apesar deste cenário de crescimento da sua popularidade, há uma certa carência de ferramentas e *frameworks* que suportem a implementação de LPS utilizando POC em Groovy. Dado este cenário, a motivação deste trabalho é disponibilizar a abordagem de composição de software utilizada pelo `Feature House` para a linguagem de programação Groovy, de forma a fomentar e facilitar a adoção de linhas de produtos de software nesta tecnologia. Esta abordagem agiliza a criação de um novo produto, além de fazer com que não seja necessário fazer alterações no código a cada nova configuração de produto.

Há muitos trabalhos relacionados a este, pois de forma geral todo trabalho que apresenta uma forma de compor características de uma LPS utilizando POC como base pode ser considerado um trabalho relacionado. Os principais trabalhos relacionados a este são os decorrentes da criação do `Feature House`, como o trabalho de Apel &

[2]sigla para a expressão Groovy *for Feature Oriented Programming*
[3]www.python.org
[4]www.ruby-lang.org
[5]www.smalltalk.org

Lengauer [2008]. Nesse trabalho foi apresentada a abordagem de composição chamada superposição, que é utilizada pelo *framework* para compor as características. O outro principal trabalho relacionado é o de Apel et al. [2009], no qual o `Feature House` foi de fato apresentado. Apel et al. [2009] mostra a ideia por trás do *framework*, seu funcionamento e quais linguagens de programação estão integradas, que são Java, C, C#, Haskell, Alloy, JavaCC, XHTML, XMI/UML e Ant. Outros trabalhos relacionados são o de von Rhein [2008] e Dörre & Lengauer [2009] nos quais são apresentadas como compor características utilizando a superposição nas linguagens C# e XML respectivamente. Também pode-se citar o trabalho de Batory et al. [2004] no qual foi a apresentada a abordagem AHEAD, que é baseada no conceito de refinamentos sucessivos. Outros dois trabalhos relacionados são o rbFeatures [Günther & Sunkle, 2009], no qual é apresentado uma linguagem de domínio específica para POC, e o Feature C++ [Apel et al., 2005] no qual é apresentada uma extensão da linguagem C++ para dar suporte a POC.

1.2 Esboço da Solução

A solução proposta consiste na criação do `G4FOP`, que é uma extensão do *framework* `Feature House` para a linguagem de programação Groovy. Com o `G4FOP` é possível compor características de uma LPS desenvolvida em Groovy utilizando POC. A criação do `G4FOP` é baseada em uma gramática da linguagem escrita em um formato chamado `FeatureBNF`. Este formato é semelhante ao BNF, porém suporta algumas extensões e anotações [Apel et al., 2009]. Esta gramática possui duas partes, a primeira na qual são informados os *tokens* da linguagem Groovy, e na segunda parte são definidas as produções das construções gramaticais da linguagem, com todas as suas estruturas, como por exemplo os laços de repetição e a estrutura de importações de bibliotecas. Nesta parte, é definida qual será a granularidade da gramática. Ou seja, quais elementos de código serão representados como nós internos (não-terminais) e quais serão representados como nós folhas (terminais). Isso define quais sofrerão composição de acordo com as regras definidas ou quais serão armazenados apenas como texto nos nós terminais.

A partir da gramática de Groovy é executada uma ferramenta que faz parte do *framework* `Feature House`, passando a gramática como parâmetro. Ela converte a gramática do formato `FeatureBNF` para o formato JavaCC puro, que é um gerador de analisador sintático [Cognisync, 2016]. Esta ferramenta que faz parte do `Feature House` é muito útil, uma vez que abstrai grande parte da complexidade de se escrever uma gra-

mática diretamente em JavaCC. Após isso, JavaCC é executado na gramática, que irá gerar os *parsers* de Groovy. Com todos os artefatos gerados, eles são então conectados e configurados em uma outra ferramenta do `Feature House`. Esta ferramenta, a partir das regras de composição definidas na gramática anteriormente escrita, estará apta a fazer a composição das características de uma LPS. A Figura 1.1 mostra a arquitetura do `Feature House`. De uma forma geral, ele recebe a gramática de uma linguagem e gera artefatos a partir dos quais a composição das características será realizada.

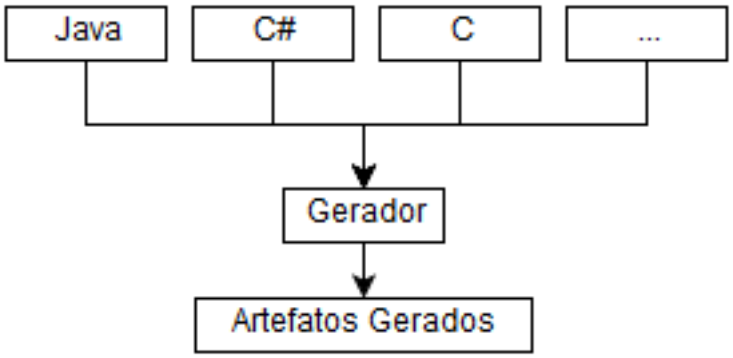

Figura 1.1. Arquitetura do Feature House.

A gramática de Groovy escrita para a criação do `G4FOP` foi baseada na gramática de Java, uma vez que estas linguagens possuem grandes semelhanças. Porém vale ressaltar que apesar de terem semelhanças, as duas também possuem diferenças, que precisaram ser retratadas na gramática escrita para Groovy. Por exemplo, Groovy é opcionalmente tipada, enquanto Java é uma linguagem tipada. Groovy possui diferentes formas de instanciar algumas estruturas, como por exemplo *arrays* e *maps*. Existem várias outras diferenças entre as linguagens, que serão abordadas no decorrer do trabalho.

1.3 Avaliação e Demonstração de Uso

A demonstração de uso foi feita utilizando uma aplicação de exemplo chamada ATM [Deitel & Deitel, 2005], que simula o funcionamento de um caixa eletrônico. Esta aplicação preenche todos os requisitos de uma LPS, pois ela pode ser modelada em torno de características, que podem ser compostas de acordo com a necessidade do cliente. Por exemplo, um caixa eletrônico pode ter somente a opção de consulta do saldo, enquanto outro pode também receber depósitos, ou fazer saques. Esta aplicação originalmente foi implementada em Java. Ela foi convertida para Groovy neste trabalho, de forma que pudesse ser composta utilizando a extensão criada.

Para garantir que as estruturas da linguagem Groovy estavam sendo suportadas por `G4FOP`, foi feita uma avaliação preliminar desta extensão. Primeiramente, foi repassada a documentação oficial da linguagem Groovy [Apache, 2016b], de forma a conferir seus operadores e estruturas. Além disso, foi utilizado um site chamado "*Learn X in Y Minutes*" [Alcolea, 2016] que possui tutoriais de diversas linguagens de programação. O tutorial de Groovy deste site é composto por um arquivo Groovy. Este arquivo foi utilizado como *benchmark* porque apresenta uma a uma todas as estruturas da linguagem. A gramática foi conferida com este arquivo, até que ele pudesse ser composto como uma característica.

1.4 Estrutura da Dissertação

Este trabalho está organizado em cinco capítulos. O Capítulo 2 detalha os conceitos teóricos necessários para o entendimento deste trabalho. Nele serão abordados temas como Linhas de Produtos de Software, Programação Orientada a Características, detalhes da ferramenta `Feature House`, características da linguagem Groovy e JavaCC.

No Capítulo 3 é explicado como foi feita a extensão do Fetaure House para Groovy, mostrando desde configurações na própria ferramenta, até a escrita de uma gramática Groovy. Esta gramática é necessária para a geração dos arquivos a serem utilizados durante a composição das características.

O Capítulo 4 demonstra o funcionamento da ferramenta, através da sua utilização em uma LPS de exemplo. Também é mostrado como as estruturas da linguagem foram testadas, de forma a mostrar que as construções da linguagem Groovy estão sendo aceitas pela gramática construída. O Capítulo 5 conclui o trabalho, fazendo algumas considerações, citando as contribuições atingidas e propondo trabalhos futuros.

Capítulo 2

Fundamentos

Linhas de Produtos de Software (LPS) estão cada vez mais presentes no mercado de software, por se tratar de um paradigma de produção de software que tem como objetivo reutilizar como forma de reduzir custos e ganhar agilidade. Há diversas formas de implementar uma LPS, sendo a Programação Orientada a Características (POC) uma técnica proposta para melhorar a modularidade e flexibilidade de uma LPS. Para a linguagem de programação Groovy, não há muitas ferramentas e *frameworks* que apoiem a implementação de uma LPS utilizando POC. Groovy vem crescendo em popularidade nos últimos anos. Dado este cenário, neste trabalho é criado o `G4FOP`, que é uma extensão de um *framework* chamado `Feature House` para a linguagem de programação Groovy. Neste capítulo serão apresentados tópicos importantes para o entendimento do trabalho.

A Seção 2.1 detalha a linguagem Groovy, suas principais características e diferenças para a linguagem Java, na qual Groovy foi inspirada. A Seção 2.2 explica o que são Linhas de Produto de Software. A Seção 2.3 explica o conceito de característica e modelo de características, enquanto a Seção 2.4 aborda Programação Orientada a Características, e alguns dos aspectos relacionados a ela que são importantes para o presente contexto. A Seção 2.5 fala mais especificamente sobre o *framework* `Feature House`, explicando seus conceitos e modo de funcionamento. Por fim a Seção 2.6 faz algumas considerações finais sobre o capítulo.

2.1 Groovy

Groovy é uma linguagem com tipagem opcional e dinâmica para a plataforma Java [Apache, 2016b]. Esta linguagem possui diversas características que foram inspiradas em outras linguagens, como Python[1], Ruby[2] e Smalltalk[3]. Desta forma, tais características a tornam disponível aos programadores Java, pois utilizam uma sintaxe muito parecida com esta. Groovy foi concebido não para substituir a linguagem Java, mas sim para complementá-la [Koenig et al., 2007]. Groovy integra sem problemas com bibliotecas existentes em Java, de forma que a curva de aprendizado seja pequena para programadores Java [Layka et al., 2013]. Groovy combina a utilidade dos recursos dinâmicos de linguagens como Ruby e Python com a estabilidade e poder de Java [Koenig et al., 2007].

Groovy foi idealizado para a Máquina Virtual Java, tanto que o código é compilado em *bytecode*, o que faz com que sua integração seja transparente com aplicações Java. Isso faz com que sua sintaxe seja parecida com Java, facilitando o aprendizado da linguagem principalmente para programadores acostumados com Java. Groovy também pode ser utilizado como linguagem de *script* [Layka et al., 2013].

Segundo o ranking do TIOBE [TIOBE, 2016], em agosto de 2015 Groovy era a 37ª linguagem de programação mais utilizada no mundo. Um ano depois, em agosto de 2016, já era a 16ª da lista. Ou seja, ela deu um salto de 21 posições em um ano. A Figura 2.1 mostra o crescimento da linguagem Groovy nos últimos 10 anos, principalmente de 2015 para 2016, saltando da 37ª para a 16ª posição das linguagens mais utilizadas pelo ranking do TIOBE. Isso pode ter ocorrido pelo fato de, neste período, a linguagem ter sido incorporada a organização Apache Software Foundation. Outro fator que pode ter contribuído para este crescimento é a popularização de frameworks Web para Groovy, sendo o principal deles o Grails [Apache, 2016a]. Segundo o ranking do site Hot Frameworks [HotFrameworks, 2016], Grails é o 24º framework de desenvolvimento mais utilizado na atualidade.

Apesar de grande parte do código Groovy ser parecido com Java, uma linguagem não é um subconjunto da outra. Por exemplo, algumas das similaridades das duas linguagens são o mecanismo de empacotamento, lançamento de exceções, instanciação de objetos e declaração e uso de genéricos e anotações. Podemos citar de uma forma geral alguns pontos como sendo os principais valores agregados por Groovy, como: (i) o fácil acesso a objetos Java através de novas expressões e operadores, (ii) o fato da linguagem

[1] www.python.org
[2] www.ruby-lang.org
[3] www.smalltalk.org

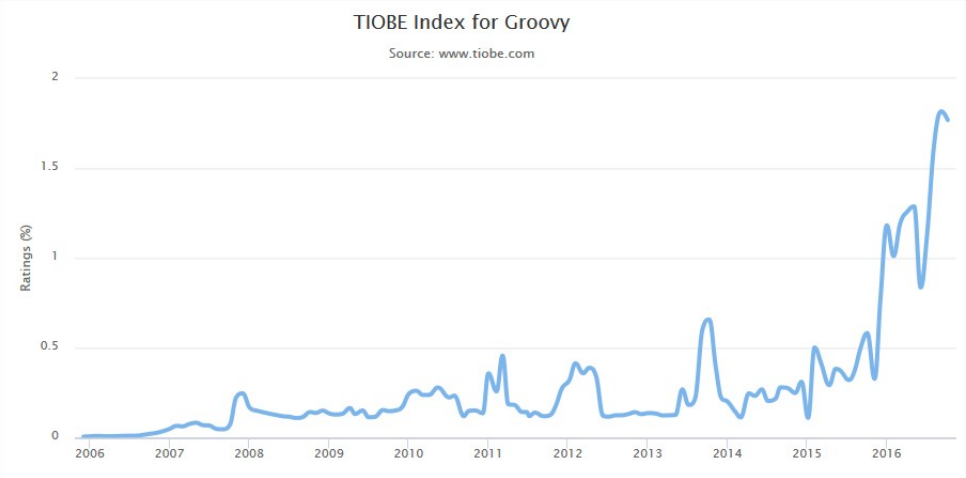

Figura 2.1. Crescimento da utilização do Groovy nos últimos anos [TIOBE, 2016].

permitir diferentes maneiras de criar objetos usando literais e (iii) proporcionar novas estruturas de controle para permitir o controle de fluxo. Também podemos citar a introdução de novos tipos de dados, juntamente com os seus operadores e expressões [Koenig et al., 2007].

Groovy importa alguns pacotes muito utilizados da linguagem Java por padrão, de forma que é possível utilizar as funcionalidades destes pacotes sem se preocupar em importá-los. Uma vez que a linguagem é dinâmica, caso haja dois métodos com o mesmo nome, porém tipos de parâmetros diferentes, Groovy irá escolher o método mais adequado em tempo de execução [Apache, 2016b]. Groovy também possui desde sua primeira versão uma funcionalidade que só foi criada em Java na versão 8, que é chamada de *closures*. *Closure* é um tipo de dados qualquer, porém ao invés de armazenar dados, armazena código executável. Isso faz com que a linguagem Groovy possa ser aplicada ao paradigma de programação funcional, pois permite que seja alcançado os três aspectos que caracterizam este paradigma: funções de primeira classe, funções puras e recursão [Weissmann, 2015]. As funções de primeira classe são passadas por parâmetro para outras funções ou são o resultado de uma função. As funções puras são aquelas que não alteram o estado do sistema. A recursão é alcançada uma vez que uma *closure* pode ser recursiva [Weissmann, 2015]. Estas adições de elementos da sintaxe fazem com que o código Groovy seja mais compacto e fácil de ler. Por exemplo, Groovy já importa alguns pacotes por padrão fazendo com que o código fique mais enxuto. Com isso, é possível utilizar os métodos que estão dentro destes pacotes sem a necessidade de referenciá-los [Koenig et al., 2007].

2.2 Linhas de Produtos de Software

Linha de Produtos de Software (LPS) pode ser definido como um paradigma para produção de uma família de sistemas de software que possuem características em comum e particularidades [Pohl et al., 2005; Do Vale & Figueiredo, 2015]. A engenharia de LPS propõe um desenvolvimento dirigido pelo reuso, no qual a ideia é focar o desenvolvimento em artefatos que possam ser reaproveitados por outro produto da família. Assim, novos produtos devem implementar somente as particularidades de cada configuração. Apesar de fornecer ganhos, são necessários investimentos iniciais por parte da empresa para manter uma LPS [Dikel et al., 1997; Pohl et al., 2005; vd Linden et al., 2007]. Ao invés de desenvolver o produto do zero, faz-se somente as suas particularidades. Dessa forma, LPS provê economia de escopo, uma vez que a maioria dos produtos são similares, uma vez que foram projetados dessa forma [Clements & Northrop, 2002]. No geral, o número de empresas que estão optando por utilizar LPS está crescendo [Pohl et al., 2005].

