UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO PARA CONSULTA EM CÓDIGO ORIENTADO A ASPECTOS

DISSERTAÇÃO DE MESTRADO

Cristiano De Faveri

Santa Maria, RS, Brasil

2013

UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO PARA CONSULTA EM CÓDIGO ORIENTADO A ASPECTOS

Cristiano De Faveri

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-Graduação em Informática, Área de Concentração em Computação da Universidade Federal de Santa Maria, (UFSM, RS), como requisito parcial para obtenção do grau de

Mestre em Ciência da Computação

Orientador: Prof. Dr. Eduardo Kessler Piveta

Santa Maria, RS, Brasil

2013

De Faveri, Cristiano

Uma linguagem específica de domínio para consulta em código orientado a aspectos / por Cristiano De Faveri. – 2013.
142 f.: il.; 30 cm.

Orientador: Eduardo Kessler Piveta
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2013.

1. Programação Orientada a Aspectos. 2. Linguagem de consulta. 3. Linguagem de programação. 4. Linguagem específica de domínio. I. Piveta, Eduardo Kessler. II. Título.

E-mail: cristiano@defaveri.com.br

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO PARA CONSULTA EM CÓDIGO ORIENTADO A ASPECTOS

elaborada por
Cristiano De Faveri

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

Eduardo Kessler Piveta, Dr.
(Presidente/Orientador)

Juliana Kaizer Vizzotto, Dra. (UFSM)

Ingrid Oliveira de Nunes, Dra. (UFRGS)

Santa Maria, 28 de Agosto de 2013.

À minha amada esposa Thais

AGRADECIMENTOS

Meu eterno agradecimento a todas as pessoas que incentivaram e contribuíram direta ou indiretamente para o desenvolvimento deste trabalho.

Ao meu orientador, professor Eduardo Piveta, pelo incondicional apoio durante essa jornada. Obrigado pelas ideias, sugestões, comentários e pela confiança de sempre. Sua positividade e brilhantismo foram essenciais na condução dessa pesquisa.

Aos professores Giovani, Deise e Juliana por me assistirem com ideias e materiais durante o trabalho.

Aos meus amigos do grupo de pesquisa de linguagens de programação e banco de dados da UFSM, Jânio, Heres, Gustavo e Guinther. Meu muito obrigado pelas revisões e pelas longas horas de discussão no (con)gelado laboratório de programação da UFSM.

Aos alunos de graduação Rodrigo e Felipe, pela oportunidade em auxiliá-los em seus trabalhos de conclusão de curso.

Aos meus pais, pela base de toda minha educação e pelos valores pautados na ética e no respeito ao próximo.

Ao meu filho Murillo, por me dar a energia de cada dia.

Por fim, e em especial, à minha esposa Thais, pela sua compreensão, dedicação, apoio e carinho durante todo esse tempo. Não há como expressar em simples palavras minha gratidão por tudo que você tem feito por mim e pela nossa família. Você é realmente uma pessoa fascinante!

“Live to learn, learn to live, then teach others”
— DOUGLAS HORTON

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

UMA LINGUAGEM ESPECÍFICA DE DOMÍNIO PARA CONSULTA EM CÓDIGO ORIENTADO A ASPECTOS

AUTOR: CRISTIANO DE FAVERI
ORIENTADOR: EDUARDO KESSLER PIVETA
Local da Defesa e Data: Santa Maria, 28 de Agosto de 2013.

Assegurar a qualidade de código é um ponto crucial durante o desenvolvimento de software. Frequentemente, os desenvolvedores recorrem às ferramentas de análise estática para auxiliá-los tanto na compreensão de código, quanto na identificação de defeitos ou de oportunidades de refatoração durante o ciclo de desenvolvimento de aplicações. Um dos pontos críticos na definição de tais ferramentas está na sua capacidade de obter informações a respeito de código. As ferramentas de análise estática dependem, em geral, de uma representação intermediária de um programa para identificar situações que atendam às condições necessárias descritas em seus algoritmos. Esse panorama se amplia com o uso de técnicas de modularização de interesses transversais, tais como a programação orientada a aspectos (POA), na qual um código pode ser afetado de forma sistêmica, por meio de combinações estáticas e dinâmicas. O principal objetivo desta dissertação é a especificação e implementação de AQL, uma DSL (linguagem específica de domínio) para a realização de busca em código orientado a aspectos. A AQL é uma linguagem declarativa, baseada em linguagem de busca em objetos (OQL) e que permite consultar elementos, relações, derivações e métricas de um programa orientado a aspectos (OA), a fim de apoiar a construção de ferramentas de análise estática e de pesquisa em código. O projeto de implementação da linguagem foi realizado em duas etapas. Primeiro, foi criado um framework (AOPJungle) para a extração de dados de programas OA. O AOPJungle além de extrair dados de programas OA, realiza a computação de métricas, inferências e ligações entre os elementos de um programa. Na segunda etapa, um compilador de referência para AQL foi construído. A abordagem adotada foi a transformação fonte a fonte, sendo uma consulta AQL transformada em uma consulta HQL (*Hibernate Query Language*) antes de sua execução. A fim de avaliar a implementação proposta, uma ferramenta de análise estática para identificação de oportunidades de refatoração em programas AO foi elaborada, usando a AQL para a busca de dados sobre esses programas.

Palavras-chave: Programação Orientada a Aspectos. Linguagem de consulta. Linguagem de programação. Linguagem específica de domínio.

ABSTRACT

Master Dissertation
Post-Graduate Program in Informatics
Federal University of Santa Maria

A DOMAIN SPECIFIC LANGUAGE FOR ASPECT-ORIENTED CODE QUERY

AUTHOR: CRISTIANO DE FAVERI
ADVISOR: EDUARDO KESSLER PIVETA

Defense Place and Date: Santa Maria, August 28th, 2013.

Ensuring code quality is crucial in software development. Not seldom, developers resort to static analysis tools to assist them in both understanding pieces of code and identifying defects or refactoring opportunities during development activities. A critical issue when defining such tools is their ability to obtain information about code. Static analysis tools depend, in general, of an intermediate program representation to identify locations that meet the conditions described in their algorithms. This perspective can be enlarged when techniques of crosscutting concerns modularization, such as aspect-oriented programming (AOP) is applied. In AOP applications, a piece of code can be systematically affected, using both static and dynamic combinations. The main goal of this dissertation is the specification and the implementation of AQL, a domain-specific language (DSL) designed to search aspect-oriented code bases. AQL is a declarative language, based on object query language (OQL), which enables the task of querying elements, relationships and program metrics to support the construction of static analysis and code searching tools for aspect oriented programs. The language was designed in two steps. First, we built a framework (AOPJungle), responsible to extract data from aspect-oriented programs. AOPJungle performs the computation of metrics, inferences and connections between the elements of the program. In the second step, we built an AQL compiler as a reference implementation. We adopted a source-to-source transformation for this step, in which an AQL query is transformed into HQL statements before being executed. In order to evaluate the reference implementation, we developed a static analysis tool for identifying refactoring opportunities in aspect-oriented programs. This tool receives a set of AQL queries to identify potential scenarios where refactoring could be applied.

Keywords: Aspect Oriented Programming, Query Language, Programming Language, Domain Specific Language.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE APÊNDICES

LISTA DE ABREVIATURAS E SIGLAS

AJDT	*AspectJ Development Tools*
AQL	*Aspect Query Language*
AST	*Abstract Syntax Tree*
BNF	*Backus-Naur Form*
COTS	*Commercial Off-The-Shelf*
DSL	*Domain-Specific Language*
EBNF	*Extended Backus-Naur Form*
GPL	*General Purpose Language*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
OA	Orientação a Aspectos
OO	Orientação a Objetos
OQL	*Object Query Language*
POA	Programação Orientada a Aspectos
UML	*Unified Modelling Language*
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

À medida que os sistemas de software se tornam mais complexos, com milhares de linhas espalhadas em centenas de módulos, garantir a qualidade do código produzido é um constante desafio. O processo de modificar um sistema de software, chamado de manutenção ou de evolução (SOMMERVILLE, 2010), pode variar entre diferentes corporações, contudo, ele pode ser generalizado em três passos: compreender o sistema atual, modificar o sistema atual e reavaliar o sistema modificado (PRESSMAN, 2005). Portanto, antes de realizar uma modificação, os desenvolvedores comumente necessitam explorar o código fonte e compreender quais partes do sistema são relevantes para então modificá-lo.

A razão pela qual um trecho de código é modificado nem sempre está associada à inclusão ou à remoção de requisitos em um sistema. De fato, um trecho de código pode ser simplesmente aperfeiçoado para que futuras implementações sejam realizadas de forma mais fácil ou ainda para que sua legibilidade seja melhorada. A refatoração (OPDYKE, 1992; FOWLER, 1999; MENS; TOURWÉ, 2004) é o processo de melhorar as estruturas internas de um sistema de software sem modificar seu comportamento externamente observável. Por meio de sucessivas transformações de código, chamadas padrões de refatoração, a refatoração visa melhorar incrementalmente a qualidade de software.

Frequentemente, os desenvolvedores recorrem a ferramentas de análise estática para auxiliá-los tanto na compreensão de código, quanto na identificação de defeitos ou de oportunidades de refatoração (FOWLER, 1999) durante o ciclo de desenvolvimento de uma aplicação. Ferramentas como FEAT (ROBILLARD; MURPHY, 2007) e ConcernMapper (ROBILLARD; WEIGAND-WARR, 2005), por exemplo, têm sido propostas para auxiliar na compreensão do código relacionado a um determinado interesse em um sistema. Outras ferramentas, tais como CheckStyle (BURN, 2012), FindBugs (HOVEMEYER; PUGH, 2004), PMD (COPELAND, 2005), JDeodorant (FOKAEFS; TSANTALIS; CHATZIGEORGIOU, 2007) e DECOR (MOHA et al., 2010), são ferramentas bem conhecidas na comunidade Java e visam identificar defeitos ou oportunidades de refatoração, tais como códigos duplicados, classes ou métodos longos, grandes listas de parâmetros em métodos, etc.

Um dos pontos críticos na definição de tais ferramentas está em sua capacidade de obter informações a respeito de código. As ferramentas de análise estática dependem, em geral, de uma representação intermediária de um programa para identificar situações que atendam

às condições necessárias descritas em seus algoritmos. Essa representação, em geral, está em forma de estruturas de dados como uma árvore de análise, uma árvore de sintaxe abstrata (AST) ou ainda em forma de código intermediário, o que leva implementadores dessas ferramentas a três desafios básicos (FEIJS; KRIKHAAR; VAN OMMERING, 1998): (i) extrair os elementos essenciais e as relações de um programa, necessários à operação da ferramenta, (ii) derivar novas relações por meio da combinação e da transformação das relações dadas e (iii) consultar por elementos e respectivas derivações para a realização da computação necessária.

Esse panorama se amplia com o uso de técnicas de modularização de interesses transversais, tais como a orientação a aspectos (OA) (KICZALES et al., 1997), na qual um código pode ser influenciado de forma sistêmica, por meio de mecanismos de composição de regras estáticas e dinâmicas. Características como transparência e quantificação (FILMAN; FRIEDMAN, 2000), comumente presentes em linguagens desse paradigma, possibilitam que um aspecto possa influenciar múltiplos interesses em diferentes abstrações, assim como uma única abstração pode ser influenciada por múltiplos aspectos. Pela natureza intrínseca de interesses transversais com as abstrações as quais eles afetam, técnicas mais rigorosas de análise são necessárias para compreender como aspectos influenciam um código base, bem como de que forma os aspectos são influenciados por outros aspectos.

Para auxiliar na resolução desses desafios, algumas linguagens de consulta em código fonte têm sido propostas (JANZEN; DE VOLDER, 2003; PFEIFFER; SARDOS; GURD, 2005; COHEN; GIL; MAMAN, 2006; HAJIYEV; VERBAERE; MOOR, 2006), sendo, em sua maioria, voltadas à busca em códigos estruturados ou orientados a objetos. Por meio de uma sintaxe própria, essas linguagens fornecem informações sobre os elementos e as relações obtidas a partir de um modelo reificado (FRIEDMAN; WAND, 1984), representando um programa.

O principal objetivo desta dissertação é a especificação e implementação de uma linguagem específica de domínio para consultas em código orientado a aspectos, denominada AQL (*Aspect Query Language*). Como resultado, espera-se que essa linguagem forneça informações a respeito de programas orientados a aspectos, seja para uso em ferramentas de análise estática ou ainda para utilização durante a fase de compreensão de um sistema de software. Neste contexto, as principais contribuições deste trabalho são:

- A definição de um modelo sintático para a linguagem de consulta AQL. O modelo sintático fornece um conjunto de instruções para a realização de consultas, envolvendo expressões com sintaxe próxima àquela das linguagens de consulta em objetos (*Object Query*

Languages - OQLs).

- Uma implementação de referência da linguagem AQL, baseada no modelo sintático definido. A AQL foi implementada com base em regras de transformação para a linguagem HQL (*Hibernate Query Language*) (BAUER; KING, 2005) para consultas de acordo com um metamodelo reificado de programas escritos em AspectJ (KICZALES et al., 2001).

- O desenvolvimento de uma ferramenta para identificação de oportunidades de refatoração em programas escritos em AspectJ, utilizando a linguagem AQL .

De forma a avaliar a linguagem proposta, um estudo de caso foi conduzido, no qual uma ferramenta de análise estática (*Aspect Tool for Refactoring* - ASTOR) foi construída para a identificação de oportunidades de refatoração em código orientado a aspectos, usando a linguagem AQL para consulta a elementos em programas AspectJ.

Esta dissertação está organizada da seguinte forma:

- O Capítulo 2, *Revisão de literatura*, apresenta a base conceitual que envolve o escopo desta dissertação. São descritos os seguintes assuntos: projeto de linguagens específicas de domínio, processo de desenvolvimento de linguagens específicas de domínio, linguagens de consulta em código, fundamentos de programação orientada a aspectos e o estado da arte em relação a linguagens de consulta em código orientado a aspectos.

- O Capítulo 3, *A Linguagem AQL*, apresenta a especificação sintática da linguagem AQL e exemplos de seu uso. Também são apresentados nesse capítulo, os requisitos mínimos de um programa orientado a aspectos para a realização de uma consulta na linguagem AQL.

- O Capítulo 4, *Implementação*, apresenta detalhes da linguagem AQL. São apresentados os seguintes assuntos: a arquitetura do framework de extração (AOPJungle), detalhes da implementação de referência de AQL, do mecanismo de transformação e das regras de transformação.

- O Capítulo 5, *Estudo de Caso*, apresenta um estudo de caso com a implementação da ferramenta ASTOR (*Aspect Tool for Refactoring*) com o uso de AQL, a qual permite identificar oportunidades de refatoração em código orientado a aspectos.

- O Capítulo 6, *Conclusão*, resume as contribuições desta dissertação e aponta algumas sugestões para trabalhos futuros.

2 REVISÃO DE LITERATURA

Este capítulo apresenta uma revisão da literatura referente ao seguintes assuntos: desenvolvimento de linguagens específicas de domínio, linguagens de consulta em código e fundamentos de programação orientada a aspectos. Além disso, este capítulo apresenta os trabalhados relevantes relacionados ao tema desta dissertação.

2.1 Linguagens Específicas de Domínio

Uma linguagem específica de domínio (DSL - *Domain Specific Language*) é uma linguagem de programação ou uma especificação executável de uma linguagem, projetada especificamente para problemas de um domínio em particular (DEURSEN; KLINT; VISSER, 2000). Como uma linguagem, uma DSL representa uma coleção de sentenças com uma sintaxe e uma semântica definidas. Historicamente, a construção de linguagens específicas de domínio não é um campo de estudo recente. De fato, o termo "Linguagem Específica de Domínio" pode ser encontrado na literatura com diferentes terminologias, tais como "Linguagens Orientadas a Aplicações" (SAMMET, 1972), como "Linguagens de Propósito Especial" (WEXELBLAT, 1981), como "Linguagens de Tarefas Específicas" (NARDI, 1993), como "Linguagens de Aplicação" (MARTIN, 1985) ou simplesmente como "Linguagens Especializadas" (BERGIN; GIBSON, 1996).

A diversidade de domínios para os quais DSLs têm sido criadas é bastante abrangente. Alguns domínios bem conhecidos envolvem compiladores, como por exemplo as linguagens LEX (LESK et al., 1990), YACC (JOHNSON, 1979), Bison (SYSTEM, 2012) e BNF (BACKUS, 1959), pesquisa e manipulação de repositório de dados, tais como SQL (*Structured Query Language*) (CHAMBERLIN et al., 1976) e XQuery (WALMSLEY, 2009), formatação de texto, como LaTeX (LAMPORT, 1994), e descrição de páginas web, como por exemplo HTML (W3C, 2012). Outros domínios pelos quais DSLs têm sido aplicadas estão multimídia (KAMIN; HYATT, 1997; ELLIOTT, 1999; FISHER; GRUBER, 2005), telecomunicações (KLARLUND; SCHWARTZBACH, 1999; LADD; RAMMING, 1994a) e projeto de hardware (JENNINGS; BEUSCHER, 2000).

O que tais linguagens têm em comum é a capacidade de atender a um propósito específico de forma mais eficiente quando comparadas a linguagens de propósito geral (GPLs - *General Purpose Languages*). O uso de conceitos e terminologias do domínio no projeto de

uma DSL aproxima o usuário desse domínio da especificação de um sistema computacional. Uma vez que o nível de abstração utilizado para modelar soluções está em um nível conhecido por esse usuário, a compreensão da intenção pela qual o sistema está sendo projetado tende a ser maior.

A construção de uma DSL pode envolver altos custos e um grande esforço de implementação (DEURSEN; KLINT; VISSER, 2000; WILE, 2001). Como alternativa, o uso de bibliotecas de subrotinas e frameworks orientados a objetos têm sido soluções mais frequentemente aplicadas para realizarem tarefas em um domínio bem conhecido. Em muitos casos, programas escritos em uma DSL são traduzidos para chamadas a essas subrotinas e uma DSL pode ser vista como uma forma de ocultar os detalhes dessa biblioteca.

Apesar da complexidade frequentemente associada à construção de DSLs, ferramentas como o Xtext (EYSHOLDT; BEHRENS, 2010) e o Meta-Programming System (MPS) (JETBRAINS, 2012) têm se mostrado promissoras para facilitar as atividades de especificação, de implementação e de testes de uma linguagem, tornando o desenvolvimento de DSLs cada vez mais atrativo.

2.2 Projeto de Linguagens Específicas de Domínio

O desenvolvimento de uma linguagem específica de domínio requer, fundamentalmente, o conhecimento em dois tópicos. Primeiro, o desenvolvedor necessita de conhecimento em construção de linguagens. Segundo, há a necessidade de conhecimento do domínio para o qual a linguagem está contextualizada.

Distinguir as características de uma DSL em relação a uma GPL é importante para diminuir os riscos durante seu projeto e implementação. Enquanto as linguagens de propósito geral, tais como Java, C e C#, oferecem uma grande quantidade de recursos para resolver problemas de qualquer espécie, uma DSL é uma linguagem projetada para um domínio em particular. Isso implica em conhecer em detalhes esse domínio e delinear um nível de abstração adequado que torne a DSL mais interessante quando comparada a uma GPL.

Ao projetar DSLs, um ponto importante a ser considerado é seu modo de especificação: desenvolver uma linguagem do zero ou integrar com uma linguagem já existente são abordagens que oferecem vantagens e desvantagens, conforme descritas a seguir.

2.2.1 Vantagens e Desvantagens

O uso e o desenvolvimento de uma DSL podem trazer riscos e oportunidades que devem ser levados em consideração antes de sua adoção (DEURSEN; KLINT; VISSER, 2000; MERNIK et al., 2003; FOWLER, 2010). A principal motivação para a criação de DSLs está na possibilidade de permitir que soluções sejam expressas no idioma e no nível de abstração do domínio do problema. Isso possibilita que especialistas do domínio possam ler programas escritos na DSL e compreendê-los mais facilmente, apontando, por exemplo, inconsistências na implementação de uma determinada regra de negócio. Nesse sentido, uma DSL adequadamente projetada fornece recursos para diminuir as diferenças entre o universo dos programadores e dos usuários, melhorando a comunicação entre eles (FOWLER, 2010). Em última instância, especialistas no domínio, sem conhecimentos específicos de programação, poderiam escrever programas usando a DSL para o contexto para o qual ela foi projetada (HUDAK, 1998; FOWLER, 2010).

Trabalhar em um nível de abstração mais elevado oferece alguns benefícios, tais como: (i) a diminuição da carga de detalhes técnicos em domínios nos quais uma computação intensiva for necessária, (ii) a melhoria no nível de documentação de código em relação a uma linguagem de propósito geral, visto que o conhecimento e as terminologias do domínio são, em geral, parte da notação utilizada na linguagem e (iii) a escrita de programas mais concisos, facilitando sua leitura e compreensão. Esses benefícios não somente permitem facilitar a escrita de programas, como possibilitam também outras vantagens, conforme observadas em diversos estudos (LADD; RAMMING, 1994b; KIEBURTZ et al., 1996; DEURSEN; KLINT, 1997; RÉVEILLÈRE et al., 2000; DEURSEN; KLINT; VISSER, 2000), tais como aumento da produtividade, maior portabilidade e manutenibilidade e, validação e otimização em nível de domínio.

As desvantagens em relação às DSLs podem ser classificadas sob três aspectos: financeiros, técnicos e comportamentais. Em relação aos riscos financeiros, os custos relativos ao projeto, à implementação e à manutenção de uma DSL são maiores quando comparados somente à implementação de sua biblioteca ou modelo que irá apoiá-la. (DEURSEN; KLINT; VISSER, 2000; FOWLER, 2010). Aliados a esse fator, os programadores devem ter conhecimento tanto do domínio quanto do projeto de linguagens computacionais, o que pode acarretar em custos maiores, devido ao uso de mão de obra mais especializada. Além disso, podem ha-

ver custos envolvidos no treinamento dos usuários para o uso da nova linguagem (SPINELLIS, 2001) .

Em relação ao riscos técnicos, uma DSL pode apresentar uma perda de eficiência em relação ao código escrito em uma GPL (HUDAK, 1998; DEURSEN; KLINT; VISSER, 2000; SPINELLIS, 2001). Esse cenário pode ser observado quando seções de código são traduzidas diretamente a chamadas de funções em uma biblioteca, sem análise da intenção do programador no domínio, podendo gerar um código ineficiente. Por outro lado, o desenvolvimento em um nível de abstração mais elevado permite que a intenção do programador seja mais claramente declarada. Juntamente com o conhecimento do domínio em questão, o compilador da DSL pode ser otimizado para gerar um código mais eficiente. Outros fatores de riscos técnicos a serem observados são o balanceamento de recursos disponíveis na DSL (DEURSEN; KLINT; VISSER, 2000) e sua integração no processo de desenvolvimento de software (SPINELLIS, 2001).

Por fim, o uso de uma DSL pode requerer mudanças comportamentais por parte da equipe de desenvolvimento, as quais podem conduzir até mesmo à sua rejeição. Um dos problemas em relação à adoção de uma DSL refere-se ao acréscimo de mais uma linguagem no processo de desenvolvimento (FOWLER, 2010). A adoção de uma DSL no processo de desenvolvimento de software implica em aprendizado, o que pode levar alguns usuários a considerarem um aumento do esforço, com mais pessoas envolvidas. Paradoxalmente, as DSLs também podem apresentar riscos devido à falta de recursos necessários à sua manutenção e à sua evolução (FOWLER, 2010). Quando uma organização decide pela criação de uma DSL para apoiar o desenvolvimento de uma solução, deve levar em consideração que a manutenção dessa linguagem deve ocorrer de acordo com as evoluções tecnológicas e do conhecimento, as quais a linguagem está fundamentada.

2.2.2 Uma Taxonomia para DSLs

As DSLs, assim como as linguagens de propósito geral, podem ser categorizadas sob múltiplos aspectos, o que torna particularmente complexa a realização de uma classificação mais abrangente. Contudo, para um projetista, é revelante distinguir as famílias de DSLs sob o ponto de vista de construção, para que as decisões sobre as diferentes abordagens de projeto possam ser melhores fundamentadas. As DSLs podem ser categorizadas sob três diferentes dimensões (MERNIK et al., 2003; GHOSH, 2010): origem, aparência e implementação. A

Figura 2.1 ilustra uma taxonomia para DSLs.

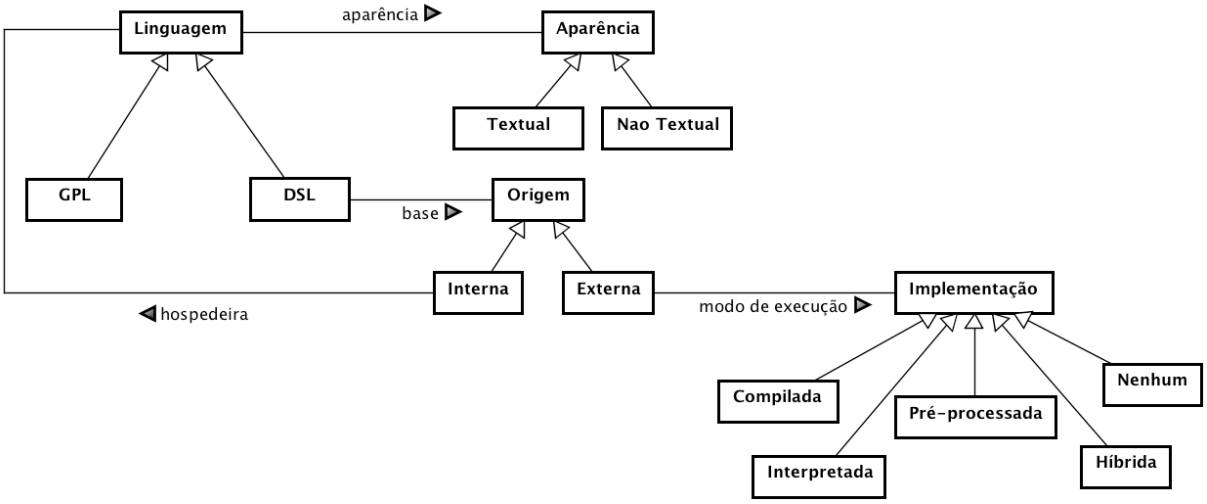


Figura 2.1 – Uma taxonomia para linguagens específicas de domínio

Uma DSL pode ser classificada, em relação à sua origem, como interna ou externa. Uma DSL interna é uma linguagem projetada a partir de regras sintáticas e semânticas de uma linguagem já estabelecida. Dessa forma, uma DSL interna delega as atividades de verificação léxica, sintática, semântica e transformação de código ao compilador da linguagem hospedeira. Uma linguagem hospedeira pode tanto ser outra DSL quanto uma GPL, embora GPLs sejam mais comumente empregadas para esse fim.

Em geral, qualquer linguagem de propósito geral pode ser utilizada como uma linguagem hospedeira, porém algumas delas possuem propriedades mais apropriadas para essa função, como por exemplo, as linguagens Ruby (CUNNINGHAM, 2008), Python (PAUL, 2005), Haskell (HUDAK, 1996), Java (KABANOV; RAUDJÄRV, 2008) e Boo (RAHIEN, 2008). O trecho de código a seguir ilustra o uso de uma linguagem interna, tendo a linguagem Java como GPL hospedeira:

```
binder.bind(Service.class).
        to(ServiceImpl.class).
        in(Scopes.SINGLETON)
```

O exemplo anterior ilustra o uso da API Google Guice[1] (VANBRABANT, 2008), no qual uma implementação (`to(ServiceImpl.class)`) é associada a um tipo (`bind(Service.class)`) usando o padrão de instanciação *Singleton* (`in(Scopes.SINGLETON)`). O trecho de código traduz-se em uma sequência encadeada de chamadas aos métodos da API resultando em uma frase coerente dentro do contexto (neste caso, em língua inglesa). Como

[1] A API Google Guice é um framework utilizado para injeção de dependência

resultado, uma DSL interna deve fornecer formas para a montagem de sentenças completas, que tenham propósito no domínio do usuário da linguagem. Por reunir nomes coerentes, que quando em conjunto, formam uma sentença que fluem com significado para o usuário de um contexto, as DSLs internas também são chamadas de interfaces fluentes (FOWLER, 2010).