Na literatura, há diversas técnicas que apoiam a implementação de linhas de produtos de software, como por exemplo a Engenharia de Software Baseada em Componentes [Heineman & Councill, 2001], a Compilação Condicional [Hu et al., 2000], a Programação Orientada a Aspectos [Kiczales et al., 1997] e a utilizada neste trabalho, a Programação Orientada a Características [Figueiredo et al., 2008]. A Engenharia de Software Baseada em Componentes, que propõe que o desenvolvimento de um sistema deve ser através de componentes reutilizáveis, conectando as partes, que por sua vez devem ser mantidas e customizadas conforme necessário [Crnković, 2003]. É possível construir uma LPS utilizando este conceito em uma linguagem orientada a objetos, como Java por exemplo, mas há algumas desvantagens. Pode-se citar como desvantagens um maior esforço e tempo necessários para o desenvolvimento de componentes reusáveis, gerência não preditiva e ineficiente de requisitos, conflitos entre reusabilidade e usabilidade, custo de manutenção elevado e grande sensibilidade a mudanças [Crnković, 2003].

Uma outra maneira é com a Compilação Condicional [Gacek & Anastasopoules, 2001], que tem como característica habilitar o controle sobre segmentos do código que serão incluídos ou excluídos da compilação do programa. Estes segmentos são marcados através de diretivas sobre as porções do código onde há variabilidade. Uma grande vantagem desta técnica é o encapsulamento de múltiplas implementações em um mesmo módulo. Uma desvantagem desta técnica é que as diretivas condicionais não suportam recursão ou outro tipo de *loop*, o que faz a seleção de características através de algoritmos avançados impossível. Outra desvantagem é que como as diretivas estão

misturadas no código, não é possível ter uma visão geral das variabilidades [Gacek & Anastasopoules, 2001].

Uma outra alternativa é a Programação Orientada a Aspectos (POA) [Kiczales et al., 1997; Figueiredo et al., 2008]. POA foi proposta para modularizar interesses transversais. Seu principal mecanismo de modularização é um aspecto, que encapsula o código de um interesse, que pode estar emaranhado e disperso pelo código de outros interesses [Gaia et al., 2014]. Esta técnica define os conceitos de aspecto, interesses transversais, adendos, pontos de junção, pontos de corte e declarações intertipo. Também é mostrado como é possível resolver com este técnica, problemas de programação que técnicas de Programação Orientada a Objetos não são capazes de resolver.

2.3 Característica e Modelo de Características

Há diversas definições para característica, devido a diversidade de pesquisas na área. Podemos definir característica como sendo uma unidade de funcionalidade de um sistema que satisfaz um requisito, representando uma decisão de projeto, e provê uma potencial opção de configuração [Apel & Kästner, 2009]. É uma abstração funcional capaz de ser identificada, e que deve ser implementada, testada e mantida [Kang et al., 1998]. Também pode ser definida como uma estrutura que estende e modifica a estrutura de um programa com o objetivo de satisfazer o requisito do cliente, de forma a implementar e encapsular a decisão de projeto, e oferecer uma opção de configuração [Apel et al., 2008].

O modelo de características (ou diagrama de caraterísticas) é um conjunto hierárquico de características, no qual os relacionamentos entre uma característica pai e suas filhas (ou sub-características) podem ser categorizadas de 5 maneiras. A Figura 2.2, inspirada em Batory [2005], demonstra as notações utilizadas em um diagrama de características para representar estes relacionamentos. A primeira é o relacionamento "e", no qual todas as sub-características devem ser selecionadas. Este relacionamento não possui uma representação especial, há apenas uma junção com uma linha comum entre a característica pai e suas filhas. A segunda maneira é o relacionamento alternativo, no qual apenas uma sub-características deve ser selecionada. Este relacionamento é representado por um semicírculo não colorido. O terceiro tipo de relacionamento é o "ou", onde uma ou mais sub-característica pode ser selecionada [Batory, 2005]. Ele possui representação similar ao alternativo, porém com o semicírculo colorido. Uma característica obrigatória deve ser selecionada se a sua característica pai é também obrigatória ou se seu pai é opcional e foi selecionado. Características obrigatórias definem

pontos em comum, uma vez que elas devem ser selecionadas para todos os produtos provenientes daquela linha de produto de software [Metzger & Pohl, 2014]. Por fim, há as características opcionais, que podem ou não ser selecionadas [Batory, 2005]. Uma característica obrigatória possui um círculo preenchido, enquanto uma opcional possui um círculo não preenchido.

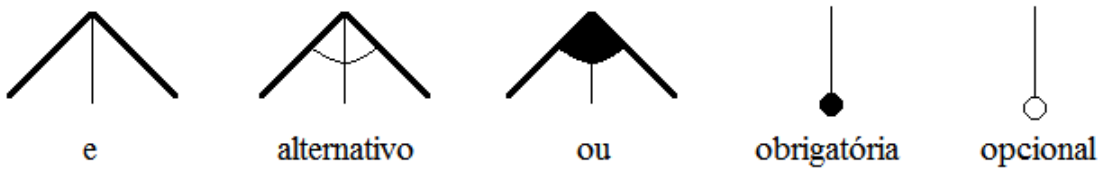

Figura 2.2. Notações do diagrama de características.

Também é possível descrever restrições em um modelo de características através de restrições dos tipos "requer" e "exclui". A primeira diz que para uma característica ser selecionada é necessário que uma outra também tenha sido selecionada. Ou seja, a sua seleção requer a escolha de uma outra. A restrição do tipo "exclui" diz que caso uma característica tenha sido escolhida, a outra citada pela restrição não pode ser selecionada. Ou seja, uma exclui a outra [Pohl et al., 2005].

A Figura 2.3 mostra um modelo de características de exemplo. Este é o modelo da LPS chamada ATM [Deitel & Deitel, 2005], que simula um caixa eletrônico bancário, e que será utilizada no Capítulo 4 para fazer a demonstração de uso da extensão para Groovy. Neste modelo estão representadas as características obrigatórias e opcionais, e um relacionamento "ou" entre duas características, no qual uma ou mais podem ser escolhidas. Também estão representadas duas restrições do tipo "requer" entre características. A primeira restrição diz que a característica "Withdrawal" só pode ser selecionada caso a característica "CashDispenser" tenha sido selecionada. De modo similar, a segunda restrição diz que a característica "Deposit" só pode ser selecionada caso a característica "DepositSlot" tenha sido selecionada.

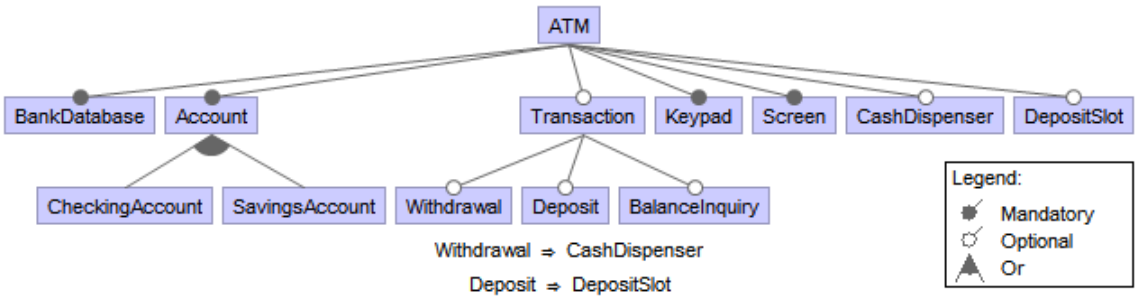

Figura 2.3. Modelo de características da LPS ATM.

2.4 Programação Orientada a Características

Programação Orientada a Características (POC) é um paradigma que favorece a aplicação sistemática do conceito de característica em todas as fases do ciclo de vida de software. Em POC, o conceito de característica é usado para analisar, projetar, implementar, customizar, depurar e evoluir o sistema [Apel & Kästner, 2009]. A Figura 2.4, inspirada em Kastner et al. [2009], demonstra as fases do processo de desenvolvimento sob o paradigma de orientação a características. Na primeira fase, o domínio é analisado para identificar suas características e relacionamentos. A segunda fase é a implementação da característica, que continua na engenharia de domínio, porém passando para o espaço da solução, e não mais do problema. Para desenvolver um produto de fato, o modelo caminha para a terceira fase, voltando para o espaço do problema, porém avançando para a engenharia de solução. Nesta fase serão analisados os requisitos do produto, e então serão selecionadas as características que farão parte do produto de forma a satisfazer os requisitos. A partir desta seleção, na última etapa, novamente no espaço da solução, as características são compostas e o produto final é obtido [Kastner et al., 2009].

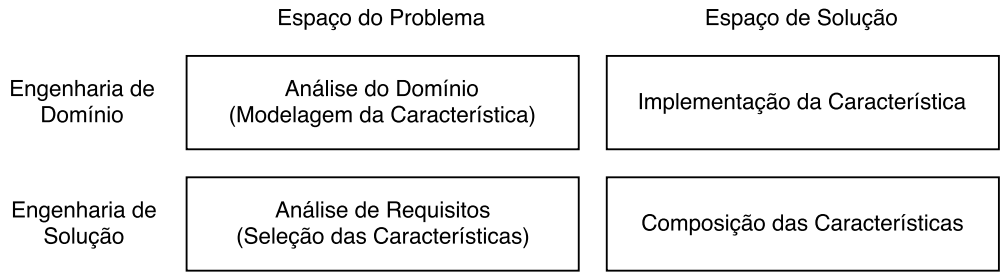

Figura 2.4. Fases do processo de POC.

Em POC pode-se desenvolver um produto baseando-se na técnica de refinamentos sucessivos [Batory et al., 2004]. Refinamentos sucessivos é uma técnica para desenvolvimento de software a partir de um sistema original incrementando funcionalidades por meio de componentes. Um componente pode ser uma constante ou pode ser um refinamento. Uma constante é um elemento básico que corresponde ao sistema original. Um refinamento é um elemento que aperfeiçoou uma constante adicionando a ela funcionalidades. Por exemplo, um refinamento pode adicionar atributos e métodos a uma constante, ou até mesmo substituir métodos existentes na constante [Batory et al., 2004].

A Figura 2.5 mostra um exemplo de código Java extraído do ATM que pode ser composto por meio do `Feature House`. Esta figura mostra um exemplo de refinamento

realizado pela característica filha chamada Withdrawal, na unidade de código ATM. A característica Withdrawal possui duas classes, ATM e Withdrawal. A classe Withdrawal não é refinamento de nenhuma outra. Ela é uma classe que só existe dentro dessa característica. Já a classe ATM da característica Withdrawal é um refinamento da classe ATM do pacote principal. No exemplo, este refinamento acrescenta ao menu do caixa eletrônico a opção de saque. O conteúdo da classe ATM da característica Withdrawal é inserido no lugar da sentença "original()". Este é um mecanismo criado pelo `Feature House` para mesclar o corpo de dois métodos que possuem o mesmo nome. Esta funcionalidade será melhor explicada no Capítulo 3.

Característica ATM

```
1 public class ATM {
2    Screen screen;
3    public Integer displayMainMenu() {
4       screen.displayMessageLine("Main menu:");
5       original();
6       screen.displayMessage("Enter a choice:");
7    }
8 }
```

Caracerística Withdrawal

```
1 public class ATM {
2    Screen screen;
3    public int displayMainMenu() {
4       screen.displayMessageLine("1 - Withdrawal");
5    }
6 }
```

Figura 2.5. Trecho de código da LPS ATM.

2.5 Feature House

`Feature House` é um *framework* para composição de software suportado por um conjunto de ferramentas [Apel et al., 2009]. Ele provê facilidades para a composição de software baseado em um modelo independente de linguagens e em um *plugin* para integração de novos artefatos de linguagem. `Feature House` utiliza uma abordagem de composição de software chamada superposição, que é o processo de compor artefatos de software mesclando as suas subestruturas correspondentes [Apel et al., 2009].

`Feature House` possui extensões para as linguagens Java, C, C#, Haskell, Alloy, JavaCC, XHTML, XMI/UML e Ant. Ou seja, `Feature House` pode ser utilizado para compor software escritos nessas linguagens. `Feature House` é composto basicamente por duas ferramentas. FSTComposer e `FSTGenerator`. FSTComposer se baseia em um modelo geral de estrutura de artefatos de software, chamado Feature Structure Tree (FST). Por exemplo, um artefato que foi escrito em Java possui pacotes, classes, métodos, entre outras estruturas, que podem ser representados em uma FST como nós. Um nó contém seu nome e o tipo da sua estrutura. Os nós internos são chamados de não terminais, enquanto os nós folhas são chamados de terminais [Apel et al., 2009].

A Figura 2.6, inspirada em Apel et al. [2009], demonstra como um trecho de código pode ser representado através de uma FST. A granularidade da linguagem define quais elementos serão terminais ou não terminais, e é feita durante a criação da extensão. Normalmente, uma granularidade como da Figura 2.6 é mais utilizada, definindo pacotes e classes como nós não terminais, e métodos e campos como nós terminais. O que é caracterizado como terminal é armazenado como texto no nó. FSTComposer espera uma lista de unidades que serão compostas, organizadas em uma estrutura de subdiretórios [Apel et al., 2009]. A Figura 2.7 representa a FST do trecho de código da Figura 2.6. A declaração "package" da Figura 2.6 é representada pelo retângulo do topo da Figura 2.7. A declaração da classe "Withdrawal" é representada por um retângulo conectado no retângulo do "package". Logo abaixo estão as declarações dos campos "amount", "keypad" e do método "displayMenuOfAmounts". Na Figura 2.7 os três estão conectados no retângulo da classe "Withdrawal".

```
1  package withdrawal;
2  public class Withdrawal extends Transaction {
3      private Integer amount;
4      private Keypad keypad;
5      public Integer displayMenuOfAmounts() {
6          Integer userChoice = 0;
7          Screen screen = getScreen();
8          while (userChoice.equals(0)) {
9              screen.displayMessageLine("1-$20");
10             screen.displayMessageLine("2-$40");
11         }
12     }
13 }
```

Figura 2.6. Trecho de código de uma Feature Structure Tree.

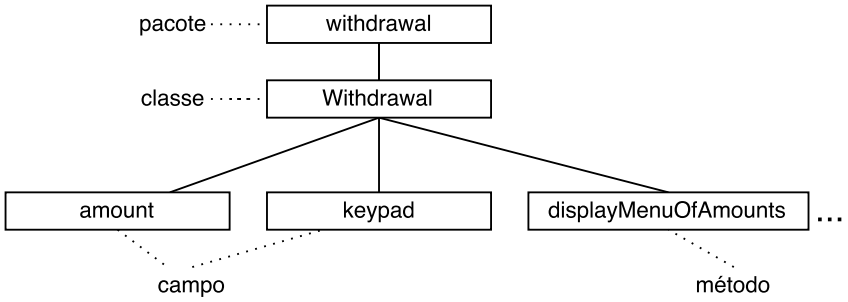

Figura 2.7. Exemplo de uma Feature Structure Tree.

A Figura 2.8, inspirada em Apel et al. [2009], mostra dois tipos de granularidade possíveis. O tipo de granularidade indica quais estruturas da linguagem serão consideradas como terminais ou não terminais. A primeira é uma granularidade grossa, pois o arquivo Groovy é considerado como não terminal, e o seu conteúdo inteiro já é considerado como terminal. O segundo tipo demonstra uma granularidade mais fina, pois dentro do arquivo ainda temos estruturas não terminais, como a classe, e o os métodos e campos ficam sendo as estruturas terminais [Apel et al., 2009].

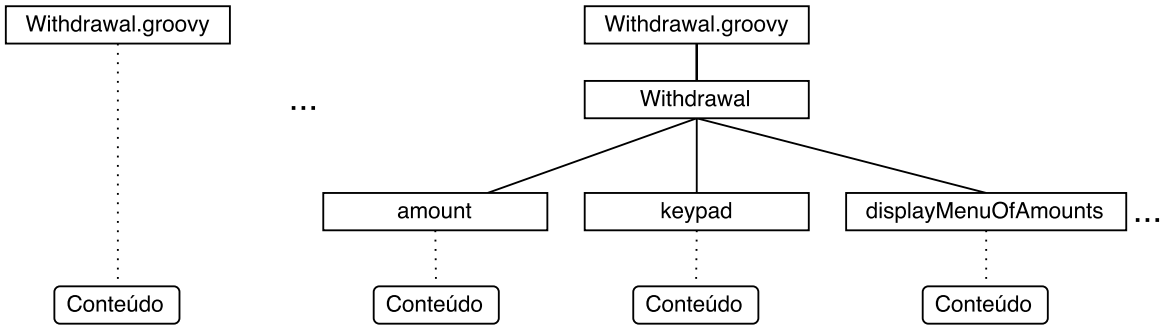

Figura 2.8. Exemplo de tipos de granularidade.

Com o `Feature House` é possível fazer a composição de software de uma LPS implementada utilizando POC. Composição de software é o processo de construir sistemas a partir de um conjunto de componentes [Apel & Lengauer, 2008]. Seu objetivo é favorecer a reutilização, personalização e manutenção de grandes sistemas. Uma popular abordagem para composição de software, que a utilizada pelo `Feature House`, é a superposição. Superposição é o processo de compor artefatos de software de diferentes componentes através da mescla de suas subestruturas correspondentes. As árvores de dois componentes são compostas recursivamente através da composição de nós do mesmo nível, contados a partir da raiz, com o mesmo nome e tipo. Com a superposição, duas árvores são compostas mesclando seus nós correspondentes, começando da raiz e progredindo recursivamente. Dois nós são compostos para formar um novo nó

quando seus pais, se existirem, tiverem sido compostos. Ou seja, quando os dois nós estão no mesmo nível e quando os dois têm o mesmo nome e tipo. O novo nó recebe o mesmo nome e tipo dos dois que foram compostos. Se dois nós são compostos, o processo continua com seus filhos. Se um nó está sozinho, ou seja, não há outro nó para que ele seja composto, ele é adicionado como filho a nova árvore que foi criada [Apel & Lengauer, 2008].

A Figura 2.9, inspirada em Apel & Lengauer [2008], mostra o processo de superposição de duas FST. Está sendo mostrado a composição de duas características da linha de produto de software ATM [Deitel & Deitel, 2005]. A classe principal chamada ATM é composta com a classe ATM da característica Deposit. O pacote "atm" é composto com o pacote "deposit" e é criado um novo pacote chamado "comp", proveniente da composição. Os nós chamados "ATM" são compostos com os nós de mesmo nome da outra árvore, e suas sub-árvores são compostas recursivamente. A Figura 2.10, também inspirada em Apel & Lengauer [2008], mostra o código resultante dessa composição.

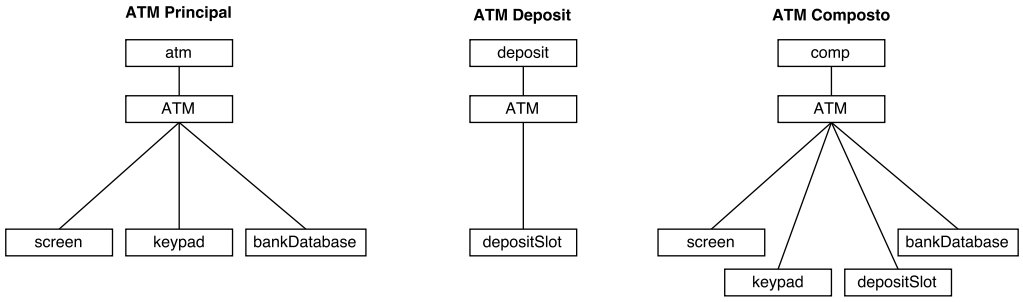

Figura 2.9. Superposição de FST.

Enquanto a composição de dois nós não terminais continua recursivamente ao longo da FST, a composição de dois nós terminais termina a recursão e necessita de um tratamento especial. Há duas opções de seguir o processo, a primeira é não permitir que dois nós terminais com mesmo nome e tipo sejam compostos. A segunda é permitir que seja feita a composição, desde que cada tipo forneça sua própria regra de composição. Há três regras básicas de composição que podem ser aplicadas por cada tipo de estrutura para compor um nó: sobrescrita, substituição e concatenação. Por exemplo, em Java dois nós terminais que sejam uma lista de importações (*imports*) são compostas concatenando suas entradas e removendo as duplicatas [Apel & Lengauer, 2008].

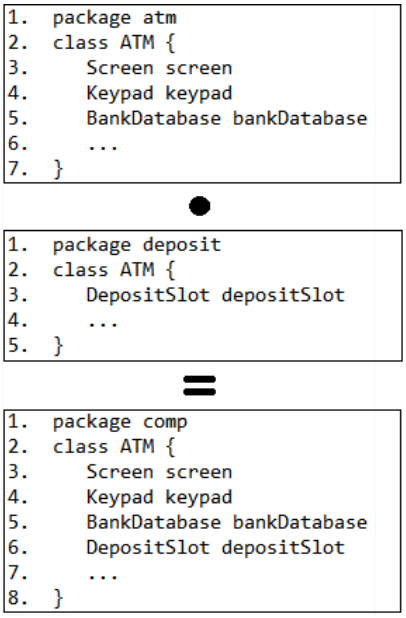

Figura 2.10. Código resultante da superposição de duas FST.

2.6 Considerações Finais

Linhas de Produto de Software (LPS) é um conjunto de sistemas software que compartilha um conjunto de funcionalidades em comum que satisfazem um segmento de mercado. Ao invés de desenvolver o produto do zero, faz-se somente as suas particularidades. Dessa forma, LPS provê economia de escopo, uma vez que a maioria dos produtos são similares em uma família de sistemas. Há diversas técnicas e metodologias para desenvolver e manter uma LPS. Uma delas é a Programação Orientada a Características (POC). POC se baseia no conceito de característica, que é uma unidade de funcionalidade de um sistema que satisfaz um requisito. POC prega que o software deve ser desenvolvido a partir de suas características, fazendo refinamentos quando preciso. Para compor um novo produto de software, deve ser observado o modelo de características da LPS. Modelo de características é um conjunto hierárquico de características, com seus relacionamentos e restrições.

Neste contexto, existe um framework chamado `Feature House`, que auxilia na composição de software, utilizando uma abordagem de composição chamada superposição. Superposição consiste em compor artefatos de software mesclando as suas subestruturas correspondentes. `Feature House` possui duas ferramentas: `FSTGenerator` e FSTComposer. `FSTGenerator` recebe uma gramática escrita no formato `FeatureBNF`, similar ao BNF com algumas particularidades. Ele gera alguns artefatos que serão utilizados pela ferramenta FSTComposer para fazer a composição do software a partir

da configuração da LPS. `Feature House` possui extensões para algumas linguagens de programação, como Java e C#.

O próximo capítulo irá mostrar a criação do `G4FOP`, que é uma extensão do *framework* `Feature House` para a linguagem de programação Groovy. Será mostrada a escrita de gramática de Groovy no formato `FeatureBNF` e a utilização da ferramenta `FSTGenerator` para converter a gramática para o formato do JavaCC. Serão apresentados os artefatos gerados pelo gerador de *parser* JavaCC e sua inclusão e configuração na ferramenta FSTComposer. Esta ferramenta é a responsável por fazer a composição das características de acordo com as regras definidas na gramática.

Capítulo 3

G4FOP: Uma Extensão do Feature House para Groovy

No capítulo anterior foram apresentados uma visão geral de conceitos ligados a Linhas de Produtos de Software (LPS) e formas de implementar e manter uma LPS. Uma das formas discutidas foi a Programação Orientada a Características (POC). Foi mostrado um *framework* chamado `Feature House`, que a partir de suas duas principais ferramentas, chamadas FSTComposer e `FSTGenerator`, apoia a composição de uma LPS utilizando POC. Também foi mostrado que este *framework* suporta algumas linguagens de programação, como Java, C# e Haskell por exemplo.

Este capítulo apresenta a criação do `G4FOP`, que se trata de uma extensão do `Feature House` para a linguagem de programação Groovy. A Seção 3.1 demonstra um passo a passo do que é necessário fazer para integrar uma nova linguagem ao `Feature House`. A Seção 3.2 discute a implementação da gramática de Groovy no formato adequado ao `Feature House`. A Seção 3.3 mostra como integrar os artefatos gerados a partir da gramática escrita e trabalhada pelo JavaCC no `FSTGenerator`, e para que serve cada um deles. A Seção 3.4 discute como utilizar os arquivos gerados pelo `FSTGenerator` para fazer a composição de um software utilizando o FSTComposer, além de mostrar o que é necessário implementar no FSTComposer para a integração do Groovy, e como ele funciona. Por fim, a Seção 3.5 discute as considerações finais, resumindo as principais ideias do capítulo.

3.1 Extensão do Feature House para Groovy

A ideia por trás do `Feature House` é de que, apesar de os artefatos de software variarem bastante de linguagem para linguagem, o processo de composição de software através da superposição é muito similar. Para qualquer nova linguagem que vá ser integrada no *framework* é preciso fornecer quatro artefatos principais. O primeiro é um *parser* e suas classes representando a árvore sintática. O segundo é adaptador que mapeia a árvore sintática para a Feature Structure Tree (FST). O terceiro é uma gramática com um conjunto de regras de composição específicas da linguagem e o quarto é um impressor para escrever as FST superpostas no disco [Apel et al., 2009]. A Seção 3.2 explica melhor esses artefatos.

A partir de uma gramática escrita no formato `FeatureBNF`, que corresponde ao terceiro artefato citado no parágrafo anterior, a ferramenta `FSTGenerator` gera os demais artefatos [Apel et al., 2009]. Com todos estes artefatos em mãos, o FSTComposer pode ler um arquivo com as características da LPS que foram selecionadas para passarem pela composição, e seguir com o processo. A Figura 3.1 mostra as fases do processo de criação do `G4FOP`, que serão explicadas nos parágrafos abaixo. Para a criação do `G4FOP`, de forma que fosse possível compor características implementadas em Groovy utilizando o `Feature House`, primeiro foi preciso escrever uma gramática da linguagem no formato `FeatureBNF`. Esta gramática teve suas estruturas anotadas com @FSTTerminal para os nós terminais e @FSTNonTerminal para estruturas que correspondem a nós não terminais.

O segundo passo foi executar a ferramenta `FSTGenerator`, que faz parte do *framework* `Feature House`, passando como parâmetro a gramática escrita no formato `FeatureBNF`. Ele irá gerar um impressor, representando pelo arquivo Java chamado "SimplePrintVisitor.java", que será o responsável por escrever os arquivos resultantes da composição em disco. O `FSTGenerator` também fará a conversão da gramática do formato `FeatureBNF` para o formato JavaCC. A gramática convertida para JavaCC é representada por um arquivo na extensão ".jj". Ao executar o JavaCC na gramática convertida (passo 3 na Figura 3.1, são gerados mais 6 arquivos Java: "TokenMsgError.java", "ParseException.java", "Token.java", "CharStream.java", "GroovyParser.java", "GroovyParserTokenManager.java" e "GroovyParserConstants.java". Estes arquivos devem ser integrados no FSTComposer, de forma que ele possa utilizá-los no momento da composição.

O quarto passo (Figura 3.1) é utilizar o FSTComposer para fazer a composição. Para isso, deve-se fornecer a ele o caminho onde está localizado o código da LPS. As características selecionadas devem estar em um arquivo com extensão ".features", das

características. A seleção das características que serão compostas podem ser feitas manualmente verificando as restrições entre elas, ou pode se usar a ferramenta `FeatureIDE` [Kastner et al., 2009], que possui integração com `Feature House`, para fazer a seleção das características. `FeatureIDE` é um ambiente de desenvolvimento baseado no Eclipse que suporta todas as fases de POC no desenvolvimento de LPS [Kastner et al., 2009]. Para utilizar o `FeatureIDE`, é preciso configurar o modelo de características da LPS na ferramenta. Dessa forma o `FeatureIDE` não irá permitir que seja selecionada uma configuração inválida. A Seção 3.4 mostra melhor estes pontos.

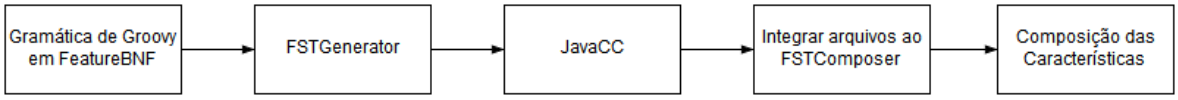

Figura 3.1. Fases do processo de criação do G4FOP.

3.2 Criando a Gramática

A criação de uma gramática para ser integrada ao `Feature House` deve ser feita utilizando o formato `FeatureBNF` [Kästner et al., 2008]. Podemos dizer que `FeatureBNF` é uma extensão da notação BNF, que foi introduzida por John Backus e Peter Naur na década de 60 como uma notação formal para descrever a sintaxe de uma linguagem [Marcotty & Ledgard, 2012]. `FeatureBNF` adicionou alguns artefatos ao BNF, de forma que o programador possa anotar a gramática da linguagem com atributos. A Figura 3.2 mostra um exemplo do uso da anotação na gramática. As duas principais formas de fazer anotações na gramática com `Feature House` são colocar um @FSTTerminal ou um @FSTNonTerminal. O primeiro define que aquele nó é terminal e o segundo define que o nó não é terminal. Por padrão, todas as regras de produção que não receberem anotações serão definidas como terminais. Sendo assim, a vantagem de utilizar o @FSTTerminal é poder especificar o nome que irá aparecer no nó da FST e na composição de dois nós terminais correspondentes [Apel et al., 2009].

```
1 @FSTTerminal
2 ImportDeclaration:
3    "import" [ "static" <NONE> ] Name [ImportPackage] @! [<SEMICOLON>]
4 ;
```

Figura 3.2. Trecho da gramática Groovy no formato FeatureBNF.

Grande parte da gramática que deve ser escrita para integrar uma linguagem ao `Feature House` é feita utilizando o JavaCC [Cognisync, 2016]. JavaCC é a abreviação para Java Compiler Compiler e é um gerador de *parser* e de analisador léxico. Ele faz a leitura da descrição de uma linguagem e gera o código, escrito em Java, que vai ler e analisar aquela linguagem.

Uma gramática escrita em `FeatureBNF` consiste em duas partes. A primeira contém instruções para o JavaCC, especialmente para a análise léxica, uma vez que nesta parte são declarados os *tokens* da linguagem. A segunda parte é separada da primeira pela palavra "GRAMMARSTART", e consiste nas construções sintáticas da linguagem [Kästner et al., 2008]. A gramática escrita para o Groovy foi baseada na gramática do Java, uma vez que as linguagens são parecidas. Porém, diversas construções e alterações tiveram de ser feitas para que a gramática se adaptasse ao Groovy.

A primeira parte da gramática é destinada aos *tokens* da linguagem, e é a partir dela que será gerado o *token manager* da gramática. O *token manager* é um analisador léxico. Ou seja, ele analisa a entrada de caracteres, quebra ela em partes chamadas *tokens*, e define cada um como um tipo de *token* [Norvell, 2007]. A segunda parte da gramática consiste em produções no formato BNF, mais as anotações do `Feature House`. As produções seguem o seguinte padrão: "NomeDaProdução : Tokens :: NomeDaEscolha | Tokens :: NomeDaEscolha | ... ;". Primeiro vem o nome da produção, depois múltiplas escolhas, onde cada uma representa uma sequência de tokens, e pode ou não ser nomeada. Os *tokens* informados podem ser não terminais, de forma a referenciarem outras produções, ou podem ser *tokens* léxicos [Apel et al., 2009]. A Figura 3.3 mostra a produção da estrutura *closure* da gramática de Groovy, como exemplo. Na Figura há a criação da produção chamada "ClosureExpression". Dentro dela são chamada mais duas produções, "ClosureParameterInitializer" e "Expression". "ClosureParameterInitializer" representa a inicialização de uma *closure*, que é uma lista de parâmetros separados por vírgula. "<IDENTIFIER>" representa um identificador qualquer, utilizado para dar nome a um atributo. "<COMMA>" representa uma vírgula e "<ARROW>" uma seta (->). Estes símbolos são definidos na primeira parte da gramática. O que está dentro de colchetes é interpretado como opcional. O que está entre parênteses com um asterisco no final possibilita que este conteúdo se repita várias vezes.

```
1 ClosureExpression:
2    "{" [<IDENTIFIER>] (ClosureParameterInitializer)*
3         [<ARROW>] (Expression)* "}";
4 ClosureParameterInitializer:
5    <COMMA> <IDENTIFIER>;
```

Figura 3.3. Produção da estrutura *closure*.

Para cada produção deve-se informar ao `Feature House`, através de anotações, se um nó é terminal ou não terminal. Se for terminal, basta colocar um @FSTTerminal na linha acima da produção. Caso seja não terminal, basta colocar @FSTNonTerminal. Caso uma produção não sofra anotação, ela será considerada como terminal. `Feature House` também fornece mais duas anotações, uma chamada FSTExportName e a outra FSTInline. FSTExportName serve para definir o nome de exportação do nó. FSTInline serve para colocar a produção na mesma linha da produção que a chama, servindo assim apenas para facilitar a leitura e escrita da gramática [Apel et al., 2009]. Um importante mecanismo utilizado na escrita da gramática foi o chamado *lookahead*, que é um mecanismo que permite informar quantos tokens devem ser analisados antes de escolher um caminho, sendo importante para evitar recursões infinitas [Norvell, 2007].