Uma DSL externa é uma linguagem com sua própria sintaxe, a qual depende de uma infraestrutura separada para análise léxica, sintática, semântica, interpretação, compilação, otimização e geração de código. Comparada a uma GPL, uma DSL externa possui requisitos similares para sua construção, porém seu conjunto de recursos se limita ao contexto para o qual ela foi projetada.

Em relação à aparência, uma linguagem específica de domínio pode ser classificada como textual ou não textual. No formato textual, as DSLs permitem expressar o domínio através de caracteres, que são combinados em palavras, expressões e instruções seguindo as regras gramaticais da linguagem.

As linguagens não textuais são linguagens que permitem ao usuário expressar conhecimento de domínio por meio de modelos visuais, com o auxílio de símbolos gráficos além de elementos textuais. Esse modo de expressar conhecimento tem resultado em conceitos como a programação intencional (WA; SIMONYI; SIMONYI, 1995), na qual, por meio de editores de projeção, os especialistas do domínio podem especificar uma visão apropriada de seu conhecimento utilizando elementos representativos em sua especialidade, tais como tabelas, figuras, conectores, nomenclaturas específicas e fórmulas.

As DSLs podem ser implementadas seguindo diferentes modelos de execução (MERNIK et al., 2003). De acordo com suas características, um programa escrito em uma DSL pode ser relacionado como uma especificação, uma definição ou uma descrição. Por exemplo, a linguagem BFN é uma DSL com um caráter puramente declarativo que pode também servir como uma entrada para um gerador de analisador léxico/sintático. Segundo MERNIK (2003), quatro graus de execução em relação à classificação de DSLs são identificados: (i) semântica de execução bem definida: são linguagens com definições claras para sua execução (por exemplo, *Excel Macro Language* (ROMAN, 2008)), (ii) entrada para um gerador de aplicação: são linguagens executáveis, porém com características mais declarativas e com menor grau de definição de execução semântica; o gerador de aplicação em questão é um compilador para a DSL, (iii) não executáveis e úteis para geradores de aplicação: são linguagens que embora não sejam executáveis, são úteis como parâmetro de entrada para outra aplicação. A linguagem BNF,

citada acima, é um exemplo dessa categoria de linguagem, e (iv) DSLs não concebidas para serem executadas: são linguagens sem propósito de serem executadas, como por exemplo uma estrutura de dados de domínio específico (WILE, 2001).

2.3 Processo de Desenvolvimento de DSLs

O processo para definição e desenvolvimento de DSLs é composto pelas seguintes fases: decisão, análise, projeto, implementação, validação, instalação e manutenção (MERNIK et al., 2003; BRYANT et al., 2010). Cada fase do processo, ocorre, em geral, de forma interpolada. A fase de decisão pode ser influenciada por uma análise preliminar, que pode fornecer informações para questões não previstas na fase de projeto, a qual pode ser influenciada por decisões de implementação. A maioria das fases de construção de uma DSL é apoiada por um conjunto de padrões de projeto (SPINELLIS, 2001; MERNIK et al., 2003; AKKI et al., 2012). Tais padrões fornecem soluções para situações comuns, as quais projetistas podem potencialmente encontrar durante a execução do projeto, e que já foram aplicadas com sucesso no desenvolvimento de DSLs no passado. A Figura 2.2 ilustra as fases de desenvolvimento de uma DSL, explicadas a seguir.

2.3.1 Decisão

A fase de decisão deve levar em consideração o domínio, o tamanho da comunidade que utilizará a DSL, as ferramentas disponíveis para seu desenvolvimento (se for o caso), além dos custos envolvidos para sua implementação. Em MERNIK (2003), nove padrões para auxiliar na tomada de decisão em relação à construção de uma DSL são recomendados: (i) *Notation.* adiciona notações (novas ou existentes) do domínio, (ii) AVOPT (*Domain-Specific Analysis, Verification, Optimization, Parallelization and Transformation*): consiste em um padrão aplicado no qual as atividades de análise, verificação, otimização, paralelização e transformação são extremamente complexas quando realizadas com uma linguagem de propósito geral, (iii) *Task Automation.* elimina tarefas repetitivas, (iv) *Product Line.* especifica membros de uma linha de produto de software, (v) *Data Structure Representation.* facilita a descrição de dados, (vi) *Data Structure Traversal.* facilita o exame de estrutura de dados complexas, (vii) *System front-end.* facilita a configuração do sistema (viii) *Interaction.* permite interações programáveis e (ix) *GUI construction.* facilita a construção de interfaces gráficas com o usuário.

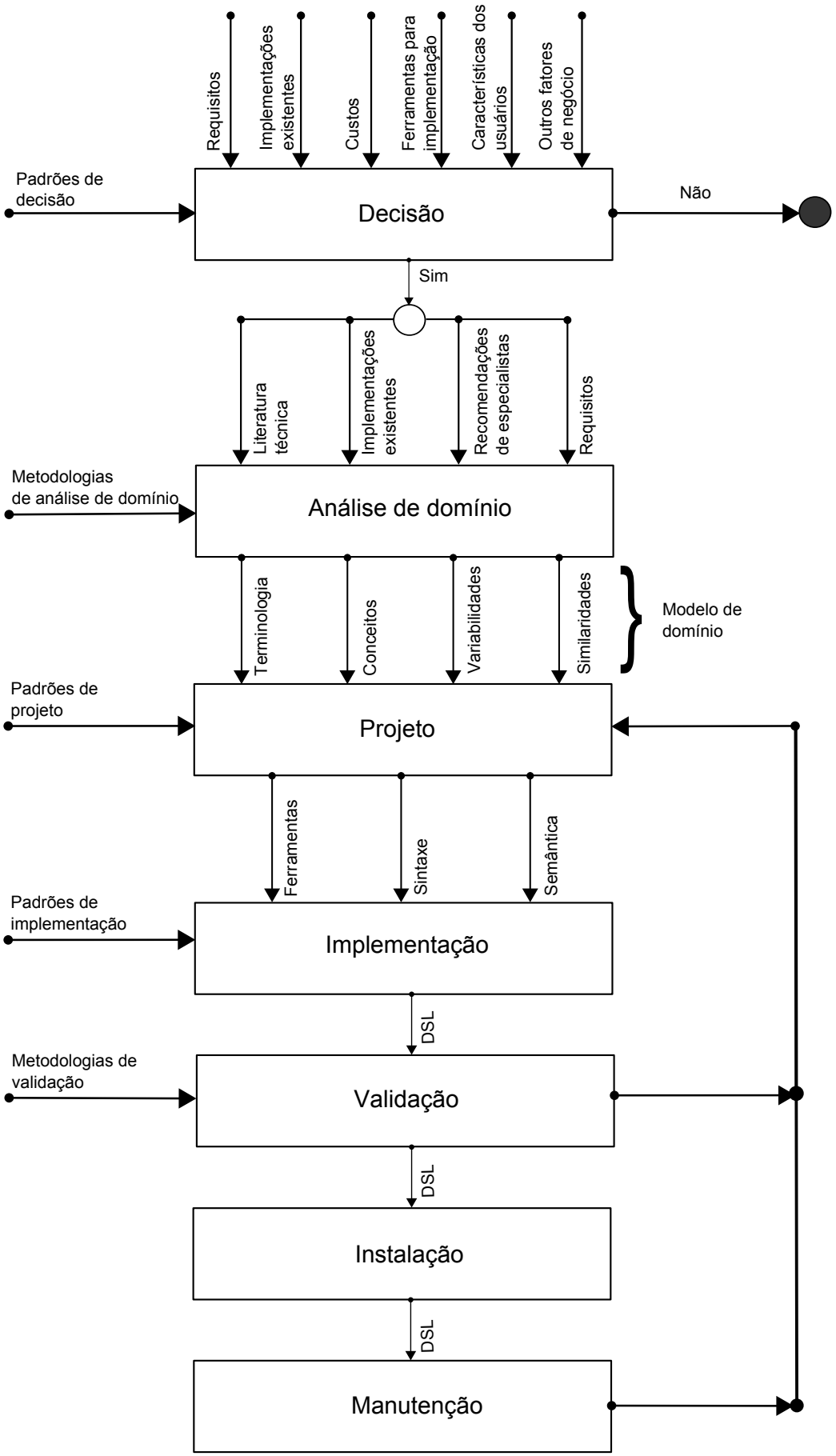


Figura 2.2 – Fases de desenvolvimento de uma DSL

2.3.2 Análise

Uma vez que a decisão de projetar uma DSL tenha sido tomada, o próximo passo é realizar uma análise de domínio. As entradas para essa fase são as várias fontes implícitas e explícitas de conhecimento, tais como documentos, manuais, pesquisas com usuários e entrevistas com especialistas de domínio. Como saída, essa fase gera um modelo de domínio consistindo de: (i) uma definição do domínio, delimitando o escopo do mesmo, (ii) a terminologia do domínio (vocabulário, ontologia, etc.), (iii) descrições do conceitos do domínio, (iv) modelos de características descrevendo similaridades e variabilidades dos conceitos do domínio e suas interdependências (MERNIK et al., 2003). Metodologias como FAST (WEISS; LAI, 1999), *Organization Domain Model* (ODM) (SIMOS, 1995), *Domain-Specific Software Architecture* (DSSA) (TAYLOR; TRACZ; COGLIANESE, 1995) ou *Feature-Oriented Domain Analysis* (FODA) (KANG et al., 1990) são comumente utilizadas para apoiar a análise de domínio formal. Outro método que pode ser empregado é a extração de conhecimento a partir de código escrito em uma GPL, o qual pode ter partes refatoradas para uma DSL.

2.3.3 Projeto

A fase de projeto leva em consideração duas abordagens ortogonais, as quais descrevem a natureza da especificação do projeto e o relacionamento com linguagens existentes. Em relação à sua natureza, a especificação pode ocorrer de modo formal ou informal. Um projeto formal implica em escrever uma especificação utilizando uma notação rígida, como, por exemplo, expressões regulares e gramáticas para a definição de sintaxe e semântica operacional, gramática de atributos ou notação denotacional para a especificação semântica. Informalmente, a especificação do projeto é realizada por meio de linguagem natural, incluindo um conjunto de programas ilustrativos escritos na DSL proposta. O relacionamento com linguagens existentes define se a DSL se integra a uma linguagem existente ou se constitui uma criação isolada. A integração entre uma DSL e uma linguagem existente pode ser realizada, segundo Spinellis (2001) e Mernik (2003), de diferentes maneiras: utilizando a abordagem *Piggyback*, a abordagem *Specialization* ou a abordagem *Extension*. As duas primeiras abordagens mantêm a linguagem base sem modificações, enquanto a abordagem *Extension* cria uma linguagem derivada, conforme descrito a seguir.

2.3.3.1 Piggyback

O padrão *Piggyback* é um padrão estrutural que utiliza parcialmente os recursos de uma linguagem existente, denominada linguagem base, para criação de uma nova DSL. Esse padrão é utilizado sempre que a DSL compartilha elementos comuns de uma linguagem existente, sem alterar nenhuma de suas características. Em uma abordagem compilada, o compilador traduz os elementos linguísticos da DSL em código da linguagem base para que sejam, então, compilados e executados. Se a DSL for implementada como um interpretador, a mesma estratégia pode ser aplicada, caso a linguagem base tenha recursos para facilitar a chamada a seu interpretador com os argumentos do interpretador da DSL. Um exemplo de utilização dessa estratégia são os recursos de macro da linguagem C, que traduzem instruções da linguagem de domínio para a linguagem C (base).

2.3.3.2 Specialization

No padrão *Specialization* o compilador ou interpretador de uma linguagem base tem seus recursos restringidos até atender aos requisitos da DSL. Um caso particular de aplicação desse padrão emerge quando aspectos de segurança requeridos por uma DSL somente podem ser atendidos quando características não seguras de uma linguagem base são removidas, tais como alocação dinâmica de memória, *threads* ou ponteiros sem ligação. À medida que a DSL torna-se um subconjunto da linguagem base, o processo de redução da sintaxe e da semântica pode ser garantido por meio de um processador que verifica a conformidade com a DSL. Exemplos de linguagens que utilizam esse padrão são JavaLight (NIPKOW; VON OHEIMB, 1998) e uma versão da linguagem Pascal destinada ao seu ensino (SAVITCH, 1986).

2.3.3.3 Extension

O padrão *Extension* implica na adição de uma sintaxe e de uma semântica que permitem novas alternativas em regras existentes na linguagem base. Em geral, a DSL herda todas as funcionalidades da linguagem base, enquanto adiciona novas características, tais como novos tipos de dados, novas estruturas de controle, elementos semânticos e simplificações sintáticas (*syntax sugar*). Em linguagens compiladas, as técnicas de extensão são frequentemente empregadas por meio de pré-processadores que traduzem um programa escrito na DSL para a linguagem base (SPINELLIS, 2001). Alternativamente, traduções fonte-a-fonte (FAITH; NYLAND; PRINS,

1997), composição de código (STICHNOTH; GROSS, 1997) ou programação intencional são métodos que permitem estender uma linguagem usando operadores de alto nível. Um exemplo desse tipo de padrão é utilizado em LINQ (CORPORATION, 2008), na qual instruções SQL podem ser utilizadas como expressões dentro de uma linguagem de propósito geral existente.

Outra abordagem para o desenvolvimento de uma DSL é projetá-la a partir do zero, sem nenhuma integração com uma linguagem existente. Nesse sentido, o processo de desenvolvimento de uma DSL se assemelha bastante ao desenvolvimento de uma GPL, sendo necessária a formalização de uma gramática (geralmente realizada por meio de notação *Backus-Naur Form* (BNF) ou *Extended Backus-Naur Form* (EBNF)), de uma semântica, de um processo de transformação (caso tenha) e de geração de código alvo (código de máquina, código intermediário ou ainda um outro código fonte).