Para a escrita da gramática Groovy, foi utilizada como base a gramática da linguagem Java, que já estava integrada no `Feature House`. Porém foram necessárias diversas adaptações na mesma, visto que a linguagem Groovy traz algumas diferenças para Java. Foi necessário adicionar algumas palavras chave na gramática. Por exemplo para “as” e “in” foi necessário retirar a obrigatoriedade do ponto e vírgula no fim de cada comando, uma vez que em Groovy isto é opcional. Também foi preciso alterar para opcional a tipagem de variáveis, uma vez que em Groovy é possível declarar uma variável utilizando a palavra “def” sem definir seu tipo em tempo de compilação.

Também foram criadas novas formas de declarar algumas estruturas da linguagem, como *array*, *map* e *for*. Por exemplo, em Groovy um *array* de 0 a 20 pode ser definido com o seguinte comando: (0..20).toArray(). Foi preciso criar novas estruturas, como por exemplo a chamada *closure*, que é um bloco de código que pode receber parâmetros, retornar um valor e pode ser associado a uma variável [Apache, 2016b]. Também foi necessário fazer com que a gramática reconhecesse a estrutura chamada *expando*, que é uma meta-classe capaz de expandir seu estado e comportamento [Koenig et al., 2007]. Foi preciso fazer com que a gramática suportasse novos operadores que estão presentes em Groovy, como os chamados *spread* e *safe*, por exemplo. A Figura

3.4 exemplifica o operador *spread*. Ele executa uma ação em todos os componentes de uma coleção. Neste exemplo, todas as palavras da lista seriam convertidas para letra maiúscula. A Figura 3.5 mostra um exemplo de utilização do operador *safe*. Ele é utilizado para evitar exceções do tipo *NullPointerException*, pois ele verifica se o objeto é nulo antes de tentar resgatar o atributo solicitado. A gramática de Groovy está no Apêndice A deste trabalho.

```
1 def palavras = ['Caracteristica','Software','Gramatica']
2 palavras*.toUpperCase()
```

Figura 3.4. Operador *spread* da linguagem Groovy.

```
1 def user = User.get(1)
2 def username = user?.username
```

Figura 3.5. Operador *safe* da linguagem Groovy.

3.3 Geração dos *Parsers*

A ideia por trás do `FSTGenerator` é abstrair a parte mais complexa da escrita de uma gramática em JavaCC, que é representada por um arquivo com extensão ".jj", através da escrita no formato `FeatureBNF`. Com a gramática de Groovy escrita no formato `FeatureBNF`, o `FSTGenerator` gera a gramática no formato que o JavaCC conseguirá ler, e mais um impressor. O impressor é representado pelo arquivo SimplePrintVisitor.java, que irá escrever os nós que sofrerem a superposição no disco. O `FSTGenerator` é chamado utilizando o seguinte comando: "java -jar fstgen.jar groovy_fst.gcide groovy.jj", aonde fstgen.jar é o `FSTGenerator` no formato executável Java, groovy_fst.gcide é a gramática escrita no formato `FeatureBNF` e groovy.jj é o nome do arquivo que será gerado.

A Figura 3.6 mostra um trecho do código da estrutura *closure* de Groovy que foi gerado pelo `FSTGenerator` a partir da produção escrita no formato `FeatureBNF`, que foi exemplificado pela Figura 3.3. Pode-se perceber como é grande a complexidade abstraída pelo *framework*, uma vez que é muito mais fácil escrever produções no formato `FeatureBNF` e passar para a ferramenta convertê-lo no formato JavaCC, do que escrever nesse formato diretamente.

```
1  FSTInfo ClosureExpression(boolean inTerminal) : {
2     Token first=null,t;FSTInfo n;
3  }
4  { {first=getToken(1); productionStart(inTerminal); } (
5          "{" [<IDENTIFIER>] (n=ClosureParameterInitializer(true){
6             replaceName(n);}) *
7          [<ARROW>] (n=Expression(true){ replaceName(n);})* "}" {
8             return productionEndTerminal( "ClosureExpression","-","-",
9               "Replacement","Default",first ,token);
10         }
11 )  }
12
13 FSTInfo ClosureParameterInitializer(boolean inTerminal) : {
14    Token first=null,t;FSTInfo n;
15 } { {first=getToken(1); productionStart(inTerminal); } (
16    <COMMA> <IDENTIFIER> {
17       return productionEndTerminal(
18         "ClosureParameterInitializer","-","-",
19         "Replacement","Default",first ,token);
20    }
21 )}
```

Figura 3.6. Produção da estrutura *closure* em JavaCC.

Com o arquivo "groovy.jj" pronto, basta executar o JavaCC passando este arquivo como parâmetro, através do seguinte comando: "javacc.bat groovy.jj". Serão gerados o analisador léxico e o *parser*, através de seis arquivos: "TokenMsgError.java", "ParseException.java", "Token.java", "CharStream.java", "GroovyParser.java", "GroovyParserTokenManager.java" e "GroovyParserConstants.java". O "TokenMsgError" é uma classe simples de erro, que é usada para erros detectados pelo analisador léxico. O "ParseException" é outra classe de erro, que é utilizada para erros detectados pelo *parser*. "Token" é uma classe que representa os *tokens*, na qual cada objeto possui como atributos seu código de identificação e a sequência de caracteres que representa este *token*. A classe "GroovyParserConstants" é uma interface que define os *tokens* utilizados pela linguagem, fornecendo estes dados para o analisador léxico e para o *parser*. A classe "CharStream" é uma classe que entrega caracteres para o analisador léxico, que por sua vez é representado pela classe GroovyParserTokenManager. O *parser* é representado pela classe "GroovyParser" [Norvell, 2007].

Para que o FSTComposer possa utilizar os arquivos que foram gerados foi preciso criar uma pasta chamada "groovy" dentro do projeto do `FSTGenerator`. Dentro desta

pasta deve-se colocar as seguintes classes: “GroovyParser”, “GroovyParserConstantes”, “GroovyParserTokenManager”, “SimplePrintVisitor” e o arquivo “groovy.jj” gerado. As demais classes, “Token”, “CharStream”, “TokenMsgError” e “ParseException” não precisam ser copiadas para dentro do projeto uma vez que são classes genéricas, que independem da linguagem que está sendo trabalhada. Desta forma, elas já estão configuradas dentro do `Feature House`. Feito isso, o `FSTGenerator` está configurado para a linguagem Groovy, e basta agora configurar o FSTComposer para acessar estes artefatos quando identificar que os arquivos que vão ser compostos possuem a extensão do Groovy, e então prosseguir com a composição.

3.4 Composição das Características

FSTComposer é a ferramenta do *framework* `Feature House` responsável por fazer a composição das características selecionadas de uma LPS utilizando a abordagem de composição chamada superposição, que já teve seu funcionamento explicado na Seção 2.5 do Capítulo 2. Esta ferramenta possui uma funcionalidade chamada “original”. Ela é responsável por mesclar o corpo de dois métodos de mesmo nome e de características diferentes. Sendo assim, não é permitido que a palavra “original” seja utilizada para declarar atributos ou métodos, de forma a não gerar conflitos. No momento da composição de duas características, caso as duas tenham um método com o mesmo nome, o FSTComposer irá procurar no corpo delas a palavra original. Caso encontre, esta palavra será substituída pelo corpo do outro método. Caso não encontre, a composição se dará através da substituição de um método pelo outro [Apel et al., 2009].

Esta forma de mesclar dois corpos de método é uma das regras de composição do `Feature House`, e é chamada de sobrescrita de método. Há mais seis regras de composição utilizadas, “concatenação de construtor”, “especialização de campo”, “união de lista de implements”, “especialização de modificador”, “substituição” e “concatenação de conteúdo”. A regra chamada “concatenação de construtor” adiciona as declarações de um construtor às declarações do outro. Por exemplo, caso dois métodos construtores estejam sendo compostos, suas declarações serão concatenadas formando apenas uma. A “especialização de campo” atribui um valor inicial a um campo se ele não tivesse outro antes. A regra “união de lista de implements” une a lista de interfaces que as classes implementam, excluindo as duplicatas. A “especialização de modificador” especializa os modificadores. Por exemplo, caso dois atributos de mesmo nome estejam sendo compostos, um do tipo “Integer” e o outro do tipo “int”, o resultado da composição será do tipo “int”, já que ele é o tipo mais específico. A regra chamada “substituição” faz a

substituição de um nó terminal pelo outro. Por último a "concatenação de conteúdo" concatena o conteúdo representado por texto de dois nós terminais [Apel et al., 2009]. É o próprio FSTComposer que no momento da composição analisa caso a caso e verifica qual a melhor regra para ser aplicada.

A cada composição que o FSTComposer faz, ele imprime no console a regra que foi aplicada. Para chamar o FSTComposer, é preciso passar como parâmetro um arquivo com extensão .features, no qual deve estar presente as características que farão parte daquela composição. Este arquivo consiste no nome das características uma embaixo da outra. Caso elas tenham sido selecionadas manualmente, é importante respeitar as hierarquias e restrições de acordo com o modelo de características da LPS. Uma ferramenta que auxilia nessa seleção é a chamada `FeatureIDE` [Kastner et al., 2009], que possui integração com o `Feature House`.

3.5 Considerações Finais

`Feature House` é um *framework* para composição de software que utiliza o paradigma de composição chamado superposição. Sua principal ferramenta que faz a composição de software é chamado FSTComposer. Este *framework* possui integrações com algumas linguagens de programação, como Java, C e Haskell, por exemplo. Para estendê-lo a demais linguagens, é preciso escrever uma gramática no formato `FeatureBNF`, que será lida por uma ferramenta pertencente a este *framework*, chamada `FSTGenerator`. Esta ferramenta irá gerar a gramática no formato que o gerador de *parser* chamado JavaCC compreende, de forma a gerar os artefatos necessários para que o FSTComposer funcione. Neste capítulo foi mostrado a criação de uma extensão do `Feature House` para a linguagem Groovy, mostrando também as alterações que foram realizadas na gramática base utilizada, Java. Esta gramática foi escolhida pela similaridade existente entre ambas.

O próximo capítulo faz uma demonstração de uso da extensão criada, através da sua utilização na composição de uma linha de produto de software de exemplo. Também será mostrado como foram avaliadas as construções gramaticais da linguagem Groovy no momento da escrita da gramática. Para isso foi utilizado um site que ensina a programar em Groovy através de um arquivo com exemplos das estruturas da linguagem. Posteriormente, foi repassada a documentação oficial da linguagem, de forma a verificar as demais estruturas.

Capítulo 4

Avaliação

No capítulo anterior foi apresentada a criação do `G4FOP`, uma extensão do `Feature House` para uma linguagem de programação Groovy. Foi mostrado como a gramática foi descrica, como utilizar o JavaCC e o `FSTGenerator` para gerar os artefatos necessários e como fazer a composição utilizando o FSTComposer. Este capítulo apresenta como utilizar a extensão criada na prática. A Seção 4.1 faz uma demonstração de uso por meio da modelagem de uma Linha de Produto de Software (LPS) simples. Esta seção também mostra a implementação da LPS em Groovy respeitando as características modeladas e refinadas e fazendo a composição desta LPS utilizando a extensão criada. A Seção 4.2 mostra como foram avaliadas as construções gramaticais da linguagem Groovy. Por fim, a Seção 4.3 discute as considerações finais, resumindo as principais ideias do capítulo.

4.1 Demonstração de Uso

Para demonstrar a utilização do `G4FOP`, foi modelada e posteriormente implementada em Groovy o projeto ATM [Deitel & Deitel, 2005] no formato de uma LPS. Este projeto trata-se da implementação de um caixa eletrônico comum. Este caixa possui um teclado para que o usuário possa digitar as informações e uma tela para que ele possa ver as informações. O caixa também possui um dispositivo no qual o usuário pode retirar o dinheiro proveniente de um saque, e um dispositivo de depósito no qual o usuário pode inserir o dinheiro a ser creditado na conta informada. Além disso, o ATM possui acesso a conta dos usuários, com seu saldo, e possibilita que eles possam fazer transações bancárias, como saque, depósito ou consulta ao saldo. A conta de um usuário pode ser do tipo corrente ou poupança. Este projeto pode ser modelado como uma LPS, a qual pode ser montado cada produto do ATM de acordo com a necessidade.

Por exemplo, um caixa eletrônico que fica dentro de um supermercado geralmente não recebe depósito. Dessa forma, bastaria não selecionar a característica relacionada com depósito no momento da composição do produto para este equipamento. Em alguns banco há caixas que não fazem saque, somente permitem fazer depósitos e consultar saldo. Dessa forma, bastaria também selecionar as características necessárias no momento da composição do produto.

Cada funcionalidade foi extraída para uma característica separada, e organizada de acordo com sua estrutura hierárquica e restrições. A Figura 4.1 mostra o modelo de características da LPS implementada. As características "BankDatabase", "Account", "Keypad" e "Screen" são obrigatórias, uma vez que todo caixa eletrônico deve ter estas características. A característica "Account" possui duas filhas, "CheckingAccount" e "SavingsAccount". A primeira representa uma conta corrente e a segunda uma conta poupança. Elas estão em um relacionamento do tipo "ou", no qual uma das duas deve ser selecionada. Todo caixa deve ter uma tela para o usuário visualizar as informações, que é representada pela características "Screen". Todo caixa deve ter um teclado para que ele possa entrar com os dados necessários para a ação a qual ele deseja executar, que é representado pela características "Keypad". Além disso, todo equipamento deve possuir acesso a conta dos usuários para poder fazer as operações solicitadas, que é representado pela características "BankDatabase". As características "CashDispenser", "DepositSlot" e "Transaction" são características opcionais. Desta forma, pode existir um ATM que não faz depósito e assim não faz sentido ter o dispositivo de inserir o dinheiro. De forma similar, pode existir um caixa que não faz saque, então não é necessário a presença de um dispositivo que permite o usuário retirar o dinheiro. A característica "Transaction" possui três filhas: "Withdrawal", "Deposit" e "BalanqueInquiry". Todas elas são opcionais. Ou seja, uma ou mais de uma característica pode ser selecionada, pois cada uma das três operações podem existir separadamente uma da outra.

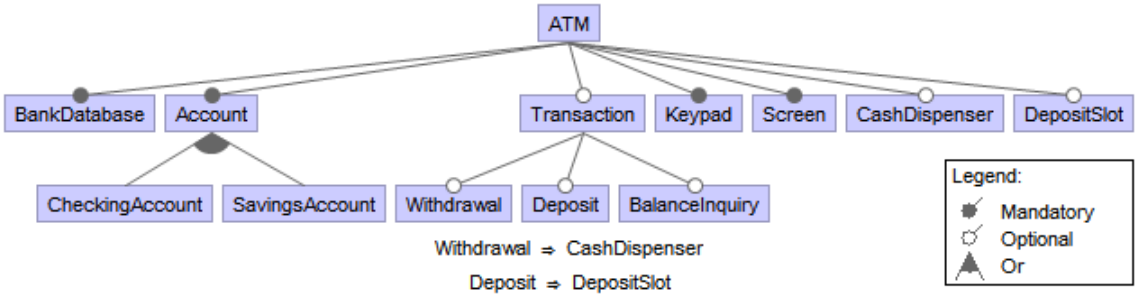

Figura 4.1. Modelo de características da LPS ATM.

Neste modelo há duas restrições. A primeira expressa a necessidade da existência do dispositivo de retirar dinheiro para que seja realizado um saque. Dessa forma a característica "Withdrawal" só pode ser selecionada se a "CashDispenser" tiver sido selecionada. Ou seja, só é possível sacar o dinheiro se o ATM tiver por onde dispensá-lo. A outra restrição é que a característica "Deposit" só pode ser selecionada caso a "DepositSlot" esteja selecionada. Ou seja, só é possível depositar dinheiro se tiver onde colocar o dinheiro no caixa.

A Figura 4.2 mostra um exemplo de parte da implementação do ATM em Groovy. Primeiro é apresentada a característica abstrata "Transaction", que possui o método abstrato chamado "execute". Este método deve ser implementado pela características que estenderem a "Transaction". Logo depois da implementação da "Transaction", é apresentado um trecho da implementação da característica "Deposit". Esta característica estende a "Transaction" e, como consequência implementa o método "execute", sendo o responsável porreceber o envelope depositado pelo usuário do ATM.

Cada uma das características foi organizada separadamente em pacotes de mesmo nome. Desta forma o `Feature House` pode reconhecê-las no momento da composição. Após a seleção das características no arquivo com extensão ".features", por exemplo "comp.features", a ferramenta de composição do `Feature House` é executada. Ao notar que os arquivos compostos são de extensão Groovy, utiliza-se então a extensão criada. Por fim, gera-se então um diretório chamado "comp", e nele é incluído todas as classes provenientes da composição.