2.3.4 Implementação

Quando uma DSL executável é projetada, diferentes abordagens podem ser empregadas durante sua implementação, dependendo do modelo de execução escolhido. Linguagens compiladas e interpretadas são as abordagens comumente utilizadas na implementação de uma DSL. Contudo, abordagens como pré-processadores, extensão de compiladores/interpretadores, *Commercial Off-The-Shelf* (COTS) e implementações internas podem também ser empregadas. A seguir, cada abordagem é resumidamente descrita:

- **Compilador.** É uma abordagem de transformação fonte-a-fonte, na qual o código é analisado e transformado para uma representação intermediária, como por exemplo uma árvore de sintaxe abstrata (AST). Em um segundo momento, essa representação interna então é transformada para um código executável específico para a uma determinada plataforma.

- **Interpretador.** É uma abordagem na qual as construções da DSL são carregadas, decodificadas e executadas. Para cada instrução, as expressões são validadas no contexto de execução atual do interpretador.

- **Pré-processador.** Nessa abordagem, a transformação de fonte-a-fonte ocorre antes do processamento da linguagem em si. Esse recurso pode ser utilizado para expandir uma linguagem de programação, apesar de oferecer suporte limitado à verificação estática (MERNIK et al., 2003).

- **Interna.** Nessa abordagem, as construções da DSL são adaptadas em uma GPL existente, por meio da definição de novos tipos abstratos de dados e operadores. As bibliotecas de aplicação são uma forma básica de implementação interna.

- **Compilador / Interpretador extensível.** Nessa abordagem, um compilador ou interpretador é estendido com regras de otimização e regras específicas do domínio para geração de código.

- **Commercial Off-The-Shelf (COTS).** A implementação de uma DSL via COTS envolve utilizar notações e/ou ferramentas existentes e previamente testadas para um domínio específico.

- **Híbrida.** É uma abordagem que emprega dois ou mais mecanismos acima mencionados para a realização de transformações.

Além dessas abordagens, uma DSL pode ainda não necessitar nenhuma transformação de código, tendo assim, um caráter puramente informativo. Em geral, essas DSLs são empregadas para expressar um conhecimento relevante do domínio.

2.3.5 Validação

A fase de validação permite avaliar a DSL em relação aos seus requisitos. As abordagens para a validação de uma DSL podem empregar experimentos de naturezas quantitativas e/ou qualitativas (GABRIEL; GOULÃO; AMARAL, 2011). Um experimento de natureza quantitativa utiliza propriedades mensuráveis a partir de dados reais, com o objetivo de confirmar ou refutar a hipótese do testador. Por exemplo, em uma linguagem fortemente tipada, espera-se que a falta da declaração de um tipo antes de seu uso implique em um erro de compilação.

Um experimento de natureza qualitativa é baseado na qualidade dos dados obtidos por meio de observações, entrevistas, questionários, etc. de uma população em específico. A facilidade de uso de uma linguagem é determinada por meio de testes de usabilidade, os quais são conduzidos junto aos usuários finais. Além disso, testes de usabilidade são úteis para reduzir o número de funcionalidades redundantes ou mesmo desnecessárias antes da entrega da versão final de uma linguagem (GABRIEL; GOULÃO; AMARAL, 2011; AKKI et al., 2012).

2.3.6 Instalação

A fase de instalação consiste em um conjunto de atividades inter-relacionadas que devem garantir à DSL um método para identificação de versões, sua configuração e ativação no ambiente do usuário (caso seja executada localmente), sua atualização, em caso de uma sobreposição de versões e ainda procedimentos para sua desinstalação e remoção do ambiente de produção.

2.3.7 Manutenção

A fase de manutenção é a fase pela qual a linguagem passa por um processo de atualização para refletir novos requisitos. Uma DSL pode evoluir por necessidade de novos requisitos advindos de uma modificação nas regras do seu domínio de atuação, por solicitação dos usuários, por modificações na sua plataforma de atuação (caso seja dependente da mesma) ou ainda por adequação à linguagem hospedeira, caso seja uma DSL interna. Como um programa, uma DSL pode ainda requerer modificações para corrigir falhas, melhorar seu desempenho ou ainda outros atributos de qualidade.

2.4 Linguagens de Consulta em Código

A consulta em código de programas constitui uma tecnologia para as atividades de reengenharia de software (KULLBACH et al., 1999). Seja na atividade de compreensão de código ou durante as atividades de refatoração, as linguagens de consulta em código fornecem recursos para que desenvolvedores recuperem informações, coletem métricas e naveguem pelas estruturas de um programa.

As abordagens para a realização de consulta em código variam desde ferramentas de navegação por elementos de programas, até consultas a um repositório de dados com informações de um programa. Nessa última abordagem, uma linguagem de consulta ao repositório pode ser utilizada, como por exemplo SQL para um banco relacional ou XQuery para um repositório XML, ou ainda empregar o uso de DSLs, particularmente projetadas para esse fim.

2.4.1 Estratégias de Implementação de Consulta em Código

A implementação de técnicas de consulta em código tem sido abordada de diferentes maneiras, de acordo com seus objetivos primários, tais como escalabilidade, reuso, facilidade de integração com ambientes de desenvolvimento, escopo de busca e expressividade da linguagem. Percebe-se, contudo, que na maioria dos trabalhos apresentados (CLEMENT, 2003; PFEIFFER; SARDOS; GURD, 2005; BARRENECHEA, 2007), variações do framework desenvolvido por Feijs e Van Ommering (FEIJS; KRIKHAAR; VAN OMMERING, 1998) têm sido aplicadas para o desenvolvimento de linguagens e ferramentas de consulta em código.

Tal framework define três etapas para a obtenção de resultados para a análise em artefatos de software: extração, abstração e apresentação. Cada uma dessas etapas é realizada por um conjunto de ferramentas que podem ser genéricas ou específicas para a linguagem a ser analisada. As seções seguintes descrevem cada uma das etapas e apontam algumas alternativas para sua implementação.

2.4.1.1 Extração

A atividade de extração é responsável por coletar, selecionar e mapear os elementos da linguagem de forma que esses possam ser transformados em uma base de conhecimento sobre um sistema. Adicionalmente, a extração pode ainda coletar outras informações relativas ao fluxo de execução normal e excepcional de um programa, além de padrões no uso local (métodos e comportamentos transversais) de variáveis, a fim de fornecer subsídios para análise de fluxo de dados. A sofisticação em relação ao mecanismo de extração é proporcional à abrangência da linguagem de busca, podendo envolver elementos estruturais, relações e controle de fluxo e de dados (COHEN; GIL; MAMAN, 2006). As atividades de coleta, seleção e mapeamento são descritas da seguinte forma:

- **Coleta.** A coleta é responsável por selecionar uma origem e então realizar a varredura dos elementos de um programa para serem então selecionados. A origem para a atividade de coleta pode ser o próprio código fonte e a documentação de um programa ou alguma representação intermediária, como por exemplo, sua AST ou *bytecode* no caso de uma implementação para a máquina virtual Java. Em geral, a coleta por meio de uma representação intermediária fornece mais subsídios para o mapeamento de alguns relacionamentos em comparação à coleta diretamente em código fonte, pela qual seria exigida

maior computação.

- **Seleção.** A seleção é a atividade responsável por filtrar os elementos coletados para que sejam então mapeados em suas relações, de acordo com a semântica da linguagem. Dependendo do escopo de pesquisa que a linguagem de busca implementa, mais elementos poderão ser selecionados durante essa fase. O processo de seleção é especialmente ligado à implementação de uma determinada linguagem, sendo necessário conhecer suas regras sintáticas e semânticas para que os filtros sejam corretamente aplicados.

- **Mapeamento.** A atividade de mapeamento visa organizar os elementos coletados e selecionados para que possam ser posteriormente consultados. A representação dos relacionamentos entre elementos de uma linguagem tem sido apresentada de diferentes modos. Dentre os mais comuns encontrados na literatura, destacam-se a álgebra relacional de Codd (CODD, 1972), o cálculo relacional de Tarski (TARSKI, 1941) e a programação lógica (STERLING; SHAPIRO, 1986). Outros modelos de representação do código também podem serem vistos, como por exemplo, uma estrutura orientada a objetos, uma estrutura de árvore ou linguagens de grafo, tais como grUML (EBERT et al., 2007).

A extração é um processo que pode ocorrer em diferentes momentos, dependendo de como a linguagem de consulta é projetada e quais são seus objetivos, principalmente em termos de escalabilidade e desempenho. A extração pode ocorrer sob demanda, ou seja, a obtenção dos elementos necessários para a consulta é realizada de acordo com a solicitação feita pela linguagem de busca ou ainda utilizar um modelo reificado para reter informações do programa, como uma fase de preparação. Na abordagem sob demanda, a coleta, a seleção e o mapeamento são realizados de acordo com a necessidade, ou seja, a consulta é analisada, e os elementos necessários para sua resolução são coletados e então mapeados. Essa operação é realizada sempre que uma nova consulta for realizada. Na abordagem utilizando um modelo reificado, os elementos de um programa são selecionados e mapeados para um modelo que representa os elementos da linguagem e seus relacionamentos. Esse modelo pode ser então mantido em memória para consulta ou ainda ser persistido para um repositório de dados como por exemplo um banco de dados orientado a objetos, um banco de dados relacional, um banco de fatos, um repositório XML, etc. Uma abordagem híbrida também pode ser considerada, onde a reificação dos elementos do programa é realizada parcialmente e, de acordo com a consulta, mais informações são obtidas.

2.4.1.2 Abstração

O processo de abstração (cálculo) é a fase na qual a consulta e a análise ocorrem. Nessa fase, novos relacionamentos são derivados e informações adicionais sobre um programa são computadas a partir dos relacionamentos primitivos obtidos na fase de extração. Por exemplo, uma classe *C2* sendo um subtipo de *C1* e a classe *C3* um subtipo de *C2*, pela regra de transitividade de tipos de OO, é possível derivar que o tipo *C3* também é um subtipo de *C1*.

2.4.1.3 Apresentação

Apresentação é o processo de expor o resultado de uma consulta. Muitas ferramentas de consulta em código fornecem, junto à linguagem de busca, formas de apresentação dos resultados. Outras, somente entregam o resultado para que seja então tratado por outras camadas de software. Em geral, ferramentas que auxiliam no desenvolvimento do código e se integram diretamente às IDEs fornecem todas as atividades de consulta, desde a extração até a apresentação.

2.5 Programação Orientada a Aspectos

A programação orientada a aspectos é um paradigma que permite a modularização de interesses transversais por meio de mecanismos de abstração e de combinação que apoiam a clara separação e composição de interesses que se encontrariam, de outra maneira, fragmentados de forma sistêmica ao longo de uma aplicação.

Na abordagem orientada a aspectos, a separação de interesses transversais pode ser auxiliada por meio de entidades de primeira classe denominadas aspectos. Um aspecto é uma unidade de modularização que permite aos programadores definirem elementos que identifiquem pontos de execução em um programa (em classes ou outros aspectos) e que descrevam o código a ser executado sempre que esses pontos de execução sejam alcançados. Além disso, um aspecto também fornece construções que permitem modificar a estrutura estática de um programa. Essas construções possibilitam a inclusão de novos membros, como métodos e atributos, além de permitirem a reestruturação de hierarquia de tipos e implementação de novas interfaces.

Um típico processo de desenvolvimento envolvendo orientação a aspectos implica nos seguintes passos (SOARES, 2004): (i) o desenvolvedor deve identificar os interesses a serem

implementados a partir dos requisitos de software, (ii) um conjunto de componentes é escrito em uma linguagem (código base), como por exemplo Java, para implementar os requisitos funcionais da aplicação, resultando em uma decomposição primária. Em contrapartida, um conjunto de interesses transversais relacionado às propriedades que afetam o sistema de forma geral é implementado como aspectos, (iii) o terceiro passo é responsável por compor os interesses em um processo denominado combinação (KICZALES et al., 1997) que é realizado por um combinador da linguagem orientada a aspectos. O processo de combinação pode ser realizado de forma estática – manipulando a estrutura do código que implementa componentes (código base) – ou dinamicamente – usando a habilidade de refletir o estado de execução do programa para condicionalmente modificá-lo. Não há regras rígidas quanto ao momento em que o processo de combinação é executado, podendo ocorrer em tempo de pré-processamento, de compilação, de carga ou de execução.

A linguagem AspectJ é uma linguagem de programação de propósito geral, que estende a linguagem Java, e fornece construções para a implementação de programas orientados a aspectos. A principal abstração de AspectJ é o aspecto, uma unidade que encapsula elementos que permitem afetar diferentes classes e mesmo outros aspectos de forma estática ou dinâmica.

Um aspecto é estruturalmente similar a uma classe, ou seja, pode conter métodos, atributos e ser definido em uma estrutura hierárquica, por meio de aspectos especializados. Estaticamente, além da introdução de membros e modificações hierárquicas em tipos complexos definidos em Java, AspectJ oferece abstrações para suavizar exceções[2] e inserir anotações em elementos de um programa. De forma dinâmica, um aspecto pode influenciar no comportamento de um programa modificando a forma como o mesmo é executado.

Um aspecto pode selecionar pontos de execução bem definidos em um programa, chamados pontos de junção (*joinpoints*), como por exemplo chamadas ou execuções de métodos, a execução de construtores, a leitura e a escrita de campos e a execução de tratadores de exceções.

Por exemplo, a classe *Circle*, descrita na listagem 2.1, contém um método chamado *area()*, representando a operação de cálculo da área de um círculo, e um campo chamado *radius* representando o raio do círculo. Alguns pontos de junção identificados nessa classe seriam a chamada ao método *area* (linha 11), a execução do método *area* (linha 5), a escrita e leitura do campo *radius* (linhas 10 e 5, respectivamente).