Característica Transaction

```
1  public abstract class Transaction {
2     private int accountNumber
3     private Screen screen
4     private BankDatabase bankDatabase
5     abstract public void execute();
6  }
```

Característica Deposit

```
1  public class Deposit extends Transaction {
2     private double amount
3     private Keypad keypad
4     private DepositSlot depositSlot
5     @Override
6     public void execute() {
7        BankDatabase bankDatabase = getBankDatabase()
8        Screen screen = getScreen()
9        amount = promptForDepositAmount();
10       if (amount != 0) {
11          screen.displayMessage(
12             "\nPlease insert a deposit envelope containing " )
13          screen.displayDollarAmount(amount)
14          screen.displayMessageLine(".")
15          boolean envelopeReceived = depositSlot.isEnvelopeReceived()
16          if (envelopeReceived) {
17             screen.displayMessageLine(
18                "\nYour envelope has been received")
19             bankDatabase.credit( getAccountNumber(), amount )
20          }
21          else {
22             screen.displayMessageLine(
23                "\nYou did not insert an envelope, " +
24                "so the ATM has canceled your transaction." )
25          }
26       }
27       else{
28          screen.displayMessageLine( "\nCanceling transaction..." )
29       }
30    }
31 }
```

Figura 4.2. Trecho da característica Deposit e Transaction em Groovy.

4.2 Avaliação das Construções Gramaticais

Para a avaliação das construções gramaticais foi utilizado o site "*Learn X in Y Minutes*" [Alcolea, 2016]. Este website reúne um conjunto de tutoriais rápidos sobre diferentes linguagens de programação. Ele disponibiliza exemplos de estruturas de linguagens de programação em um único arquivo. Entre as linguagens abordadas pelo site, está Groovy. Em um só arquivo são demonstradas as principais estruturas da linguagem, como *closures* e *spread*, por exemplo. A Figura 4.3 mostra um trecho do arquivo deste tutorial, no qual é mostrado como declarar e utilizar uma *closure*. Na linha 1 é declarada a *closure* que imprime na tela a frase *"Hello World!"*. A linha 4 executa a *closure* que foi criada na linha 1. Na linha 7 é criada uma *closure* chamada "*sum*". Ela recebe dois parâmetros e imprime o resultado da soma entre eles. A linha 8 executa a closure criada na linha anterior.

```
1 def clos = { println "Hello World!" }
2
3 println "Executing the Closure:"
4 clos()
5
6 //Passing parameters to a closure
7 def sum = { a, b -> println a+b }
8 sum(2, 4)
```

Figura 4.3. Trecho do código do tutorial de Groovy "*Learn X in Y Minutes*" [Alcolea, 2016].

A avaliação da gramática de Groovy escrita foi feita utilizando este arquivo como base. Ele está no Apêndice B desta dissertação. A gramática deveria permitir que este arquivo pudesse ser processado utilizando a extensão criada. Após a avaliação preliminar da gramática utilizando este arquivo como auxílio, a documentação da linguagem Groovy [Apache, 2016b] foi repassada de forma a garantir que todas as construções e estruturas de Groovy foram cobertas pela linguagem.

4.3 Restrições de G4FOP e Ameaças a Validade

Duas restrições tiveram de ser impostas na gramática. A primeira diz respeito a ser obrigatório que todo código esteja dentro das chaves de uma "*class*". Ou seja, todo código tem de pertencer a uma classe. Groovy não tem essa restrição de origem, pois ele pode ser utilizado como linguagem de *script*. *Scripts* não obrigam que todos os

comando estejam em uma classe. A segunda restrição é de que todo o código dentro da "*class*", exceto declaração de variáveis, devem obrigatoriamente estar dentro de um método. Em Groovy isto também não é obrigatório devido a possibilidade de se escrever scripts com a linguagem. Porém, para efetuar a composição utilizando a extensão criada isso não é possível de ser utilizado. Estas restrições não trazem muito impacto para a utilização do `G4FOP`, já que na implementação de LPS geralmente utiliza-se classes para representar as unidades de códigos. Além destas, outra restrição é que pode ser que nem todas as estruturas da linguagem Groovy estão cobertas pela gramática escrita.

Há duas ameaças a validade desta dissertação. A primeira é que a LPS utilizada para demonstração de uso, o ATM, pode não refletir uma LPS real, devido ao seu tamanho reduzido. Acredita-se que, apesar de ser uma LPS simples, ela sirva para demonstrar o funcionamento do `G4FOP`, pois seu modelo de características ela possui todos os tipos de variabilidades. A segunda ameaça a validade é a possibilidade de o arquivo utilizado para avaliação das construções gramaticais não cobrir todas as estruturas da linguagem Groovy.

4.4 Considerações Finais

Este capítulo mostrou como é possível fazer a composição de uma LPS a partir da extensão para Groovy do *framework* `Feature House`. É importante modelar a LPS de acordo com seu modelo de características, respeitando as restrições e hierarquias do modelo. As características devem ser implementadas em arquivos separadamente, e organizadas em diretórios com o mesmo nome da característica. Características que fazem um refinamento de uma outra também devem estar incluídas nestes diretórios. O resultado da composição é gerada em um diretório de mesmo nome do arquivo de extensão .features, que recebe as características selecionadas para composição. Também foi mostrado neste capítulo o critério utilizado para validar se as estruturas da linguagem Groovy foram suportadas pela gramática escrita.

O próximo capítulo conclui este trabalho citando os trabalhos relacionados a implementação de LPS utilizando POC. Também é feita as considerações finais de forma a resumir as principais contribuições atingidas. Por fim são apresentados os trabalhos futuros que podem ser feitos a partir deste.

Capítulo 5

Conclusões e Trabalhos Futuros

Esta dissertação propõe uma maneira de fazer composição de software de uma LPS implementada em Groovy. Inicialmente foi apresentada a linguagem Groovy, mostrando o seu crescimento e importância no cenário atual do mercado de software. Depois da apresentação da linguagem Groovy foi discutido conceitos como POC e LPS, mostrando como modelar e implementar uma LPS de forma que facilite a composição de suas características. Foi então apresentado o *framework* `Feature House`, e como seu conjunto de ferramentas pode ser utilizado para fazer a composição de software, utilizando o paradigma de composição chamado superposição. Foi proposta e desenvolvida a extensão criada do *framework* `Feature House` para a linguagem Groovy, chamada `G4FOP`.

Este capítulo irá apresentar as considerações finais acerca do trabalho elaborado. A Seção 5.1 apresenta os trabalhos relacionados com este, tanto aqueles que abordam diretamente `Feature House`, como aqueles que trabalham com outras formas de utilizar a Programação Orientada a Características (POC) para desenvolver sistemas de software. A Seção 5.2 irá apresentar as conclusões atingidas após o desenvolvimento da extensão e sua posterior avaliação. Esta seção também faz um resumo das contribuições atingidas. A Seção 5.3 irá discutir a respeito de possíveis trabalhos futuros que possam vir a surgir a partir deste.

5.1 Trabalhos Relacionados

Os principais trabalhos relacionados a este são os correspondentes a criação do *framework* `Feature House`, uma vez que este é uma extensão da mesma. `Feature House` surgiu da proposta da abordagem genérica de composição de software superposição para POC, o que resultou na criação da ferramenta FSTComposer [Apel & Lengauer, 2008]. Posteriormente esta abordagem foi generalizada, criando o `Feature House`, juntamente com a ferramenta `FSTGenerator` que facilitou a integração de novas linguagens de programação ao *framework* [Apel et al., 2009].

Antes da criação do `Feature House`, foi criada uma extensão da ferramenta FSTComposer para a linguagem de programação C#, possibilitando que fossem feitas composições nesta linguagem utilizando a abordagem de superposição [von Rhein, 2008]. De forma similar, também foi criada uma extensão para artefatos XML [Dörre & Lengauer, 2009]. Também é importante mencionar a abordagem Color IDE (CIDE), que é parte do `Feature House`. CIDE investiga como recursos podem ser extraídos de software legado, e hoje está implementada na ferramenta chamada gCIDE, que é parte do *framework* `Feature House` [Kästner et al., 2008].

De forma geral, todo trabalho que apresenta uma forma de compor características de uma LPS utilizando a POC como base, é um trabalho relacionado. Desta forma pode-se citar o trabalho de Batory et al. [2004], no qual foi apresentada a abordagem AHEAD, que é baseada no conceito de refinamentos sucessivos. Refinamentos sucessivos é um paradigma para desenvolvimento de programas complexos a partir de um simples, através do incremento de detalhes. Funcionalidades originais são chamadas de constantes e os incrementos são chamadas de refinamentos. AHEAD é um conjunto de ferramentas que suporta o desenvolvimento e composição de características. Ele é baseado na linguagem de programação Jakarta (Jak) que é um superconjunto da linguagem Java com algumas modificações. Constantes e refinamentos são definidos em arquivos Jak, porém constantes possuem código Java puro, enquanto os refinamentos são identificados pela palavra-chave "*refines*" [Batory et al., 2004].

Outro trabalho relacionado é o chamado FeatureC++ [Apel et al., 2005] que é uma extensão da linguagem C++ para dar suporte a POC. No FeatureC++, as características são implementadas por *mixin layers*. Um *mixin layer* consiste em um conjunto colaborativo de *mixins*, onde cada um implementa um fragmento de classe. *Mixin layer* pode estar presente em várias classes. Ele usa o mesmo conceito de constantes e refinamentos que o explicado para o AHEAD no parágrafo anterior. Desta forma, constantes são os *mixins* que começam uma cadeia de refinamentos, e os demais são os refinamentos [Apel et al., 2005].

Outro trabalho relacionado é o rbFeatures [Günther & Sunkle, 2009], que é uma linguagem de domínio específica para POC. Uma linguagem de domínio específica fornece as abstrações adequadas e notações para o domínio, sendo especificamente adaptada para o mesmo. No rbFeatures, há quatro entidades de primeira classe (*Feature*, *FeatureModel*, *ProductLine* e *ProductVariant*) que são de alto nível de abstração e representam uma LPS completa e suas variações. As características são implementadas através de *mixins*, assim como o FeatureC++ [Günther & Sunkle, 2009].

5.2 Resumo das Contribuições

Esta dissertação propôs uma forma de compor software de uma LPS desenvolvida com a linguagem de programação Groovy. Inicialmente foi apresentada a linguagem Groovy, mostrando o seu crescimento e importância no cenário atual do mercado de software. O crescimento da linguagem pode ser justificado pela popularização de *frameworks* Web para Groovy que visam agilizar o desenvolvimento, como o Grails, e pelo fato de a linguagem ser parecida com Java, o que torna a curva de aprendizado para programadores Java pequena. O fato de Groovy ter sido desenvolvido para executar na Máquina Virtual Java é uma grande vantagem, uma vez que pode-se aproveitar bibliotecas e códigos Java existentes com facilidade. Além de prover diversas facilidades em relação a sintaxe em comparação com Java, Groovy também provê mecanismos interessantes como o suporte a programação funcional.

Após a apresentação da linguagem Groovy, foi discutido nesta dissertação os conceitos como POC e LPS, mostrando como modelar e implementar uma LPS de forma que facilite a composição de suas características. Neste contexto, foi apresentado o *framework* `Feature House`, e como seu conjunto de ferramentas pode ser utilizado para fazer a composição de software utilizando o paradigma de composição chamado superposição. Foi apresentada a possibilidade de integrar novas linguagens ao *framework* a partir da escrita de uma gramática no formato `FeatureBNF`. Foi apresentado o nível de granularidade utilizado na escrita da gramática Groovy, o qual considera como terminais apenas métodos e atributos.

Foi apresentada a extensão criada do *framework* `Feature House` para a linguagem Groovy, como se deu a escrita da gramática, que foi baseada em Java devido a similaridade de sintaxe das linguagens, porém com várias diferenças e adições. Discutiu-se os passos efetuados para gerar os artefatos necessários para integrar a linguagem ao `Feature House`, e posteriormente como fazer a composição de software utilizando a extensão. Foi feita uma demonstração de uso em uma LPS de exemplo que foi imple-

mentada em Groovy, mostrando como a mesma foi modelada e implementada seguindo os conceitos da POC. Também foi demonstrada uma avaliação das estruturas da linguagem Groovy suportadas pela extensão.

A principal contribuição deste trabalho foi fornecer para a comunidade uma maneira de implementar LPS utilizando a linguagem Groovy. Para isto, foi criada uma extensão de um *framework* reconhecido no meio acadêmico, que utiliza um famoso paradigma de composição de software, e sem criar muitas restrições no código a ser composto.

5.3 Trabalhos Futuros

A partir desta dissertação, trabalhos futuros podem ser desenvolvidos. Por exemplo, deve-se buscar remover as limitações impostas pela extensão de Groovy que foi criada. As restrições impostas dizem respeito a impossibilidade da extensão fazer a composição de *scripts*. A remoção destas limitações faria com que a gramática e o `Feature House` suportassem a composição de *scripts* e não exijissem que todo o código esteja dentro de uma classe e dentro de um método, com exceção de variáveis para este último ponto. Outro ponto interessante seria integrar a extensão a ferramenta `FeatureIDE` [Pereira et al., 2013]. De forma, ao selecionar as características do modelo de características de uma LPS, o arquivo com a lista de características é automaticamente criado respeitando as restrições do modelo. Assim, a composição seria feita automaticamente sem a necessidade de chamar o `Feature House` manualmente. Hoje, esta integração já existe para linguagens que fazem parte a mais tempo do `Feature House`.

Há diversas outras linguagens que estão sendo bastante utilizadas pelo mercado, como por exemplo Ruby. Um possível trabalho futuro seria integrar mais linguagens ao `Feature House`, de forma a popularizá-lo cada vez mais. De forma a garantir que a extensão criada funciona perfeitamente, um outro trabalho futuro seria avaliar a utilização da extensão em uma LPS real e de grande escala implementada em Groovy. Infelizmente, uma LPS com tais características não está disponível no momento da realização deste trabalho.

Referências Bibliográficas

Alcolea, A. B. R. P. (2016). Learn groovy in y minutes. https://learnxinyminutes.com/docs/groovy/. Acesso em Setembro/2016.

Apache (2016a). The grails framework. http://www.grails.org/. Acesso em Outubro/2016.

Apache (2016b). The groovy programming language. http://www.groovy-lang.org/. Acesso em Outubro/2016.

Apel, S. & Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49--84.

Apel, S.; Kastner, C. & Lengauer, C. (2009). Featurehouse: Language-independent, automated software composition. Em *Proceedings of the 31st International Conference on Software Engineering*, pp. 221--231. IEEE Computer Society.

Apel, S.; Leich, T.; Rosenmüller, M. & Saake, G. (2005). Featurec++: feature-oriented and aspect-oriented programming in c++. Em *Proceedings of 4th International Conference on Generative Programming and Component Engineering*. Citeseer.

Apel, S. & Lengauer, C. (2008). Superimposition: A language-independent approach to software composition. Em *International Conference on Software Composition*, pp. 20--35. Springer.

Apel, S.; Lengauer, C.; Möller, B. & Kästner, C. (2008). An algebra for features and feature composition. Em *International Conference on Algebraic Methodology and Software Technology*, pp. 36--50. Springer.

Batory, D. (2005). Feature models, grammars, and propositional formulas. Em *International Conference on Software Product Lines*, pp. 7--20. Springer.

Batory, D.; Sarvela, J. N. & Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355--371.

Clements, P. & Northrop, L. (2002). *Software product lines*. Addison-Wesley,.

Cognisync, P. K. O. (2016). Java compiler compiler (javacc) - the java parser generator. https://javacc.java.net/.

Crnković, I. (2003). Component-based software engineering-new challenges in software development. *CIT. Journal of computing and information technology*, 11(3):151--161.

Deitel, H. M. & Deitel, P. J. (2005). Java: Como programar, 6a. edição.

Dikel, D.; Kane, D.; Ornburn, S.; Loftus, W. & Wilson, J. (1997). Applying software product-line architecture. *Computer*, 30(8):49--55.

Do Vale, G. A. & Figueiredo, E. M. L. (2015). A method to derive metric thresholds for software product lines. Em *Software Engineering (SBES), 2015 29th Brazilian Symposium on*, pp. 110--119. IEEE.

Dörre, J. & Lengauer, C. (2009). Feature-oriented composition of xml artifacts. *master's thesis, Dept. of Informatics and Math., Univ. of Passau*.

Figueiredo, E.; Cacho, N.; Sant'Anna, C.; Monteiro, M.; Kulesza, U.; Garcia, A.; Soares, S.; Ferrari, F.; Khan, S.; Castor Filho, F. et al. (2008). Evolving software product lines with aspects. Em *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 261--270. IEEE.

Gacek, C. & Anastasopoules, M. (2001). Implementing product line variabilities. Em *ACM SIGSOFT Software Engineering Notes*, volume 26, pp. 109--117. ACM.

Gaia, F. N.; Ferreira, G. C. S.; Figueiredo, E. & de Almeida Maia, M. (2014). A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Science of Computer Programming*, 96:230--253.