[2] A suavização de exceções é um mecanismo aplicado à linguagem Java, que permite modificar exceções verificadas para não verificadas, a fim de modularizar o tratamento de exceções em aspectos.

```
1  public class Circle extends Figure {
2     private double radius;
3
4     public double area() {
5        return Math.PI * Math.pow(radius, 2);
6     }
7
8     public static void main(String[] args) {
9        Circle c = new Circle();
10       c.radius = 10;
11       double circleArea = c.area();
12       System.out.println(circleArea);
13    }
14 }
```

Listagem 2.1 – Ilustração de pontos de junção em um código Java

A seleção de pontos de junção é realizada por meio de de uma linguagem declarativa específica, descrita em construções denominadas pontos de corte (*pointcuts*). A declaração de um ponto de corte define um predicado, no qual, sempre que satisfeito, causa a execução dos adendos associados a esse ponto de corte. A seleção de pontos de junção pode ser definida com o uso de operadores lógicos *and*, *or* e *not* (&&, || e ! respectivamente).

Os adendos (*advices*) são ações associadas a um ponto de corte. Eles são definidos para serem executados antes, após ou no entorno de um ponto de junção selecionado. Em AspectJ, a definição do momento de execução de um adendo é realizada por diferentes palavras-chave (*before*, *after*, *around*). Um adendo executado após o ponto de junção selecionado possui ainda duas variações: ele pode ser executado após o sucesso da execução do ponto de junção selecionado ou após a sinalização de uma exceção.

A listagem 2.2 ilustra a declaração de um aspecto em AspectJ. As linhas 2, 3, 4 e 5 definem pontos de corte com critérios para seleção de pontos de junção de leitura, escrita, chamada de método e execução de método, respectivamente. Na linha 7 é definido um adendo do tipo *before* associado com os pontos de corte *getRadius* e *setRadius*. Na linha 11 é definido outro adendo, do tipo *after returning*, associado aos pontos de corte *callArea* e *executionArea*.

Outros exemplos de pontos de junção expostos em AspectJ são: a chamada e execução de construtores, a execução de um adendo, a inicialização de uma classe e a inicialização e a pré-inicialização de um objeto.

```
1  public aspect CircleAspect {
2     pointcut getRadius() : get (double Circle.radius);
3     pointcut setRadius() : set (double Circle.radius);
4     pointcut callArea() : call (* Circle.area());
5     pointcut executionArea() : execution (* Circle.area());
6
7     before () : getRadius() || setRadius() {
8        System.out.println(thisJoinPointStaticPart.toLongString());
9     }
10
11    after() returning : callArea() || executionArea() {
12       System.out.println(thisJoinPointStaticPart.toLongString());
13    }
14 }
```

Listagem 2.2 – Exemplo de definição de um aspecto em AspectJ

2.6 Trabalhados Relacionados

As linguagens de consulta em código têm sido extensivamente estudadas durante a última década, principalmente no contexto OO, para o qual ferramentas e linguagens têm sido propostas. Tais ferramentas são relevantes no estudo de tecnologia de busca em código orientado a aspectos pois, em grande parte, possuem funcionalidades reutilizáveis, visto a própria característica de linguagens como AspectJ como uma extensão da linguagem Java. Esta seção foca na apresentação do estado da arte em linguagens de consulta em código orientado a aspectos e cita alguns trabalhos relevantes no contexto de orientação a objetos.

2.6.1 ActiveAspect

A ferramenta *ActiveAspect* (COELHO; MURPHY, 2005) é apresentada como um instrumento de apresentação de informações sobre interesses transversais por meio de diagramas de estruturas. A ferramenta foi escrita em AspectJ, como um *plug-in* para a plataforma Eclipse. O modelo de visualização, denominado modelo ativo, é uma estrutura de dados que representa a organização de interesses transversais de um programa com o propósito de mostrá-la por meio da construção de visões interativas para o programador. Uma vez que a visão é construída, os usuários podem interagir para revelar novos relacionamentos, código fonte relativo ao interesse ou, ainda, informações de um determinado elemento.

O ponto inicial de construção de uma visão é a operação de projeção, na qual o conteúdo do modelo é produzido, por meio da seleção de partes da estrutura do programa que são potencialmente de interesse do programador. Essas partes são determinadas por meio de um

conjunto de heurísticas baseadas em propriedades estruturais e nas características do interesse, como por exemplo, o tipo de elemento (classe ou membro), o tipo de membro (campo, método ou construtor), a visibilidade do membro (pública, protegida, privada, etc), etc. A Figura 2.3 ilustra a visão de um modelo construído após a operação de projeção.

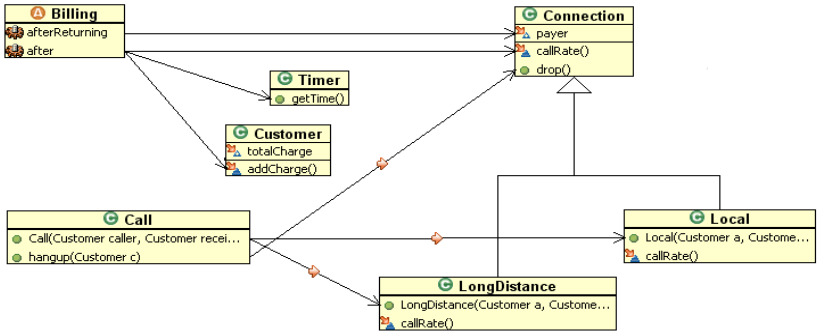


Figura 2.3 – Modelo de representação de interesses transversais após a projeção

Uma vez que a visão está construída, o programador pode expandir o modelo, adicionando mais informações, tanto verticalmente (mais detalhes sobre os elementos atuais do diagrama), quanto horizontalmente, acrescentando mais elementos. A operação chamada abstração é executada para permitir a contração do diagrama, resumindo informações a respeito dos relacionamentos.

O processo de extração de *ActiveAspect* é realizado sob demanda e tem como origem uma extensão da estrutura de dados produzida pelo compilador AspectJ, chamada Modelo de Estrutura Abstrato (MEA). Essa estrutura fornece informações a respeito dos elementos de um programa e de suas relações após compilado, como por exemplo, os aspectos e os elementos que seus comportamentos transversais atuam.

2.6.2 Lost

A linguagem *Lost* (PFEIFFER; SARDOS; GURD, 2005) é uma linguagem de consulta baseada em OQL (*Object Query Language*) (CATTELL; BARRY, 2000) projetada para busca em elementos estruturais de Java e AspectJ. Seu uso é facilitado por meio de um *plug-in* para a plataforma Eclipse, no qual programadores podem escrever consultas e obter os resultados sobre um programa escrito em AspectJ.

Por exemplo, a linguagem permite determinar quais pontos de junção um determinado conjunto de pontos de junção atua, ou ainda, quais aspectos estão atuando sobre um determinado ponto de junção. Além de aspectos, a linguagem *Lost* também atua sobre elementos Java,

como classes, métodos, construtores e campos. A ferramenta é implementada sob o framework CQT (*Code Query Tools*), que encapsula a maioria das funcionalidades comuns para modelos similares de consulta de código, como construtores e analisadores. O exemplo a seguir ilustra uma consulta realizada em Lost:

```
select Method
where Method.hasName("set|put")
where Method.isAdvisedBy(Aspect)
where !Method.isAdvisedBY(Aspect[2])
where Aspect.hasName("FormatChecker")
&& Aspect[2].hasName("PolicyEnforcer")
```

A consulta busca por todos os métodos cujo nome seja *set* ou *put* e que tenham atuação de um aspecto chamado *FormatChecker* e não tenha atuação de um aspecto chamado *PolicyEnforcer*. Não há evidências na literatura a respeito do modo de operação do processo de extração de *Lost*, portanto, não há como descrever sua origem. O processo de abstração ocorre com o cálculo do resultado a ser obtido por meio da análise e da execução da consulta. A apresentação é realizada através de um navegador hierárquico interativo, integrado à plataforma Eclipse.

2.6.3 Mylar

A ferramenta *Mylar* (KERSTEN; MURPHY, 2005) é uma ferramenta bastante similar à ferramenta *Lost*, porém sem uma linguagem explícita para consulta de código. O principal foco de *Mylar* está na apresentação, visando facilitar a visualização de elementos de um programa por intermédio de navegadores customizados. *Mylar* também é apresentada como um *plug-in* da plataforma Eclipse e é aplicável tanto para programas Java quanto AspectJ. O modelo empregado leva em consideração as atividades do programador, para capturar o grau de relevância de elementos do código em relação à tarefa executada, a fim de montar um mapa de grau de interesse (GDI). Por exemplo, quando um programador seleciona ou edita um elemento de um programa, a ferramenta aumenta o grau de interesse daquele elemento. O grau de interesse é diminuído sempre que com o passar do tempo, o elemento não for mais selecionado ou editado. Por meio do mapa de interesses, a ferramenta preenche as visões de Java e AspectJ na plataforma Eclipse, usando um espectro de cores. Para o processo de extração, a ferramenta *Mylar* utiliza um processo similar ao de *ActiveAspect*, obtendo informações de código através do MEA, de acordo com os filtros de interesse do mapa. Na sequência, a abstração é realizada e a apresentação ocorre por meio de visões estruturadas.

2.6.4 .QL

A linguagem .QL (VERBAERE; HAJIYEV; DE MOOR, 2007) é destinada a buscas em elementos de um programa orientado a objetos. É uma linguagem proprietária, utilizada principalmente na ferramenta de análise estática SemmleCode (VERBAERE; HAJIYEV; DE MOOR, 2007), voltada à análise de programas Java. A .QL é uma linguagem baseada em álgebra relacional, com estilo de programação orientada a objetos. Uma consulta escrita em .QL é otimizada e transformada em SQL para então ser executada em um banco de dados relacional. O exemplo a seguir ilustra a aplicação de .QL, a qual consulta busca por tipos que definam métodos com o nome *compareTo*, mas que não possuam métodos com o nome *equals*.

```
from Method m
   where m.hasName("compareTo") and
   not(m.getDeclaringType().declaresMethod("equals"))
   select m, "Is compareTo consistent with equals?"
```

2.6.5 Aspect Mining Tool

O framework *Aspect Mining Tool* (AMT) (HANNEMANN; KICZALES, 2001) tem por objetivo principal apoiar a identificação de interesses em um sistema. O AMT é composto por dois módulos: analisador e visualizador. O módulo analisador é baseado em uma versão modificada do compilador AspectJ e tem como principal objetivo extrair estatísticas baseadas em linhas de código fonte ou em tipos utilizados, além de outras informações relevantes, tais como pacotes, hierarquia de classes, etc. e colocá-las em arquivos de dados. Os arquivos de dados são então utilizado pelo módulo visualizador para representar o sistema por meios de unidades compiladas, possibilitando consultas à sua base de dados. O AMT oferece recursos a dois tipos de consulta: léxica (substrings e expressões regulares) e estrutural (tipos utilizados).

2.6.6 Outras tecnologias de consulta de código

Outras tecnologias relevantes para consulta em código têm sido desenvolvidas, sendo a maioria para sistemas orientado a objetos. A seguir, algumas dessas tecnologias são descritas:

- ***JQuery*** (MCCORMICK; DE VOLDER, 2004) é uma ferramenta baseada em TyRuBa (VOLDER, 1998), um sistema de propósito geral, fundamentado em metaprogramação lógica. A ferramenta é implementada como um *plug-in* para a plataforma Eclipse e apresenta resultados em navegadores hierárquicos que podem ser refinados por meio de con-

sultas escritas pelos programadores. O código a seguir apresenta um exemplo de consulta em JQuery na qual são selecionados métodos, classes e pacotes, cujo nome do método inicie com a string *draw*.

```
method(?M);
name(?M; ?name);
child(?C; ?M);
package(?C; ?P);
rematch(/^draw; ?name)
```

- ***CodeQuest*** (HAJIYEV; VERBAERE; MOOR, 2006) combina o uso de programação lógica com banco de dados tradicionais para realizar consultas em código. A estratégia utilizada por *CodeQuest* é utilizar Datalog como a linguagem de consulta e banco de dados relacionais, como SQL-Server e IBM-DB2 para armazenar os dados relacionados a um programa. Uma vez que a ferramenta é integrada à plataforma Eclipse, o processo de atualização do banco de dados é realizado de forma automática após uma unidade de compilação ser compilada pela plataforma. Ao executar uma consulta, *CodeQuest* traduz as construções de Datalog em código SQL equivalente para ser validado pelo gerenciador de banco de dados, que retorna o conjunto de dados resultante.

- **JTL (*Java Tool Language*)** (COHEN; GIL; MAMAN, 2006) é uma linguagem projetada para consulta de código Java. A linguagem permite a consulta de módulos estruturais de Java e suas inter-relações, comportamento condicional na execução do método e código interno aos métodos para fins de análise de fluxo de dados. O modelo fornecido por JTL também emprega Datalog para representar o modelo conceitual do programa, que é obtido por intermédio da inspeção de *bytecodes* para extrair dados do código. O código da linguagem de consulta segue a ideia do modelo *Consulta por exemplo* (ZLOOF, 1975). O código a seguir ilustra a busca de classes que contenham campos do tipo *long* ou *int* e nenhum método abstrato.

```
abstract class {
[long | int] field;
no abstract method;
```

2.7 Considerações Finais

Este capítulo discutiu os principais conceitos necessários a esta dissertação. O conceito de DSLs foi discutido e suas principais características em relação a uma GPL foram apresenta-

das. Foram abordados também os riscos e as oportunidades relacionados à criação e/ou ao uso de DSLs, sendo classificados em aspectos financeiros, técnicos e comportamentais. Um processo de desenvolvimento de DSLs foi apresentado, e as atividades de decisão, análise, projeto, implementação, validação, instalação e manutenção foram descritas. Em particular, as estratégias de implementação de DSLs de consulta em código foram abordadas no contexto do padrão extração-abstração-apresentação, presente em implementações desse tipo de DSL. Por fim, o capítulo discutiu os principais conceitos de programação orientada a aspectos e os trabalhos relacionados a esta dissertação. O capítulo seguinte trata da especificação da linguagem de busca proposta neste trabalho.