Günther, S. & Sunkle, S. (2009). Feature-oriented programming with ruby. Em *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pp. 11--18. ACM.

Heineman, G. T. & Councill, W. T. (2001). Component-based software engineering. *Putting the pieces together, addison-westley*, p. 5.

HotFrameworks (2016). Web framework ranking | hotframeworks. http://www.hotframeworks.com/.

Hu, Y.; Merlo, E.; Dagenais, M. & Laguë, B. (2000). C/c++ conditional compilation analysis using symbolic execution. Em *Software Maintenance, 2000. Proceedings. International Conference on*, pp. 196--206. IEEE.

Kang, K. C.; Kim, S.; Lee, J.; Kim, K.; Shin, E. & Huh, M. (1998). Form: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*, 5(1):143--168.

Kastner, C.; Thum, T.; Saake, G.; Feigenspan, J.; Leich, T.; Wielgorz, F. & Apel, S. (2009). Featureide: A tool framework for feature-oriented software development. Em *Proceedings of the 31st International Conference on Software Engineering*, pp. 611--614. IEEE Computer Society.

Kästner, C.; Trujillo, S. & Apel, S. (2008). Visualizing software product line variabilities in source code. Em *SPLC (2)*, pp. 303--312.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M. & Irwin, J. (1997). Aspect-oriented programming. Em *European conference on object-oriented programming*, pp. 220--242. Springer.

Koenig, D.; Glover, A.; King, P.; Laforge, G. & Skeet, J. (2007). *Groovy in action*, volume 1. Manning.

Layka, V.; Judd, C. M.; Nusairat, J. F. & Shingler, J. (2013). *Beginning Groovy, Grails and Griffon*. Springer.

Marcotty, M. & Ledgard, H. (2012). *The world of programming languages*. Springer Science & Business Media.

Metzger, A. & Pohl, K. (2014). Software product line engineering and variability management: achievements and challenges. Em *Proceedings of the on Future of Software Engineering*, pp. 70--84. ACM.

Norvell, T. S. (2007). The javacc tutorial.

Pereira, J. A.; Souza, C.; Figueiredo, E.; Abilio, R.; Vale, G. & Costa, H. A. X. (2013). Software variability management: An exploratory study with two feature modeling tools. Em *Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium on*, pp. 20--29. IEEE.

Pohl, K.; Böckle, G. & van Der Linden, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

Souza, C. & Figueiredo, E. (2014). How do programmers use optional typing?: an empirical study. Em *Proceedings of the 13th international conference on Modularity*, pp. 109--120. ACM.

TIOBE, T. S. Q. C. (2016). Tiobe index. http://www.tiobe.com/tiobe-index/.

vd Linden, F.; Schmid, K. & Rommes, E. (2007). Software product lines in action: The best industrial practice in product line engineering. secaucus.

von Rhein, A. (2008). Feature-orientierte program-mierung mit superimposition in c#.

Weissmann, H. L. (2015). *Falando de Grails*. Casa do Código.

Apêndice A

Gramática de Groovy em FeatureBNF

```
1
2  options {
3    JAVA_UNICODE_ESCAPE = true;
4    ERROR_REPORTING = true;
5    STATIC = false;
6    USER_CHAR_STREAM = true;
7  }
8
9
10 PARSER_BEGIN(GroovyParser)
11 package de.ovgu.cide.fstgen.parsers.generated_groovy;
12
13 import java.io.*;
14 import java.util.*;
15 import cide.gast.*;
16 import cide.gparser.*;
17 import de.ovgu.cide.fstgen.ast.*;
18
19
20 public class GroovyParser extends AbstractFSTParser
21 {
22         public GroovyParser(){}
23 }
24
25 PARSER_END(GroovyParser)
26
27 SPECIAL_TOKEN :
28 {
29   " "
30 | "\t"
31 | "\n"
32 | "\r"
33 | "\f"
34 | "\b"
```

```
| "\\"
| "\'"
| "\""
}

MORE :
{
  "//" : IN_SINGLE_LINE_COMMENT
|
  <"/**" ~["/"]> { input_stream.backup(1); } : IN_FORMAL_COMMENT
|
  "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_FORMAL_COMMENT>
SPECIAL_TOKEN :
{
  <FORMAL_COMMENT: "*/" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_FORMAL_COMMENT,IN_MULTI_LINE_COMMENT>
MORE :
{
  < ~[] >
}

TOKEN :
{
  < ABSTRACT: "abstract" >
| < AS: "as" >
| < ASSERT: "assert" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
| < CATCH: "catch" >
| < CHAR: "char" >
| < CLASS: "class" >
| < CONST: "const" >
| < CONTINUE: "continue" >
| < _DEFAULT: "default" >
| < DO: "do" >
| < DOUBLE: "double" >
```

```
90  | < ELSE: "else" >
91  | < ENUM: "enum" >
92  | < EXTENDS: "extends" >
93  | < FALSE: "false" >
94  | < FINAL: "final" >
95  | < FINALLY: "finally" >
96  | < FLOAT: "float" >
97  | < FOR: "for" >
98  | < GOTO: "goto" >
99  | < IF: "if" >
100 | < IMPLEMENTS: "implements" >
101 | < IMPORT: "import" >
102 | < IN: "in" >
103 | < INSTANCEOF: "instanceof" >
104 | < INT: "int" >
105 | < INTERFACE: "interface" >
106 | < LONG: "long" >
107 | < NATIVE: "native" >
108 | < NEW: "new" >
109 | < NULL: "null" >
110 | < PACKAGE: "package">
111 | < PRIVATE: "private" >
112 | < PROTECTED: "protected" >
113 | < PUBLIC: "public" >
114 | < RETURN: "return" >
115 | < SHORT: "short" >
116 | < STATIC: "static" >
117 | < STRICTFP: "strictfp" >
118 | < SUPER: "super" >
119 | < SWITCH: "switch" >
120 | < SYNCHRONIZED: "synchronized" >
121 | < THIS: "this" >
122 | < THREADSAFE : "threadsafe" >
123 | < THROW: "throw" >
124 | < THROWS: "throws" >
125 | < TRANSIENT: "transient" >
126 | < TRUE: "true" >
127 | < TRY: "try" >
128 | < VOID: "void" >
129 | < VOLATILE: "volatile" >
130 | < WHILE: "while" >
131 | }
132 |
133 | TOKEN :
134 | {
135 |   < INTEGER_LITERAL:
136 |         <DECIMAL_LITERAL> (["l","L"])?
137 |       | <HEX_LITERAL> (["l","L"])?
138 |       | <OCTAL_LITERAL> (["l","L"])?
139 |   >
140 | |
141 |   < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
142 | |
143 |   < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
144 | |
```

```
145   < #OCTAL_LITERAL: "0" (["0"-"7"])* >
146 |
147   < FLOATING_POINT_LITERAL:
148         (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
149       | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
150       | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
151       | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
152   >
153 |
154   < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
155 |
156   < CHARACTER_LITERAL:
157       "\'"
158       (   (~["\'","\\","\n","\r"])
159         | ("\\"
160             ( ["n","t","b","r","f","u","\\","'","\""]
161             | ["0"-"7"] ( ["0"-"7"] )?
162             | ["0"-"3"] ["0"-"7"] ["0"-"7"]
163             )
164           )
165       )*
166       "\'"
167   >
168 |
169   < STRING_LITERAL:
170       "\""
171       (   (~["\"","\\","\n","\r"])
172         | ("\\"
173             ( ["n","t","b","r","f","u","\\","'","\""]
174             | ["0"-"7"] ( ["0"-"7"] )?
175             | ["0"-"3"] ["0"-"7"] ["0"-"7"]
176             )
177           )
178       )*
179       "\""
180   >
181 }
182
183 TOKEN :
184 {
185   < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
186 |
187   < #LETTER:
188       [
189        "\u0024",
190        "\u0041"-"\u005a",
191        "\u005f",
192        "\u0061"-"\u007a",
193        "\u00c0"-"\u00d6",
194        "\u00d8"-"\u00f6",
195        "\u00f8"-"\u00ff",
196        "\u0100"-"\u1fff",
197        "\u3040"-"\u318f",
198        "\u3300"-"\u337f",
199        "\u3400"-"\u3d2d",
```

```
200         "\u4e00"-"\u9fff",
201         "\uf900"-"\ufaff"
202        ]
203   >
204 |
205   < #DIGIT:
206        [
207         "\u0030"-"\u0039",
208         "\u0660"-"\u0669",
209         "\u06f0"-"\u06f9",
210         "\u0966"-"\u096f",
211         "\u09e6"-"\u09ef",
212         "\u0a66"-"\u0a6f",
213         "\u0ae6"-"\u0aef",
214         "\u0b66"-"\u0b6f",
215         "\u0be7"-"\u0bef",
216         "\u0c66"-"\u0c6f",
217         "\u0ce6"-"\u0cef",
218         "\u0d66"-"\u0d6f",
219         "\u0e50"-"\u0e59",
220         "\u0ed0"-"\u0ed9",
221         "\u1040"-"\u1049"
222        ]
223   >
224 }
225
226 TOKEN :
227 {
228   < LPAREN: "(" >
229 | < RPAREN: ")" >
230 | < LBRACE: "{" >
231 | < RBRACE: "}" >
232 | < LBRACKET: "[" >
233 | < RBRACKET: "]" >
234 | < SEMICOLON: ";" >
235 | < COMMA: "," >
236 | < DOT: "." >
237 | < AT: "@" >
238 }
239
240 TOKEN :
241 {
242   < ASSIGN: "=" >
243 | < LT: "<" >
244 | < BANG: "!" >
245 | < TILDE: "~" >
246 | < HOOK: "?" >
247 | < COLON: ":" >
248 | < EQ: "==" >
249 | < LE: "<=" >
250 | < GE: ">=" >
251 | < NE: "!=" >
252 | < SC_OR: "||" >
253 | < SC_AND: "&&" >
254 | < INCR: "++" >
```

```
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < REM: "%" >
| < LSHIFT: "<<" >
| < PLUSASSIGN: "+=" >
| < MINUSASSIGN: "-=" >
| < STARASSIGN: "*=" >
| < SLASHASSIGN: "/=" >
| < ANDASSIGN: "&=" >
| < ORASSIGN: "|=" >
| < XORASSIGN: "^=" >
| < REMASSIGN: "%=" >
| < LSHIFTASSIGN: "<<=" >
| < RSIGNEDSHIFTASSIGN: ">>=" >
| < RUNSIGNEDSHIFTASSIGN: ">>>=" >
| < ELLIPSIS: "..." >
| < ARROW: "->" >
| < SPREAD: "*." >
| < SAFE: "?." >
}

TOKEN :
{
//  < RUNSIGNEDSHIFT: ">>>" >
//| < RSIGNEDSHIFT: ">>" >
 < GT: ">" >
}

GRAMMARSTART

@FSTNonTerminal(name="{TypeDeclaration}")
CompilationUnit:
LOOK_AHEAD(2)
   [ PackageDeclaration ]
   ( ImportDeclaration )*
   ( TypeDeclaration )*
   <EOF>
|
LOOK_AHEAD(2)
   Expression
|
LOOK_AHEAD(2)
   Statement
;

PackageDeclaration:
   "package" Name @! [<SEMICOLON>]
;
```

```
310 @FSTTerminal(name="{Name}{ImportPackage}")
311 ImportDeclaration:
312   "import" [ "static" <NONE> ] Name [ImportPackage] @! [<SEMICOLON>]
313 ;
314
315 @FSTExportName(".*")
316 ImportPackage:
317         <NONE> "." "*"
318 ;
319
320 @FSTTerminal(compose="ModifierListSpecialization")
321 Modifiers:
322   (LL(2) Modifier)*
323 ;
324
325 Modifier:
326    "public" <NONE>
327   |
328    "static" <NONE>
329   |
330    "protected"<NONE>
331   |
332    "private" <NONE>
333   |
334    "final" <NONE>
335   |
336    "abstract" <NONE>
337   |
338    "synchronized" <NONE>
339   |
340    "native" <NONE>
341   |
342    "transient"<NONE>
343   |
344    "volatile" <NONE>
345   |
346    "strictfp" <NONE>
347   |
348    Annotation
349 ;
350
351 @FSTTerminal(name="{<IDENTIFIER>}")
352 Id: <IDENTIFIER>;
353
354 TypeDeclaration:
355         @FSTTerminal(name="{AUTO}")
356         ";"<NONE> :: EmptyTypeDecl
357 |
358         @FSTNonTerminal(name="{Id}")
359         LOOK_AHEAD("Modifiers()␣\"@\"␣\"interface\"")
360          Modifiers AnnotationTypeDeclaration :: AnnotationTypeDeclaration
361 |
362         @FSTNonTerminal(name="{Id}")
363         LOOK_AHEAD(2)
364          Modifiers  ClassOrInterfaceDeclaration  :: ClassDeclaration
```

```
365 |
366         @FSTNonTerminal(name="{Id}")
367         LOOK_AHEAD(2)
368         Modifiers    "enum" Id [ ImplementsList ]  EnumBody :: EnumDecl
369 ;
370
371 @FSTInline
372 AnnotationTypeDeclaration: "@" "interface" Id AnnotationTypeBody;
373
374 @FSTInline
375 ClassOrInterfaceDeclaration:
376                 ClassOrInterface Id [TypeParameters] [ExtendsList] [ImplementsList]
    ClassOrInterfaceBody;
377
378 ClassOrInterface: "class"<NONE> | "interface"<NONE>;
379
380 ExtendsList:
381    "extends" &LI  ClassOrInterfaceType  ( "," &LI  ClassOrInterfaceType )*
382 ;
383
384 @FSTTerminal(compose="ImplementsListMerging")
385 ImplementsList:
386    "implements" &LI ClassOrInterfaceType  ( "," &LI ClassOrInterfaceType )*
387 ;
388
389 @FSTInline
390 EnumDeclaration:
391   "enum" Id
392   [ ImplementsList ]
393   EnumBody
394 ;
395
396 @FSTInline
397 EnumBody:
398    "{"
399    &LI EnumConstant ( "," &LI EnumConstant )*
400    [ EnumBodyInternal ]
401    "}"
402 ;
403
404 @FSTNonTerminal(name="")
405 EnumBodyInternal: ";" ( ClassOrInterfaceBodyDeclaration )* ;
406
407 EnumConstant:
408   <IDENTIFIER> [ Arguments ] [ ClassOrInterfaceBody ]
409     |
410   <NONE>
411 ;
412
413 TypeParameters:
414    "<" &LI TypeParameter ( "," &LI TypeParameter )* ">"
415 ;
416
417 TypeParameter:
418    <IDENTIFIER> [ TypeBound ]
```

```
419 ;
420
421 TypeBound:
422    "extends" ClassOrInterfaceType ( "&" ClassOrInterfaceType )*
423 ;
424
425 @FSTInline
426 ClassOrInterfaceBody:
427   "{" @+ @! ( ClassOrInterfaceBodyDeclaration @! @! )*  @- @! "}"
428 ;
429
430 ClassOrInterfaceBodyDeclaration:
431         @FSTTerminal(name="{AUTO}")
432         LOOK_AHEAD(2)
433         Initializer :: InitializerDecl
434 |
435         @FSTNonTerminal(name="{Id}")
436         LOOK_AHEAD("Modifiers()␣ClassOrInterface()")
437         Modifiers  ClassOrInterfaceDeclaration :: InnerClassDecl
438 |
439         @FSTNonTerminal(name="{Id}")
440         LOOK_AHEAD("Modifiers()␣\"enum\"")
441         Modifiers EnumDeclaration :: InnerEnumDecl
442 |
443         @FSTTerminal(name="{ConstructorDeclaration}",
444         compose="ConstructorConcatenation")
445     LOOK_AHEAD("␣Modifiers()␣[␣TypeParameters()␣]␣<IDENTIFIER>␣\"(\"" )
446         Modifiers ConstructorDeclaration :: ConstructorDecl
447 |
448         @FSTTerminal(name="{FieldDeclaration}",compose="FieldOverriding")
449     LOOK_AHEAD( "Modifiers()␣Type()␣<IDENTIFIER>
450 ␣␣␣␣(␣\"[\"␣\"]\"␣)*␣(␣\",\"␣|␣\"=\"␣|␣\";\"␣|␣\"{\"␣)␣" )
451         Modifiers FieldDeclaration :: FieldDecl
452 |
453         LOOK_AHEAD(3)
454         Expression
455 |
456         @FSTTerminal(name="{MethodDeclaration}",compose="JavaMethodOverriding")
457         LOOK_AHEAD(2)
458         Modifiers MethodDeclaration :: MethodDecl
459 |
460         @FSTTerminal(name="{AUTO}")
461         ";" <NONE> :: EmptyDecl
462 ;
463
464 @FSTExportName("{VariableDeclarator}")
465 FieldDeclaration:
466   Type VariableDeclarator ( "," VariableDeclarator )* [<SEMICOLON>]
467 ;
468
469 @FSTExportName("{VariableDeclaratorId}")
470 VariableDeclarator:
471   VariableDeclaratorId [ "=" VariableInitializer ]
472 ;
473
```

```
474 @FSTExportName("{<IDENTIFIER>}")
475 VariableDeclaratorId:
476   <IDENTIFIER> ( "[" "]" <NONE> )*
477 ;
478
479 EmptyMapInitializer:
480         "[" <COLON> "]"
481 ;
482
483 EmptyListInitializer:
484         "[" <NONE> "]"
485 ;
486
487 VariableInitializer:
488 LOOK_AHEAD(2)
489   EmptyListInitializer
490 |
491 LOOK_AHEAD(2)
492   EmptyMapInitializer
493 |
494   Expression
495 ;
496
497 ArrayInitializer:
498   "{" [ ArrayInitializerInternal ] [ ","<NONE> ] "}"
499 ;
500
501 ArrayInitializerInternal:
502         VariableInitializer ( LOOK_AHEAD(2) "," VariableInitializer )*
503 ;
504
505 @FSTExportName("{MethodDeclarator}")
506 MethodDeclaration:
507   [ TypeParameters ]
508   ResultType
509   MethodDeclarator [ "throws" NameList ]
510   MethodDeclarationBody
511 ;
512
513 MethodDeclarationBody: Block | ";"<NONE>;
514
515 @FSTExportName("{<IDENTIFIER>}({FormalParameters})")
516 MethodDeclarator:
517   <IDENTIFIER> FormalParameters ( "[" "]"<NONE> )*
518 ;
519
520 @FSTExportName("{FormalParametersInternal}")
521 FormalParameters:
522   "(" [ FormalParametersInternal ] ")"
523 ;
524
525 @FSTExportName("{FormalParameter}^-")
526 FormalParametersInternal:
527 &LI FormalParameter ( "," &LI FormalParameter )*;
528
```

```
529 @FSTExportName("{Type}")
530 FormalParameter:
531   [ LOOK_AHEAD("Annotation()") Annotation ] [ "final" <NONE>] Type [ "..."<NONE> ]
532   VariableDeclaratorId
533 ;
534
535 @FSTInline @FSTExportName("{<IDENTIFIER>}({FormalParameters})")
536 ConstructorDeclaration:
537   [ TypeParameters ]
538   <IDENTIFIER> FormalParameters [ "throws" NameList ]
539   "{"
540     [ LOOK_AHEAD("ExplicitConstructorInvocation()") ExplicitConstructorInvocation ]
541     ( BlockStatement )*
542   "}"
543 ;
544
545 ExplicitConstructorInvocation:
546   LOOK_AHEAD("\"this\" Arguments() \";\"")
547   "this" Arguments ";" @!
548 |
549   [ LOOK_AHEAD(2) PrimaryExpression "." ] "super" Arguments ";" @!
550 ;
551
552 Initializer:
553   [ "static"<NONE> ] Block
554 ;
555
556 @FSTExportName("{TOSTRING}")
557 Type:
558    LOOK_AHEAD(2) ReferenceTypeP
559  |
560    PrimitiveType
561 ;
562
563 ReferenceTypeP:
564    PrimitiveType ( LOOK_AHEAD(2) "[" "]" <NONE> )+
565   |
566    ClassOrInterfaceType ( LOOK_AHEAD(2) "[" "]" <NONE> )*
567 ;
568
569 ClassOrInterfaceType:
570   <IDENTIFIER> [ LOOK_AHEAD(2) TypeArguments ]
571   ( LOOK_AHEAD(2) ClassOrInterfaceTypeIntern )*
572 ;
573
574 ClassOrInterfaceTypeIntern:
575  "." <IDENTIFIER> [ LOOK_AHEAD(2) TypeArguments ]
576 ;
577
578 TypeArguments:
579    "<" TypeArgument ( "," TypeArgument )* ">"
580 ;
581
582 TypeArgument:
583    ReferenceTypeP
```

```
584  |
585     "?" [ WildcardBounds ]
586  ;
587
588  WildcardBounds:
589     "extends" ReferenceTypeP
590   |
591     "super" ReferenceTypeP
592  ;
593
594  PrimitiveType:
595    "boolean"<NONE>
596  |
597    "char"<NONE>
598  |
599    "byte"<NONE>
600  |
601    "short"<NONE>
602  |
603    "int"<NONE>
604  |
605    "long"<NONE>
606  |
607    "float"<NONE>
608  |
609    "double"<NONE>
610  ;
611
612  ResultType:
613    "void"<NONE>
614  |
615    Type
616  ;
617
618  @FSTExportName("{<IDENTIFIER>}^.")
619  Name:
620    <IDENTIFIER>
621    ( LOOK_AHEAD(2) "." <IDENTIFIER> )*
622  ;
623
624  NameList:
625    &LI Name ( "," &LI Name )*
626  ;
627
628  Expression:
629          ConditionalExpression [ LOOK_AHEAD(2) AssignExp ]
630  ;
631
632  AssignExp :
633          AssignmentOperator Expression
634  ;
635
636  AssignmentOperator:
637    "="<NONE> | "*="<NONE> | "/="<NONE> | "%="<NONE> | "+="<NONE> |
638    "-="<NONE> | "<<=" <NONE>| ">>="<NONE>| ">>>="<NONE> | "&="<NONE>
```

```
639   | "^="<NONE> | "|="<NONE> | "<<"<NONE> | "in"<NONE>
640 ;
641
642 ConditionalExpression:
643                 LOOK_AHEAD("ConditionalOrExpression()␣\"?\"")
644                  ConditionalExpressionFull{Expression}
645         |
646                 LOOK_AHEAD(3)
647                  ConditionalOrExpression
648         |
649                 LOOK_AHEAD(3)
650                  ClosureExpression
651         |
652                 LOOK_AHEAD(3)
653                  MapInitializer
654         |
655                 LOOK_AHEAD(3)
656                  ArrayCreation
657 ;
658
659 MapInitializer:
660   "[" Expression ":" Expression (MapInitializarInternal)* "]"
661 ;
662
663 MapInitializarInternal:
664   <COMMA> Expression ":" Expression
665 ;
666
667 ClosureExpression:
668   "{" [<IDENTIFIER>] (ClosureParameterInitializer)*  [<ARROW>] (Expression)* "}"
669 ;
670
671 ClosureParameterInitializer:
672   <COMMA> <IDENTIFIER>
673 ;
674
675 ArrayCreation:
676   "(" Expression ".." Expression ")"
677 ;
678
679 ConditionalExpressionFull:
680   ConditionalOrExpression "?" [Expression] ":" Expression
681 ;
682
683 ConditionalOrExpression:
684   ConditionalAndExpression ( "||" ConditionalAndExpression )*
685 ;
686
687 ConditionalAndExpression:
688   InclusiveOrExpression ( "&&" InclusiveOrExpression )*
689 ;
690
691 InclusiveOrExpression:
692   ExclusiveOrExpression ( "|" ExclusiveOrExpression )*
693 ;
```

```
694
695 ExclusiveOrExpression:
696   AndExpression ( "^" AndExpression )*
697 ;
698
699 AndExpression:
700   EqualityExpression ( "&" EqualityExpression )*
701 ;
702
703 EqualityExpression:
704   InstanceOfExpression ( EqualityExpressionIntern )*
705 ;
706
707 EqualityExpressionIntern: EqualityOp InstanceOfExpression;
708
709 EqualityOp:  "=="<NONE> | "!="<NONE>;
710
711 InstanceOfExpression:
712   RelationalExpression [ "instanceof" Type ]
713 ;
714
715 RelationalExpression:
716   ShiftExpression ( RelationalExpressionIntern )*
717 ;
718
719 RelationalExpressionIntern:RelationalOp ShiftExpression;
720
721 RelationalOp: "<"<NONE> | ">"<NONE> | "<=" <NONE>| ">="<NONE> ;
722
723 ShiftExpression:
724   AdditiveExpression ( LL(2) ShiftExpressionRight )*
725 ;
726
727 ShiftExpressionRight:
728         ShiftOp AdditiveExpression
729 ;
730
731 ShiftOp: "<<"<NONE> | LOOK_AHEAD(3) ">" ">" ">"   <NONE> | LL(2) ">" ">"  <NONE>;
732
733 AdditiveExpression:
734   MultiplicativeExpression (AdditiveExpressionIntern )*
735 ;
736
737 AdditiveExpressionIntern:AdditiveOp MultiplicativeExpression ;
738
739 AdditiveOp: "+"<NONE> | "-"<NONE>;
740
741 MultiplicativeExpression:
742   UnaryExpression (MultiplicativeExpressionIntern)*
743 ;
744
745 MultiplicativeExpressionIntern:  MultiplicativeOp UnaryExpression ;
746 MultiplicativeOp: "*" <NONE>| "/" <NONE>| "%"<NONE>;
747
748 UnaryExpression:
```

```
749   AdditiveOp UnaryExpression
750 |
751   PreIncrementExpression
752 |
753   PreDecrementExpression
754 |
755   UnaryExpressionNotPlusMinus
756 ;
757
758 PreIncrementExpression:
759   "++" PrimaryExpression
760 ;
761
762 PreDecrementExpression:
763   "--" PrimaryExpression
764 ;
765
766 UnaryExpressionNotPlusMinus:
767   UnaryOp UnaryExpression
768 |
769   LOOK_AHEAD( "CastLookahead()" )
770   CastExpression
771 |
772   PostfixExpression
773 ;
774
775 UnaryOp: "~" <NONE>| "!"<NONE>;
776
777 CastLookahead:
778   LOOK_AHEAD(2)
779   "(" PrimitiveType
780 |
781   LOOK_AHEAD("\"(\"␣Type()␣\"[\"")
782   "(" Type "[" "]"
783 |
784    "(" Type ")" CastLAOp
785 ;
786
787 CastLAOp:
788       "~"<NONE> | "!"<NONE> | "(" <NONE>| "this"<NONE> | "super"<NONE> | "new"<NONE> |
789           <IDENTIFIER> |
790           Literal;
791
792 PostfixExpression:
793   PrimaryExpression [ PostfixOp ]
794 ;
795
796 PostfixOp: "++"<NONE> | "--"<NONE>;
797
798 CastExpression:
799   LOOK_AHEAD("\"(\"␣PrimitiveType()")
800   "(" Type ")" UnaryExpression
801 |
802   "(" Type ")" UnaryExpressionNotPlusMinus
803 ;
```

```
804 |
805 | PrimaryExpression:
806 |   PrimaryPrefix ( LOOK_AHEAD(2) PrimarySuffix )*
807 | ;
808 |
809 | MemberSelector:
810 |   "." TypeArguments <IDENTIFIER>
811 | ;
812 |
813 | PrimaryPrefix:
814 | LOOK_AHEAD(3)
815 |   Literal
816 | |
817 |   "this"<NONE>
818 | |
819 |   "super" "." <IDENTIFIER>
820 | |
821 | LOOK_AHEAD(3)
822 |   "[" ArgumentList "]"
823 | |
824 | LOOK_AHEAD(3)
825 |   "(" Expression ")"
826 | |
827 | LOOK_AHEAD(3)
828 |   AllocationExpression
829 | |
830 |  LOOK_AHEAD("␣ResultType()␣\".\"␣\"class\"" )
831 |   ResultType "." "class"
832 | |
833 |   Name
834 | |
835 |   "(" ArgumentList ")"
836 | ;
837 |
838 | PrimarySuffix:
839 |  LOOK_AHEAD(2)
840 |   "." "this"<NONE>
841 | |
842 |  LOOK_AHEAD(2)
843 |   "." "super"<NONE>
844 | |
845 |  LOOK_AHEAD(2)
846 |   "." AllocationExpression
847 | |
848 |  LOOK_AHEAD(3)
849 |   MemberSelector
850 | |
851 |   "[" Expression "]"
852 | |
853 |   "." <IDENTIFIER>
854 | |
855 |   Arguments
856 | ;
857 |
858 | Literal:
```

```
859   <INTEGER_LITERAL>
860 |
861   <FLOATING_POINT_LITERAL>
862 |
863   <CHARACTER_LITERAL>
864 |
865   <STRING_LITERAL>
866 |
867   BooleanLiteral
868 |
869   NullLiteral
870 ;
871
872 BooleanLiteral:
873   "true"<NONE>
874 |
875   "false"<NONE>
876 ;
877
878 NullLiteral:
879   "null"<NONE>
880 ;
881
882 Arguments:
883   "(" [ ArgumentList ] ")"
884 ;
885
886 ArgumentList:
887   Expression ( "," Expression )*
888 ;
889
890 AllocationExpression:
891   LOOK_AHEAD(2)
892   "new" PrimitiveType ArrayDimsAndInits
893 |
894   "new" ClassOrInterfaceType [ TypeArguments ] AllocationExpressionInit
895 ;
896
897 AllocationExpressionInit:
898 LOOK_AHEAD(5)
899         <LPAREN> Statement "." "in" <RPAREN>
900         |
901 LOOK_AHEAD(3)
902         <LPAREN> (ExpandoParameter)* <RPAREN>
903         |
904       ArrayDimsAndInits
905     |
906       Arguments [ ClassOrInterfaceBody ]
907 ;
908
909 ExpandoParameter:
910         Expression ":" Expression
911 ;
912
913 ArrayDimsAndInits:
```

```
914   LOOK_AHEAD(2) "[" Expression "]" ( LOOK_AHEAD(2)
915   "[" Expression "]" )* ( LOOK_AHEAD(2) "[" "]" <NONE> )*
916 |
917   ( "[" "]"<NONE> )+ ArrayInitializer
918 ;
919
920 Statement:
921   LOOK_AHEAD(2)
922   LabeledStatement
923 |
924   AssertStatement [<SEMICOLON>] @!
925 |
926   Block
927 |
928   EmptyStatement
929 |
930   StatementExpression [<SEMICOLON>] @!
931 |
932   SwitchStatement @!
933 |
934   IfStatement{Statement} @!
935 |
936   WhileStatement{Statement} @!
937 |
938   DoStatement{Statement} @!
939 |
940   ForStatement{Statement} @!
941 |
942   BreakStatement @!
943 |
944   ContinueStatement @!
945 |
946   ReturnStatement @!
947 |
948   ThrowStatement @!
949 |
950   SynchronizedStatement{Statement} @!
951 |
952   TryStatement{Statement} @!
953 |
954   ImportDeclaration [<SEMICOLON>] @!
955 ;
956
957 AssertStatement:
958   "assert" Expression [ ":" Expression ] [<SEMICOLON>]
959 ;
960
961 LabeledStatement:
962   <IDENTIFIER> ":" Statement
963 ;
964
965 Block:
966   "{" @+  @! ( BlockStatement )* @- "}" @!
967 ;
968
```

```
969  BlockStatement:
970    LOOK_AHEAD("[ \"final\" ] Type() <IDENTIFIER>")
971    LocalVariableDeclaration [<SEMICOLON>]
972  |
973    Statement
974  |
975    Annotation
976  |
977    Modifiers   ClassOrInterfaceDeclaration
978  |
979    <COMMA>
980  |
981    <ARROW>
982  |
983    <DOT>
984  |
985    <SPREAD>
986  |
987    <SAFE>
988  ;
989
990  LocalVariableDeclaration:
991    [ "final"<NONE> ] Type VariableDeclarator ( "," VariableDeclarator )*
992  ;
993
994  EmptyStatement:
995    ";"<NONE>
996  ;
997
998  StatementExpression:
999    PreIncrementExpression
1000   | PreDecrementExpression
1001   | PrimaryExpression [ StatementExpressionAssignment ]
1002 ;
1003
1004 StatementExpressionAssignment:
1005         "++"<NONE> |
1006         "--"<NONE> |
1007         AssignmentOperator Expression
1008 ;
1009
1010 SwitchStatement:
1011   "switch" "(" Expression ")" "{" @+ @!
1012     ( SwitchStatementLabel )*
1013   @- "}" @!
1014 ;
1015
1016 SwitchStatementLabel :SwitchLabel ( BlockStatement )*;
1017
1018 SwitchLabel:
1019   "case" Expression ":"
1020 |
1021   "default" ":"<NONE>
1022 ;
1023
```

```
1024 IfStatement:
1025   "if" "("Expression")" @! @+ Statement! @- [LOOK_AHEAD(1) "else" @! @+ Statement @-]
1026 ;
1027
1028 WhileStatement:
1029   "while" "(" Expression ")" @! @+ Statement! @-
1030 ;
1031
1032 DoStatement:
1033   "do" @! @+ Statement! @- "while" "(" Expression ")" [<SEMICOLON>]
1034 ;
1035
1036 ForStatement:
1037   "for" "("
1038   ForStatementInternal
1039
1040   ")" @! @+ Statement! @-
1041 ;
1042
1043 ForStatementInternal:
1044 LOOK_AHEAD(3)
1045     <IDENTIFIER> "in" <INTEGER_LITERAL> ".." <INTEGER_LITERAL>
1046       |
1047 LOOK_AHEAD(3)
1048     <IDENTIFIER> "in" ArgumentList
1049       |
1050     LOOK_AHEAD("Type()␣<IDENTIFIER>␣\":\"")
1051        [ FINAL_T ] Type <IDENTIFIER> ":" Expression
1052       |
1053     LOOK_AHEAD("FINAL_T()␣Type()␣<IDENTIFIER>␣\":\"")
1054        FINAL_T Type <IDENTIFIER> ":" Expression
1055       |
1056       [ ForInit ] ";" [ Expression ] ";" [ ForUpdate ]
1057   ;
1058
1059 FINAL_T : "final";
1060
1061 ForInit:
1062   LOOK_AHEAD( "[␣\"final\"␣]␣Type()␣<IDENTIFIER>" )
1063   LocalVariableDeclaration
1064 |
1065   StatementExpressionList
1066 ;
1067
1068 StatementExpressionList:
1069   StatementExpression ( "," StatementExpression )*
1070 ;
1071
1072 ForUpdate:
1073   StatementExpressionList
1074 ;
1075
1076 BreakStatement:
1077   "break" [ <IDENTIFIER> ] [<SEMICOLON>]
1078 ;
```

```
1079
1080 ContinueStatement:
1081   "continue" [ <IDENTIFIER> ] [<SEMICOLON>]
1082 ;
1083
1084 ReturnStatement:
1085   "return" [ Expression ] [<SEMICOLON>]
1086 ;
1087
1088 ThrowStatement:
1089   "throw" Expression [<SEMICOLON>]
1090 ;
1091
1092 SynchronizedStatement:
1093   "synchronized" "(" Expression ")" Block!
1094 ;
1095
1096 TryStatement:
1097   "try" Block! TryStatementEnd
1098 ;
1099
1100 TryStatementEnd:
1101         (CatchBlock )+ ["finally" Block ]
1102   |
1103      "finally" Block
1104 ;
1105
1106 CatchBlock:
1107    "catch" "(" FormalParameter ")" Block
1108 ;
1109
1110 Annotation:
1111    LOOK_AHEAD( "\"@\" Name() \"(\" ( <IDENTIFIER> \"=\" | \")\" )")
1112    NormalAnnotation
1113  |
1114    LOOK_AHEAD( "\"@\" Name() \"(\"" )
1115    SingleMemberAnnotation
1116  |
1117    MarkerAnnotation
1118 ;
1119
1120 NormalAnnotation:
1121    "@" Name "(" [ MemberValuePairs ] ")" @!
1122 ;
1123
1124 MarkerAnnotation:
1125   "@" Name @!
1126 ;
1127
1128 SingleMemberAnnotation:
1129   "@" Name "(" MemberValue ")" @!
1130 ;
1131
1132 MemberValuePairs:
1133    MemberValuePair ( "," MemberValuePair )*
```

```
1134 ;
1135
1136 MemberValuePair:
1137     <IDENTIFIER> "=" MemberValue
1138 ;
1139
1140 MemberValue:
1141    Annotation
1142  |
1143    MemberValueArrayInitializer
1144  |
1145    ConditionalExpression
1146  |
1147         <NONE>
1148 ;
1149
1150 MemberValueArrayInitializer:
1151   "{" MemberValue ( LOOK_AHEAD(2) "," MemberValue )* [ ","<NONE> ] "}"
1152 ;
1153
1154 @FSTInline
1155 AnnotationTypeBody:
1156   "{" ( AnnotationTypeMemberDeclaration )* "}"
1157 ;
1158
1159 AnnotationTypeMemberDeclaration:
1160         @FSTTerminal(name="{<IDENTIFIER>}")
1161         LOOK_AHEAD("Modifiers()␣Type()␣<IDENTIFIER>␣\"(\"")
1162          Modifiers Type <IDENTIFIER> "(" ")" [ DefaultValue ] ";" ::AnnotationMethodDecl
1163 |
1164         @FSTNonTerminal(name="{Id}")
1165         LOOK_AHEAD("Modifiers()␣\"@\"␣\"interface\"")
1166          Modifiers    AnnotationTypeDeclaration :: AnnotationInnerAnnotation
1167 |
1168         @FSTNonTerminal(name="{Id}")
1169         LOOK_AHEAD(1)
1170          Modifiers   ClassOrInterfaceDeclaration :: AnnotationInnerClass
1171 |
1172         @FSTNonTerminal(name="{Id}")
1173         LOOK_AHEAD(2)
1174          Modifiers    EnumDeclaration      ::AnnotationInnerEnum
1175 |
1176         @FSTTerminal(name="{FieldDeclaration}",compose="FieldOverriding")
1177         LOOK_AHEAD(1)
1178          Modifiers   FieldDeclaration ::AnnotationFieldDecl
1179  |
1180         @FSTTerminal(name="{AUTO}")
1181         ";" <NONE> :: AnnoationEmptyDecl
1182 ;
1183
1184 DefaultValue:
1185   "default" MemberValue
1186 ;
```