3 A LINGUAGEM AQL

Este capítulo descreve os recursos da DSL de consulta em código orientado a aspectos proposta neste trabalho, denominada AQL (*Aspect Query Language*). A AQL é uma linguagem declarativa, projetada para pesquisas em dados estruturados, de acordo com um metamodelo representando um programa orientado a aspectos. Os principais objetivos a serem alcançados pelo projeto de AQL são:

- Tornar a consulta a elementos de programas orientados a aspectos mais simples quando comparadas a abordagens com o uso de SQL ou OQL.
- Permitir o uso de conceitos equivalentes de OQL a fim de manter o fluxo de trabalho do programador sem alteração do paradigma de programação. Ao manter o paradigma de objetos para a realização de consultas, os programadores familiarizados com tecnologia OO tendem a tornar-se fluentes na linguagem de forma mais rápida.
- Permitir que evoluções no metamodelo de representação de programas não demandem modificações na sintaxe da linguagem. As alterações realizadas no modelo, feitas pela adição ou remoção de elementos, mantêm a sintaxe da linguagem, mesmo sendo uma linguagem específica para consulta em código.
- Permitir o uso de funções contextualizadas para a simplificação de consultas. As funções contextualizadas permitem simplificar as consultas, à medida que reduzem a quantidade de instruções para a realização de uma operação. A AQL oferece algumas funções pré-definidas, que serão apresentadas no decorrer deste capítulo.

A seguir é apresentado um exemplo introdutório para ilustrar o uso de AQL em um metamodelo de AspectJ. O código apresentado na Listagem 3.1 pesquisa por adendos que afetam classes de nome *Company*, que estejam associados a pontos de corte do tipo *call* e que estão declarados em aspectos localizados em pacotes iniciados em *br.ufsm.core*. A consulta retorna o nome qualificado do aspecto, a posição do adendo dentro do aspecto e o tipo do adendo (*before, after, around*).

```
find aspect a, class c
where a.advice.affects(c)
     and c.name = 'Company'
     and a.advice.pointcut.kind(call)
     and a.pack.name like 'br.ufsm.core%'
returns a.fullQualifiedName, a.advice.position, a.advice.kind
```

Listagem 3.1 – Exemplo de código AQL

A utilização de AQL é identificada em ao menos quatro finalidades: (i) em ferramentas de análise estática, (ii) em ferramentas de pesquisa em IDEs, (iii) em ambientes de desenvolvimento, como o AJDT, e (iv) em consultas complexas, envolvendo conectivos lógicos e relacionais, realizadas diretamente pelo programador.

Apesar de o projeto da linguagem AQL seguir modelos apoiados por linguagens de busca orientadas a objetos, ela não possui foco para definição e manipulação de dados. Seu uso é restrito a consulta de elementos e relações, não oferecendo recursos para atualização, inserção ou remoção de dados.

As seções seguintes apresentam os recursos da linguagem AQL e uma série de exemplos ilustrando sua utilização. Ao final do capítulo, são apresentados os requisitos mínimos para composição do metamodelo de representação de um programa orientado a aspectos requerido por AQL.

3.1 Visão Geral da Linguagem AQL

A linguagem AQL é uma linguagem declarativa, baseada em linguagens de consulta a objetos, que permite consultas aos elementos mapeados em um metamodelo representando um programa orientado a aspectos. Sua estrutura permite consultar objetos de maior ordem, também chamados de objetos recipientes, e seus respectivos atributos, aplicando-se, opcionalmente, condições por meio de conectivos lógicos e relacionais.

Os objetos de maior ordem são definidos em AQL como *project*, *package*, *class*, *interface*, *enum* e *aspect*, representando, respectivamente, os modelos de projeto, de pacote, de classe, de interface, de enumerações e de aspecto. O resultado de um consulta pode ser retornado como um objeto ou como uma lista de objetos. Mesmo o retorno de tipos primitivos como inteiros e tipos de ponto flutuante são encapsulados como um objeto no retorno da consulta. O resultado ainda pode ser ordenado de forma ascendente ou descendente, baseado em atributos resultantes.

A linguagem AQL fornece funções paramétricas, tanto de livre contexto, como contextualizadas. Uma diferença fundamental de AQL em relação às OQLs é sua capacidade de inferir construções de junção entre objetos, desonerando o usuário de explicitar as associações necessárias para executar determinadas consultas. Uma vez que o modelo a ser consultado seja conhecido por AQL, as junções necessárias podem ser inferidas pelo compilador, tornando as consultas mais concisas e consequentemente mais simples de serem escritas.

3.2 A Sintaxe de AQL

A linguagem AQL é formada por cinco cláusulas principais: *find*, *where*, *returns*, *order by* e *group by*. As cláusulas *find* e *returns* são as únicas cláusulas obrigatórias em uma consulta AQL. A consulta mais simples que pode ser escrita em AQL é a seguinte:

```
find aspect a returns a
```

Essa consulta retorna todos aspectos declarados no sistema como uma lista de objetos. A Figura 3.1 apresenta o grafo de sintaxe das principais cláusulas de AQL.

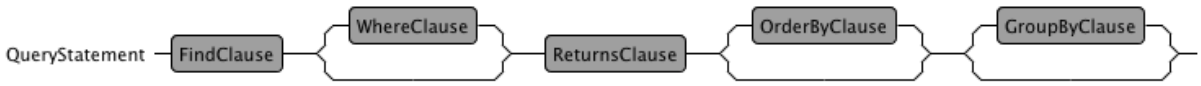


Figura 3.1 – Grafo de sintaxe das cláusulas principais de AQL

3.2.1 A Cláusula *find*

A cláusula *find* declara quais objetos de maior ordem farão parte da consulta, ou seja, quais serão utilizados na cláusula *where*, quando existir, e/ou na cláusula *returns*. A cláusula *find* é uma cláusula obrigatória na declaração de consultas e permite ainda associar a cada objeto um ou mais pseudônimos (*alias*). Cada pseudônimo representa um objeto diferente do mesmo tipo na consulta, permitindo dessa forma, a junção implícita de objetos do mesmo tipo. Para cada objeto pertencente à consulta, pelo menos um pseudônimo deve ser associado, não sendo permitidos objetos sem pseudônimos. Cada objeto é separado por vírgula, sendo permitido um mesmo objeto ser declarado mais de uma vez, desde que os pseudônimos não sejam homônimos. Alternativamente, um objeto pode conter vários pseudônimos, desde que, separados por espaço

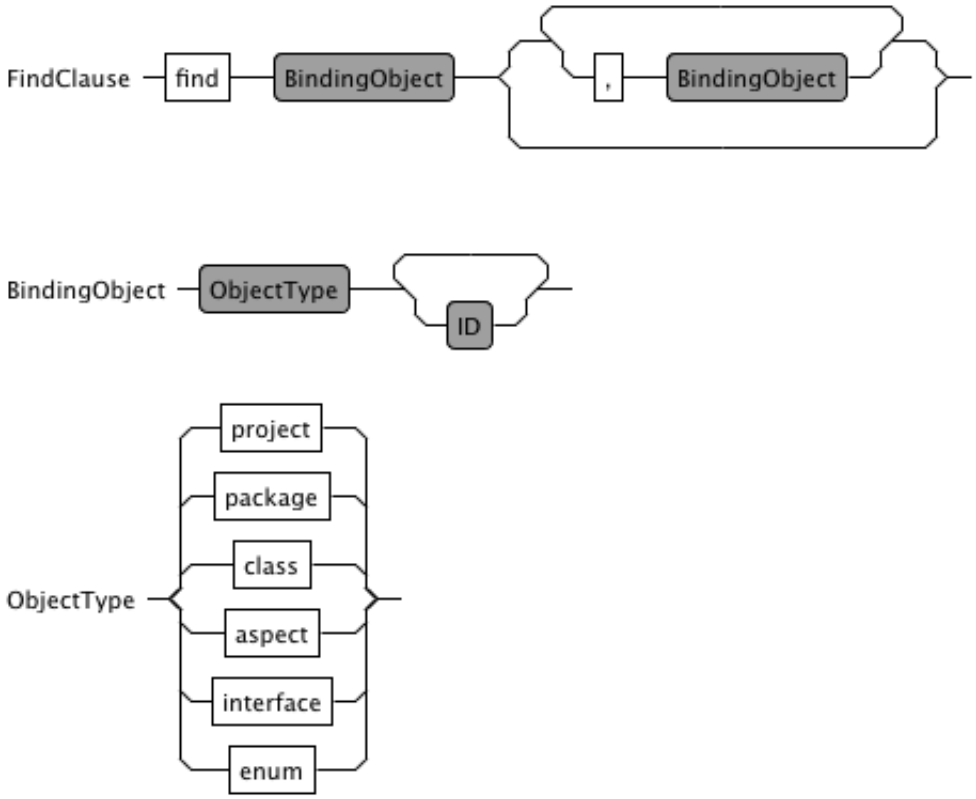


Figura 3.2 – Grafo de sintaxe da cláusula *find*

e, novamente, não violem a regra de homônimos. A Figura 3.2 apresenta o grafo de sintaxe da cláusula *find*.

Os exemplos a seguir ilustram trechos de código AQL com cláusulas *find* válidas:

```
1  find aspect a ...
2  find aspect a1 a2 ...
3  find aspect a1, class c1, class c2, interface i1 ...
```

A primeira cláusula (linha 1) mostra uma simples declaração de um objeto *aspect* com um pseudônimo associado *a*. Na cláusula *find*, declarada na linha 2, o objeto *aspect* possui dois pseudônimos nomeados *a1* e *a2*. A terceira cláusula (linha 3) apresenta o uso válido de repetição de objetos (*c1* e *c2*).

3.2.2 A Cláusula *where*

A cláusula *where* é utilizada para a aplicação de filtros na consulta. Ela permite limitar o resultado das consultas de acordo com certos critérios, de forma similar às cláusulas *where* encontradas nas principais linguagens de consulta, como JPQL (*Java Persistence Query Language*) (KEITH; SCHINCARIOL, 2006), HQL (*Hibernate Query Language*) (BAUER; KING,

2005), XQuery (WALMSLEY, 2009) e SQL. A Figura 3.3 apresenta o grafo de sintaxe da cláusula *where*. As expressões são descritas a partir de recursos que permitem a realização de computação aritmética, relacional e lógica. Na Subseção 3.2.12, as expressões permitidas em AQL são descritas em detalhes.

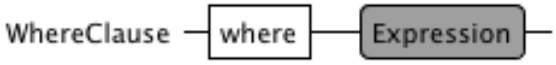


Figura 3.3 – Grafo de sintaxe da cláusula *where*

A cláusula *where* pode referenciar objetos declarados na cláusula *find*. Apenas uma cláusula *where* pode ser definida por consulta, mas múltiplas expressões encadeadas com conectivos *or* ou *and* podem ser definidas, como mostrado nas linhas 2 e 3 do exemplo a seguir.

```
1  find aspect a where a.name = 'Log' ...
2  find aspect a1 a2 where a1.extends(a2)
3      and a2.annotation.name = 'myAnnotation' ...
4  find aspect a1, class c1, class c2, interface i1 where c1.implements(i1)
      and c1.extends(c2) and a1.affects(c1) and a1.name like 'Log%' ...
```

Nos exemplos anteriores, a primeira consulta (linha 1) filtra aspectos cujos nomes são iguais a `'Log'`. A segunda consulta (linha 2) filtra aspectos *a1*, os quais tem como ascendente o aspecto *a2* e o aspecto *a2* uma anotação de nome `'myAnnotation'`. A terceira consulta (linha 3) filtra classes *c1* que implementam a interface *i1* e classes *c1* que tenham como ascendente do tipo de *c2* e que sejam afetadas pelo aspecto *a1*, sendo o nome do aspecto iniciado por `'Log'`.

3.2.3 A Cláusula *returns*

A cláusula *returns* permite definir os objetos, os atributos ou os valores que serão retornados na consulta. O grafo de sua sintaxe pode ser observado na Figura 3.4. A cláusula *returns* é composta da palavra *returns* e de uma sequência de expressões separadas por vírgula. Cada expressão pode ter um pseudônimo associado. Objetos relacionados na cláusula *returns* devem ter sido obrigatoriamente declarados na cláusula *find*, caso contrário um erro de violação semântica é apontado pelo compilador de AQL. A cláusula *returns* pode ainda conter as palavras-chave *distinct* e *all*, de mesma semântica encontrada em SQL.

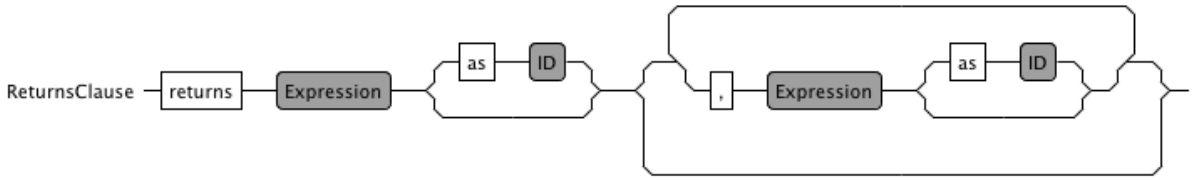


Figura 3.4 – Grafo de sintaxe da cláusula *returns*

As seguintes consultas são exemplos de uso da cláusula *returns*.

```
1  find aspect a where a.name = 'Log' returns a
2  find aspect a1 a2 where a1.extends(a2) and a2.annotation.name = '
       myAnnotation' returns a1.name, a2.name, a1.modifier.code, a1.modifier.
       name
3  find aspect a1, class c1, class c2, interface i1 where c1.implements(i1)
       and c1.extends(c2) and a1.affects(c1) and a1.name like 'Log%' returns
       a1.qualifiedName, c1.name, c2.name, i1.name
```

A primeira consulta (linha 1) retorna uma lista de aspectos. A consulta descrita na linha 2 retorna o nome do aspecto *a1*, o nome do aspecto *a2*, o código e a descrição do modificador do aspecto *a1*. A terceira consulta retorna o nome qualificado do aspecto *a1*, o nome das classes *c1* e *c2* e o nome da interface *i1*.

3.2.4 A Cláusula *order by*

A cláusula *order by* permite que o resultado de uma consulta seja ordenado por uma ou mais propriedades dos objetos retornados. Cada propriedade deve ser separada por vírgula e o objeto deve constar na cláusula *find* para que a propriedade seja válida. A Figura 3.5 apresenta o grafo de sintaxe da cláusula *order by*.

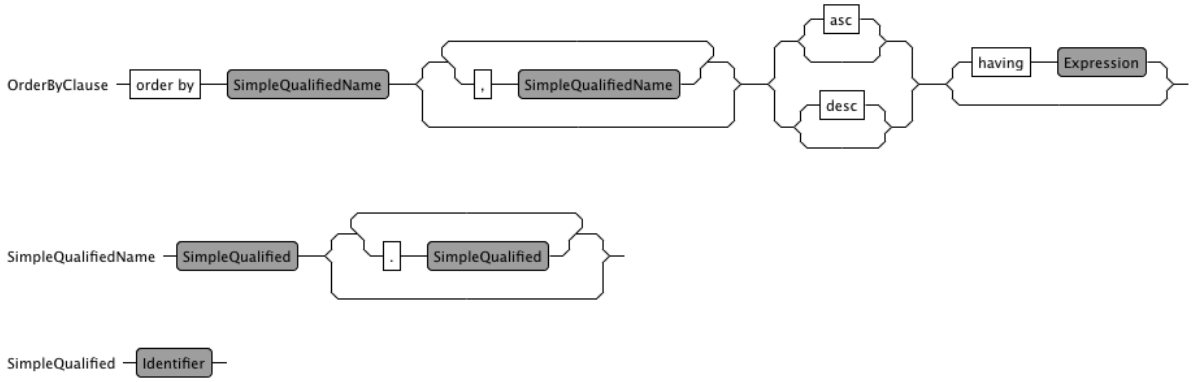


Figura 3.5 – Grafo de sintaxe da cláusula *order by*

A cláusula ainda permite o uso dos modificadores asc e $desc$, indicando se a ordenação

deve ser em ordem ascendente ou descendente, respectivamente. Se nenhum modificador for especificado, a ordenação é realizada de forma ascendente.

O exemplo a seguir ilustra o uso da cláusula *order by*. A consulta retorna o nome do pacote e o nome qualificado completo dos aspectos cujos nomes qualificados completos iniciam em `'br.ufsm.lpbd.banking.aspect.Log'`. A lista retornada será ordenada ascendentemente pelos campos nome do pacote e nome completo qualificado do aspecto.

```
find aspect a
where a.qualifiedName like 'br.ufsm.lpbd.banking.aspect.Log%'
returns a.pack.name, a.qualifiedName
order by a.pack.name, a.qualifiedName asc
```

3.2.5 A Cláusula *group by*

A cláusula *group by* de AQL possui a mesma semântica encontrada em linguagens de consulta como SQL e JPA-Query. Ela permite resumir e agrupar dados em uma consulta e é realizada por funções de agregação, como *SUM*, *COUNT*, *AVG*, *MIN* e *MAX*. O exemplo a seguir ilustra o uso da cláusula *group by*. A consulta retorna a quantidade de classes afetadas agrupadas pelo nome do aspecto que as afeta.

```
find class c, aspect a
where a.affects(c)
returns a.name, count(c)
group by a.name
```

O grafo de sintaxe da cláusula *group by* para AQL é mostrado na Figura 3.6.

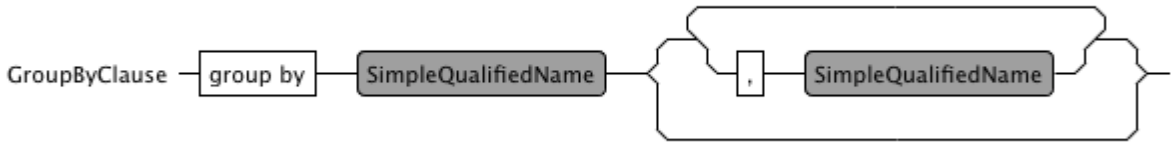


Figura 3.6 – Grafo de sintaxe da cláusula *group by*

3.2.6 Identificadores

Os identificadores em AQL devem iniciar por uma letra (maiúscula ou minúscula) ou o caracter sublinhado (_) seguido de letras, números ou caractere sublinhado, desde que não sejam palavras reservadas da linguagem (apêndice B). A definição de identificadores em EBNF

é descrita da seguinte forma:

```
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
```

Os seguintes identificadores são exemplos válidos em AQL:

```
a, aa, aspect1, abc, A_a, Z1a_b, Wabc, _123, _abc
```

3.2.7 Literais

Os literais são valores constantes representados diretamente na consulta e são utilizados em expressões ou em parâmetros de funções. A linguagem AQL reconhece duas categorias de literais: os literais numéricos e os literais *String*. Os literais numéricos podem ter a forma de um inteiro, como 10, de valores decimais, como 10.5 ou como valores de ponto flutuante como 10.5E2. Os literais *String* são reconhecidos por estarem entre aspas simples (') ou duplas ("), ou ainda pertencerem a um conjunto de símbolos de *escape* (precedidos pelo símbolo \). A AQL reconhece três sequências de caracteres com semântica especial: *null*, *true* e *false*. Elas representam respectivamente um objeto nulo, uma condição verdadeira e uma condição falsa. A definição em EBNF de literais em AQL é descrita como segue:

```
Literal : String | PontoFlutuante | Inteiro | 'null' | 'true' | 'false'

Inteiro : '0'..'9'+

PontoFlutuante : ( ('0'..'9')+ ( '.' ('0'..'9')+ ( ('E'|'e') ('-')?
    ('0'..'9')+ )? )? )

String :
   '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"'
 | "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'"
```

Os seguintes exemplos são literais válidos em AQL:

```
'abc', "abc", 1234, 09876.45, null, true, false
```

3.2.8 Nomes Qualificados

A qualificação de nomes em AQL segue a mesma semântica de Java, sendo o caractere ponto (.) o qualificador para uma sequência de identificadores, permitindo também funções em sua composição. A Figura 3.7 apresenta o grafo de sintaxe de nomes qualificados.

O exemplo a seguir ilustra a descrição de alguns nomes qualificados válidos em AQL:

```
aspect1, aspect1.attribute1, aspect1.function1(), aspect1.function1().
    attribute1
```

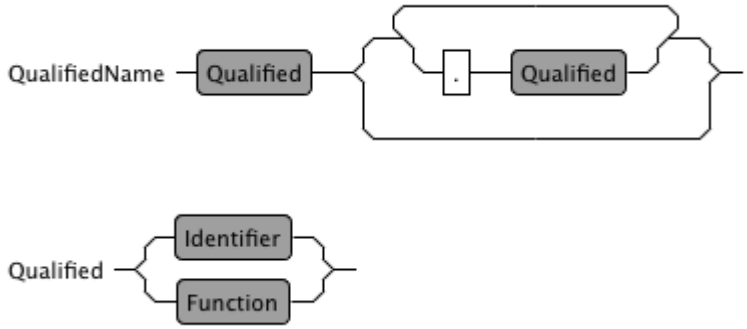


Figura 3.7 – Grafo de sintaxe de nomes qualificados

3.2.9 Comentários

As linhas de código iniciadas em dupla barra (//) são ignoradas e consideradas comentários em AQL. Adicionalmente, múltiplas linhas podem ser comentadas com o uso do bloco /* . . . */ .Uma exceção a essas regras ocorre quando os símbolos de comentário estão localizados internamente a um literal *String*. Por exemplo, o seguinte trecho de código contido entre /* e */ não será considerado comentário por AQL:

```
find aspect a1 where a1.name = 'Aspect/* Log */'
```

3.2.10 Chamada de Funções

As funções em AQL podem ser chamadas tanto na cláusula *where* como na cláusula *returns*, e são classificadas como livres de contexto e contextualizadas. As funções livres contexto são funções sem associações explícitas aos objetos da consulta, sendo referenciadas diretamente pelo seu nome e por seus parâmetros. Exemplos comuns de funções livres de contexto são SUM(. . .), COUNT(. . .) e AVG(. . .). As funções livres de contexto são funções escritas conforme a sintaxe descrita no grafo da Figura 3.8

O identificador (ID) representa o nome da função e os parâmetros são passados para a função separados um a um por vírgula. Um parâmetro é representado por uma expressão, podendo ser um literal, um nome qualificado ou ainda uma expressão complexa envolvendo conectivos lógicos, relacionais e aritméticos. O uso de parênteses é obrigatório para identificar

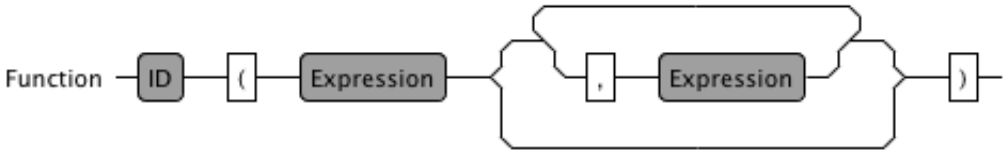


Figura 3.8 – Grafo de sintaxe de funções não contextualizadas

uma função, os quais devem ser mantidos mesmo quando não existam parâmetros na chamada da função.

As funções contextualizadas são funções escritas no formato de nomes qualificados, ou seja, antes da chamada da função um qualificador é declarado para indicar seu contexto de execução. A Figura 3.9 apresenta um exemplo de uma chamada de função contextualizada. O objeto identificado como *a1* invoca a função *extends* passando como parâmetro o objeto *a2*.

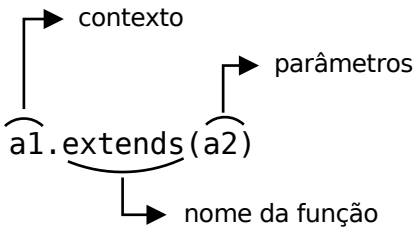


Figura 3.9 – Exemplo de função contextualizada em AQL

3.2.11 Funções Contextualizadas Pré-definidas

A fim de simplificar a construção de determinadas consultas, a linguagem AQL especifica um conjunto de funções contextualizadas que permite reduzir a escrita de instruções para a execução de uma operação.

- **Função extends:** Retorna verdadeiro se o objeto passado como parâmetro é ascendente direto do objeto de contexto.

  Assinatura: QualifiedName.extends (ID) : boolean

  Contextos aplicáveis: *class*, *aspect*, *interface*

  Uso: A função *extends* aceita um identificador de um objeto definido na cláusula *find* como parâmetro. O objeto declarado é representado por seu pseudônimo, o qual é um identificador válido em AQL.

  Exemplo:

```
find class c1 c2
where c2.extends(c1)
returns c1.name, c2.name
```

- **Função implements:** Retorna verdadeiro se o objeto passado como parâmetro é implementado pelo objeto de contexto.

  Assinatura: QualifiedName.implements ($ID_1, ID_2, \ldots, ID_n$) : boolean

  Contextos aplicáveis: *class*, *aspect*

  Uso: A função *implements* aceita uma lista de identificadores de objeto definido na cláusula *find* como parâmetro. Esse objeto deve obrigatoriamente ser do tipo (*interface*).

  Exemplo:

```
find class c , interface i
where c.implements(i)
returns c.name, i.name
```

- **Função modifier:** Retorna verdadeiro se pelo menos um tipo de modificador passado como parâmetro for encontrado no objeto de contexto.

  Assinatura: QualifiedName.modifier ($ID_1, ID_2, \ldots, ID_n$) : boolean

  Contextos aplicáveis: *class*, *aspect*, *interface*, *enum* e objetos associados que possuem modificadores

  Uso: A função *modifier* aceita uma lista de identificadores contendo os tipos de modificadores válidos para um metamodelo aplicado.

  Exemplo:

```
find class c1
where not (c1.modifier(public))
returns c1.name
```

- **Função affects:** Retorna verdadeiro se o objeto passado como parâmetro é afetado pelo objeto de contexto.

  Assinatura: QualifiedName.affects (ID) : boolean

  Contextos aplicáveis: *aspect* e objetos associados que possam afetar um outro objeto. A função *affects*, por exemplo, pode retornar aspectos ou adendos que afetam, de forma estática ou dinâmica, os pontos de junção localizados em classes ou mesmo em outros aspectos.

  Uso: A função *affects* aceita um identificador de um objeto definido na cláusula *find*

como parâmetro.

Exemplo:

```
find aspect a, class c
where a.advice.affects(c)
     and a.advice.kind(before)
returns a.name, a.advice.position, c.name
```

- **Função affectedBy:** Retorna verdadeiro se o objeto passado como parâmetro afeta o objeto de contexto.

  Assinatura: QualifiedName.affectedBy (ID) : boolean

  Contextos aplicáveis: *class*, *aspect*, *interface* e objetos associados que possam ser afetados por aspectos.

  Uso: A função *affectedBy* aceita um identificador de um objeto definido na cláusula *find* como parâmetro do tipo *aspect*.

  Exemplo:

```
find aspect a, class c
where c.method.affectedBy(a)
returns a.name, c.name, c.method.name
```

- **Função kind:** Retorna verdadeiro se pelo menos um tipo de adendo passado como parâmetro for encontrado no objeto de contexto.

  Assinatura: QualifiedName.kind (ID1, ID2, .., IDn) : boolean

  Contextos aplicáveis: *aspect.advice*, *class.pointcut*, *aspect.pointcut*

  Uso: A função *kind* aceita uma lista de identificadores contendo os tipos de adendos ou tipos de pontos de corte a serem comparados no objeto de contexto.

  Exemplo:

```
find aspect a
where a.advice.kind(before)
returns a.name, a.advice.kind, a.advice.code
```

As consultas em AQL não distinguem letras maiúsculas de minúsculas para as palavras-chave, ainda que uma implementação em particular possa aplicar regras de distinção. Contudo, para os nomes qualificados, os identificadores e as funções, a regra de distinção entre letras maiúsculas e minúsculas é válida.