Apêndice B

Código do tutorial de Groovy do website *"Learn X in Y Minutes"*

```
1
2  De acordo com a ABNT:
3
4  /*
5    Prepara-se:
6
7    1) Instale a maquina virtual de Groovy - http://gvmtool.net/
8    2) Intalse o Groovy: gvm install groovy
9    3) Inicie o console groovy digitando: groovyConsole
10
11 */
12
13 // Comentario de uma linha inicia-se com duas barras
14 /*
15 Comentario de multiplas linhas sao assim.
16 */
17
18 // Ola Mundo!
19 println "Ola mundo!"
20
21 /*
22   Variaveis:
23   Voce pode atribuir valores a vari veis para uso posterior
24 */
25
26 def x = 1
27 println x
28
29 x = new java.util.Date()
30 println x
31
32 x = -3.1499392
33 println x
34
```

```
35 x = false
36 println x
37
38 x = "Groovy!"
39 println x
40
41 /*
42    Cole   es e mapeamentos
43 */
44
45 //Criando uma lista vazia
46 def tecnologias = []
47
48 /*** Adicionando elementos a lista ***/
49
50 // Assim como Java
51 tecnologias.add("Grails")
52
53 // Shift para esquerda adiciona e retorna a lista
54 tecnologias << "Groovy"
55
56 // Adi   o de multiplos elementos
57 tecnologias.addAll(["Gradle","Griffon"])
58
59 /*** Removendo elementos da lista ***/
60
61 // Assim como Java
62 tecnologias.remove("Griffon")
63
64 // Subtra   o tambem funciona
65 tecnologias = technologies - 'Grails'
66
67 /*** Iterando sobre listas ***/
68
69 // Itera sobre os elementos da lista
70 tecnologias.each { println "Tecnologias: $it"}
71 tecnologias.eachWithIndex { it, i -> println "$i: $it"}
72
73 /*** Checando os elementos da lista ***/
74
75 //Avalia se a lista cont m o elemento 'Groovy'
76 contem = tecnologias.contains( 'Groovy' )
77
78 // Ou
79 contem = 'Groovy' in tecnologias
80
81 // Checagem por multiplos elementos
82 tecnologias.containsAll(['Groovy','Grails'])
83
84 /*** Ordenando listas ***/
85
86 // Ordena a lista (altera a lista in-place)
87 tecnologias.sort()
88
89 // Para ordenar a lista sem alterar a original
```

```
90  tecnologiasOrdenadas = tecnologias.sort( false )
91
92  /*** Manipulando listas ***/
93
94  //Substitue todos os elementos da lista
95  Collections.replaceAll(tecnologias, 'Gradle', 'gradle')
96
97  //Desorganiza a lista
98  Collections.shuffle(tecnologias, new Random())
99
100 //Limpa a lista
101 technologies.clear()
102
103 //Criando um mapeamento vazio
104 def devMap = [:]
105
106 //Adicionando valores
107 devMap = ['nome':'Roberto', 'framework':'Grails', 'linguagem':'Groovy']
108 devMap.put('ultimoNome','Perez')
109
110 //Iterando sobre os elementos do mapeamento
111 devMap.each { println "$it.key: $it.value" }
112 devMap.eachWithIndex { it, i -> println "$i: $it"}
113
114 //Avalia se um mapeamento cont m uma chave
115 assert devMap.containsKey('nome')
116
117 //Avalia se um mapeamento cont m um valor
118 assert devMap.containsValue('Roberto')
119
120 //Pega as chaves de um mapeamento
121 println devMap.keySet()
122
123 //Pega os valores de um mapeamento
124 println devMap.values()
125
126 /*
127   Groovy Beans
128
129   GroovyBeans sao JavaBeans com uma sintaxe muito mais simples.
130
131   Quando Groovy    compilado para bytecode, as seguintes regras sao usadas:
132
133     * Se o nome e declarado com um modificador de acesso(public, private or
134       protected) ent o um atributo    gerado.
135
136     * Um nome declarado sem modificador de acesso gera um campo privado com
137       getter e setter p blicos (ou seja, uma propriedade).
138
139     * Se uma propriedade e declarada como final, um campo private final e criado
140       e o setter n o    gerado.
141
142     * Voce pode declarar uma propriedade e tambem declarar seus proprios getters
143       e setters.
144
```

```
145     * Voce pode declarar uma propriedade e um campo com o mesmo nome, a propriedade
146       usar  este campo.
147
148     * Se voce quer uma propriedade private ou protected, voce deve prover seus
149       proprios getters e setter, que devem ser declarados como private ou protected.
150
151     * Se voce acessar uma propriedade dentro da classe e esta propriedade e definida
152       em tempo de compila  o com 'this', implicito ou explicito (por exemplo,
153       this.foo, ou simplesmente foo), Groovy acessara este campo diretamente, sem
154       passar pelo getter ou setter.
155
156     * Se voce acessar uma propriedade que nao existe usando foo, explicitamente ou
157       implicitamente, ent o Groovy ira acessar esta propriedade atraves da meta
158       classe, o que pode falhar em tempo de execucao.
159
160 */
161
162 class Foo {
163     // propriedade de leitura, apenas
164     final String nome = "Roberto"
165
166     // propriedade de leitura, apenas, com getter e setter publicos
167     String linguagem
168     protected void setLinguagem(String linguagem) { this.linguagem = linguagem }
169
170     // propriedade tipada dinamicamente
171     def ultimoNome
172 }
173
174 /*
175   Condicionais e loops
176 */
177
178 //Groovy suporta a sintaxe if-else
179 def x = 3
180
181 if(x==1) {
182     println "Um"
183 } else if(x==2) {
184     println "Dois"
185 } else {
186     println "X e maior que Dois"
187 }
188
189 //Groovy tambem suporta o operador ternario
190 def y = 10
191 def x = (y > 1) ? "functionou" : "falhou"
192 assert x == "functionou"
193
194 //Loop 'for'
195 //Itera sobre um intervalo (range)
196 def x = 0
197 for (i in 0 .. 30) {
198     x += i
199 }
```

```
200
201 //Itera sobre uma lista
202 x = 0
203 for( i in [5,3,2,1] ) {
204     x += i
205 }
206
207 //Itera sobre um array
208 array = (0..20).toArray()
209 x = 0
210 for (i in array) {
211     x += i
212 }
213
214 //Itera sobre um mapa
215 def map = ['name':'Roberto', 'framework':'Grails', 'language':'Groovy']
216 x = 0
217 for ( e in map ) {
218     x += e.value
219 }
220
221 /*
222   Operadores
223
224   Sobrecarregamento de Operadores para uma lsita dos operadores comuns que
225   Grooby suporta:
226   http://www.groovy-lang.org/operators.html#Operator-Overloading
227
228   Operadores Groovy  teis
229 */
230 //Operador de espalhamento: invoca uma a  o sobre todos os itens de um
231 //objeto agregador.
232 def tecnologias = ['Groovy','Grails','Gradle']
233 tecnologias*.toUpperCase() // = to tecnologias.collect { it?.toUpperCase() }
234
235 //Operador de navega  o segura: usado para evitar NullPointerException.
236 def usuario = User.get(1)
237 def nomeUsuario = usuario?.nomeUsuario
238
239
240 /*
241   Closures
242   Um closure, em Grooby,  como um "bloco de codigo" ou um ponteiro para metodo.
243   E um peda  o de c digo que e definido e executado em um momento posterior.
244
245   Mais informacao em: http://www.groovy-lang.org/closures.html
246 */
247 //Exemplo:
248 def clos = { println "Hello World!" }
249
250 println "Executando o closure:"
251 clos()
252
253 //Passando parametros para um closure
254 def soma = { a, b -> println a+b }
```

```
255 soma(2,4)
256
257 //Closdures por referir-se a variaveis que nao estao listadas em sua
258 //lista de parametros.
259 def x = 5
260 def multiplicarPor = { num -> num * x }
261 println multiplicarPor(10)
262
263 // Se voce tiver um closure que tem apenas um argumento, voce pode omitir
264 // o parametro na definicao do closure
265 def clos = { print it }
266 clos( "oi" )
267
268 /*
269   Groovy pode memorizar resultados de closures [1][2][3]
270 */
271 def cl = {a, b ->
272     sleep(3000) //  simula processamento
273     a + b
274 }
275
276 mem = cl.memoize()
277
278 def chamaClosure(a, b) {
279     def inicio = System.currentTimeMillis()
280     mem(a, b)
281     println "Os inputs (a = $a, b = $b)
282 ---- tomam ${System.currentTimeMillis() - inicio} msecs."
283 }
284
285 chamaClosure(1, 2)
286 chamaClosure(1, 2)
287 chamaClosure(2, 3)
288 chamaClosure(2, 3)
289 chamaClosure(3, 4)
290 chamaClosure(3, 4)
291 chamaClosure(1, 2)
292 chamaClosure(2, 3)
293 chamaClosure(3, 4)
294
295 /*
296   Expando
297
298   A classe Expando e um bean din mico que permite adicionar propriedade e
299   closures como metodos a uma instancia desta classe
300
301   http://mrhaki.blogspot.mx/2009/10/groovy-goodness-expando-as-dynamic-bean.html
302 */
303   def usuario = new Expando(nome:"Roberto")
304   assert 'Roberto' == nome.name
305
306   nome.lastName = 'Perez'
307   assert 'Perez' == nome.lastName
308
309   nome.showInfo = { out ->
```

```
310         out << "Name: $name"
311         out << ", Last name: $lastName"
312     }
313
314     def sw = new StringWriter()
315     println nome.showInfo(sw)
316
317
318 /*
319     Metaprogramacao (MOP)
320 */
321
322 //Usando a ExpandoMetaClasse para adicionar comportamento
323 String.metaClass.testAdd = {
324     println "adicionamos isto"
325 }
326
327 String x = "teste"
328 x?.testAdd()
329
330 //Interceptando chamadas a metodos
331 class Test implements GroovyInterceptable {
332     def soma(Integer x, Integer y) { x + y }
333
334     def invocaMetodo(String name, args) {
335         System.out.println "Invoca metodo $name com argumentos: $args"
336     }
337 }
338
339 def teste = new Test()
340 teste?.soma(2,3)
341 teste?.multiplica(2,3)
342
343 //Groovy suporta propertyMissing para lidar com tentativas de resolu   o de
344 //propriedades.
345 class Foo {
346    def propertyMissing(String nome) { nome }
347 }
348 def f = new Foo()
349
350 assertEquals "boo", f.boo
351
352 /*
353     TypeChecked e CompileStatic
354     Groovy, por natureza, e e sempre ser  uma linguagem dinamica, mas ela tambem
355     suporta typecheked e compilestatic
356
357     Mais informacoes: http://www.infoq.com/articles/new-groovy-20
358 */
359 //TypeChecked
360 import groovy.transform.TypeChecked
361
362 void testeMethod() {}
363
364 @TypeChecked
```

```
365 void test() {
366     testeMethod()
367
368     def nome = "Roberto"
369
370     println noome
371
372 }
373
374 //Outro exemplo:
375 import groovy.transform.TypeChecked
376
377 @TypeChecked
378 Integer test() {
379     Integer num = "1"
380
381     Integer[] numeros = [1,2,3,4]
382
383     Date dia = numeros[1]
384
385     return "Teste"
386
387 }
388
389 //Exemplo de CompileStatic :
390 import groovy.transform.CompileStatic
391
392 @CompileStatic
393 int soma(int x, int y) {
394     x + y
395 }
396
397 assert soma(2,5) == 7
```