3.2.12 Expressões

As expressões em AQL são unidades básicas de avaliação. Uma consulta em AQL pode conter expressões com várias subexpressões, as quais podem ser compostas de outras subexpressões. A Tabela 3.1 apresenta as categorias de expressões especificadas em AQL. Cada expressão é avaliada para uma sequência, a qual pode ser um valor atômico simples, como um literal ou um nome qualificado, ou então uma outra expressão completa.

Tabela 3.1 – Categorias de expressões em AQL

Categoria	**Operadores ou palavras-chave**	**Descrição**
Operadores aritméticos	$+, -, *, /, \%, \wedge$	Adição, subtração, multuplicação, divisão, resto da divisão e potência
Operadores relacionais	$>, <, >=, <=, =, != , like$	Comparação baseada em valores ou atributos
Operadores lógicos	and, or, not	Operadores lógicos de conjunção, disjunção e negação
Agrupamento	( )	Permitem definir prioridades de avaliação
Quantificados	$in, not\ in$	Determinam se uma sequência preenche específicas condições
Funções	<NomeFunção>(<params>...)	Chamada a funções contextualizadas e livres de contexto

A listagem completa com as definições formais da linguagem AQL, incluindo expressões, pode ser encontrada no apêndice A.

3.3 Definição de um Metamodelo de POA

Conforme mencionado no início deste capítulo, a AQL é uma linguagem voltada para a consulta de dados sobre programas orientados a aspectos, de acordo com um metamodelo que representa os elementos e as respectivas relações entre esses elementos. À medida que diferentes linguagens implementam os conceitos de orientação a aspectos de variadas formas, alguns requisitos mínimos foram definidos para que uma consulta AQL seja executável. Esses requisitos são agrupados em duas categorias: a primeira categoria define os elementos de maior ordem, em conformidade com a cláusula *find* (projeto, pacote, classe, interface, enumerações e aspecto). A segunda categoria inclui elementos que definem as características de um programa com sendo orientado a aspectos.

A linguagem AQL é independente do paradigma utilizado na representação dos dados de um programa. De fato, a decisão sobre o formato utilizado é realizado durante a implementação da linguagem, podendo ser, por exemplo, um modelo orientado a objetos, um modelo relacional, um modelo lógico, dentre outros.

Nas seções seguintes, a noção de um modelo de programas orientados a aspectos é formalizada por meio de um modelo matemático, baseado em álgebra relacional. Tal formalização é fundamentada no conceito de relações nomeadas, possibilitando a associação direta entre uma relação matemática e um significado. A formalização proposta é uma extensão para aspectos do trabalho apresentado por Robillard e Murphy (ROBILLARD; MURPHY, 2007), no qual um conjunto de funções de mapeamento para a linguagem Java é definido.

3.3.1 Definições Formais de um Programa

Com o objetivo de orientar a definição de um modelo para programas orientado a aspectos, esta seção apresenta um conjunto de definições formais para um programa.

Definição 1 (Elemento de programa). Um elemento $e \in E$ de um programa é qualquer elemento que pode ser individualmente investigado no código. Tipicamente em um programa orientado a objetos, esses elementos se traduzem em classes, interfaces, campos e métodos. Em programas orientados a aspectos, esses elementos, em geral, são constituídos por aspectos, pontos de corte e adendos.

Definição 2 (Relação). Uma relação $r = (e_1, e_2) \in R$ é uma dependência de programa entre dois elementos e_1 e e_2. Típicas relações entre elementos de um programa são chamadas a métodos e a adição de comportamento a um método realizada por um adendo.

Definição 3 (Relação Nomeada). Uma relação nomeada $R_n = (n, r)$ consiste de um nome n com uma relação binária r. Um programa é modelado genericamente como um conjunto de elementos declarados no programa e um conjunto de relações nomeadas entre esses elementos.

Definição 4 (Modelo de programa). Um modelo de programa $P = (E, N)$ consiste em um conjunto de elementos de um programa $E = \{e_1, e_2, ..., e_n\}$ e um conjunto de relações nomeadas sobre E. A definição de programa expressa que todo conhecimento em relação a um programa pode ser expresso em termos de relações entre seus elementos.

Definição 5 (Conjunto de nomes). Seja $N = \{R_{n1}, R_{n2}, ..., R_{nk}\}$ um conjunto de relações nomeadas, um conjunto de todas as relações nomeadas em N é definido como $names(N) =$

$\{n \mid \exists R : (n, R) \in N\}$.

Definição 6 (Transposição). Dada uma relação $R_n = (n, r) \in N$, sendo $r = (e_1, e_2)$, sua transposta é definida como $R_n^\tau = (n^\tau, r^\tau)$, sendo $r^\tau = (e_2, e_1)$. Em um programa, qualquer relação apresenta sua transposta, sendo $r \in R \Rightarrow r^\tau \in R$.

Definição 7 (Função de mapeamento). Seja $P_M = \{E, N\}$ um modelo de programa, a função de mapeamento M consiste em:

- Um critério definindo quais elementos declarados em um programa P deveriam estar listados em E.
- Um conjunto de nomes de relações permitidos pelo modelo.
- A definição de uma função de análise $a(n, P)$, tendo como parâmetros o nome de uma relação n e um programa P, retornando a relação nomeada $R_n \subseteq E \times E$, representando as relações entre os elementos do programa P (de encontro com os critérios de mapeamento), de acordo com a semântica de n.

As funções de mapeamento representam uma forma de se obter um modelo de um programa a partir de um programa concreto. Por definição, as funções de análise serão definidas com uso da lógica de primeira ordem, e as funções de mapeamento serão especificadas da seguinte maneira:

- O nome da função de mapeamento em negrito e grifado,
- O critério de inclusão de um elemento x de um programa P em E (o conjunto de elementos modelados),
- O conjunto de $names(N)$ dos nomes das relações permitidas, e
- A definição da(s) função(ões) de análise $a(n, P)$.

3.3.2 Definição de um Programa Orientado a Aspectos

O modelo de um programa orientado a aspectos define os elementos e as relações mínimas que caracterizam um programa como sendo orientado a aspectos. Conceitualmente, esses elementos e relações são definidas na função de mapeamento, na qual são acrescentados os elementos, os nomes de relações e de funções de análise que correspondem a atuação de aspectos, de acordo com a semântica da linguagem mapeada.

Inicialmente são definidos os elementos básicos (elementos de maior ordem) do metamodelo requerido para a execução de uma consulta em AQL. Conforme já mencionado, esses elementos representam um projeto, um pacote, uma classe, uma interface, uma enumeração ou um aspecto dentro de um programa. A Tabela 3.2 apresenta as construções e a semântica dos elementos, das relações e das funções de análise requeridos em um metamodelo para a execução de AQL.

Tabela 3.2 – Elementos e relações básicas de maior ordem

Construção	**Tipo**	**Semântica**
$IsProject(\ldots)$	Elemento	Um elemento representando o projeto no qual o programa está inserido
$IsPackage(\ldots)$	Elemento	Um elemento representando o pacote no qual o programa está inserido
$IsClass(\ldots)$	Elemento	Um elemento representando uma classe no programa
$IsInterface(\ldots)$	Elemento	Um elemento representando uma interface no programa
$IsAspect(\ldots)$	Elemento	Um elemento representando um aspecto no programa
$IsEnum(\ldots)$	Elemento	Um elemento representando uma enumeração no programa
$Declares$	Relação	Uma relação entre elementos representando uma declaração
$DeclaredBy$	Relação	A relação transposta de $Declares$
$a(Declares, P)$	Função de análise	Retorna as relações representando elementos declarados no programa
$a(DeclaredBy, P)$	Função de análise	A função transposta da função $Declares$

O metamodelo ainda necessita conter elementos e suas respectivas relações que possibilitem afetar de forma estática ou dinâmica os pontos de junção, de acordo com o modelo de ponto de junção proposto na linguagem orientada a aspectos. A Tabela 3.3 apresenta os elementos, as relações e as funções de análise requeridas para a caracterização de um programa orientado a aspectos para a AQL.

É importante notar que o elemento que representa um adendo ($isAdvice(\ldots)$) é significativo tanto para construções que afetam pontos de junção de forma dinâmica quanto de forma estática. De fato, outras relações e elementos podem ser especificados de acordo com

Tabela 3.3 – Elementos e relações básicas em um modelo para POA

Construção	**Tipo**	**Semântica**
$IsAdvice(\ldots)$	Elemento	Um elemento representando um adendo no programa
$AffectsStatic$	Relação	Uma relação entre elementos de um aspecto e elementos afetados estaticamente no programa
$AffectedByStatic$	Relação	A relação transposta da relação $AffectsStatic$
$AffectsDynamic$	Relação	Uma relação entre elementos de um aspecto e elementos afetados dinamicamente no programa
$AffectedByDynamic$	Relação	A relação transposta da relação $AffectsDynamic$
$a(AffectsStatic, P)$	Função de análise	Retorna as relações representando elementos que afetam estaticamente outros elementos no programa
$a(AffectedByStatic, P)$	Função de análise	A função transposta de $a(AffectsStatic, P)$
$a(AffectsDynamic, P)$	Função de análise	Retorna as relações representando elementos que afetam dinamicamente outros elementos no programa
$a(AffectedByDynamic, P)$	Função de análise	A função transposta de $a(AffectsDynamic, P)$

as estruturas da linguagem orientada a aspectos em que a AQL será utilizada. Por exemplo, a Tabela 3.4 apresenta uma implementação parcialmente mapeada para a linguagem AspectJ. Além dos elementos e das relações mínimas requeridas, a função $F1$ especifica alguns outros elementos e relações que fazem sentido na linguagem AspectJ, tais como $isMethod(x)$ representando um método no programa, $isPointcut$ representando um ponto de corte no programa, $ExtendsClass$ representando uma relação de extensão para classes, $ExtendsAspect$ representando uma relação de extensão para aspectos e $Implements$ representando uma relação de implementação de uma ou mais interfaces.

3.3.3 Exemplo de Modelo de Programa Orientado a Aspectos

Uma função de mapeamento especifica como produzir um modelo de programa, a partir da descrição de elementos, relações e funções de análise. Diferentes funções de mapeamento podem ser empregadas para a construção de um modelo de um programa, dependendo da abrangência de pesquisa necessária por parte da linguagem AQL. A fim de ilustrar a construção do

Tabela 3.4 – Função de mapeamento da linguagem AspectJ

Função de mapeamento : F1
$E = \{x \mid IsClass(x) \wedge IsInterface(x) \wedge IsAspect(x)$ $\wedge IsEnum(x) \wedge IsProject(x) \wedge IsPackage(x)$ $\wedge isMethod(x) \wedge isAdvice(x) \wedge isPointcut(x)\}$
$names(N) = \{Declares, AffectsStatic, AffectedByStatic,$ $AffectedByDynamic, AffectedByDynamic, ExtendsClass$ $ExtendedByClass, ExtendsAspect, ExtendedByAspect,$ $Implements, ImplementedBy\}$
$a(Declares, P) = \{(x,y) \mid Declares(x,y)\}$ $a(AffectsStatic, P) = \{(x,y) \mid AffectsStatic(x,y)\}$ $a(AffectedByStatic, P) = a(AffectsStatic, P)\tau$ $a(AffectsDynamic, P) = \{(x,y) \mid AffectsDynamic(x,y)\}$ $a(AffectedByDynamic, P) = a(AffectsDynamic, P)\tau$ $a(ExtendsClass, P) = \{(x,y) \mid ExtendsClass(x,y)\}$ $a(ExtendedByClass, P) = a(ExtendsClass, P)\tau\}$ $a(ExtendsAspect, P) = \{(x,y) \mid ExtendsAspect(x,y)\}$ $a(ExtendedByAspect, P) = a(ExtendsAspect, P)\tau\}$ $a(Implements, P) = \{(x,y) \mid Implements(x,y)\}$ $a(ImplementedBy, P) = a(Implements, P)\tau\}$

modelo de um programa, a Listagem 3.2 apresenta dois trechos de código AspectJ de um programa *P1*, no qual uma classe (*C1*) e um aspecto (*A1*) são declarados.

```
1  public class C1 extends C0 implements I1 {
2     public void m1() {}
3  }
4
5  public aspect A1 extends A0 {
6     pointcut p() : execution (* C1.m1(..));
7
8     @AdviceName("a1")
9     before () : p() {}
10 }
```

Listagem 3.2 – Programa P1

De acordo com a semântica de AspectJ e em conformidade à função de mapeamento $F1$, o conjunto E_{P1} de elementos e as relações são construídos. Na prática, tanto os elementos quanto as relações são computados por uma implementação específica, que, em geral, utiliza técnicas de análise estática para inspecionar os programas e coletar as informações para a construção do modelo. A Tabela 3.5 apresenta o modelo do programa *P1*.

Tabela 3.5 – Modelo $P1_{F1}$

Modelo $P1_{F1}$
$E_{P1} = \{C0, C1, m, A1, a1, p, A0, I1\}$
$Declares_{P1} = \{(A1, a1), (C1, m)\}$ $AffectsDynamic_{P1} = \{(a1, C1), (a1, m), (A1, C1), (A1, m)\}$ $AffectedByDynamic_{P1} = \{(C1, a1), (m, a1), (C1, A1), (m, A1)\}$ $ExtendsClass_{P1} = \{(C1, C0)\}$ $ExtendedByClass_{P1} = \{(C0, C1)\}$ $Implements_{P1} = \{(C1, I1)\}$ $ImplementedBy_{P1} = \{(I1, C1)\}$ $ExtendsAspect_{P1} = \{(A1, A0)\}$ $ExtendedByAspect_{P1} = \{(A0, A1)\}$

A implementação de referência do compilador de AQL, apresentada no capítulo 4, mostra detalhes do processo de inspeção e coleta de informações para programas AspectJ. O metamodelo proposto para essa implementação está descrito em UML e pode ser observado no apêndice C.

3.4 Considerações Finais

Este capítulo apresentou os principais recursos da linguagem AQL, como uma linguagem específica de domínio para consulta em código orientado a aspectos. A linguagem AQL é uma linguagem declarativa, derivada de OQL, a qual permite consultar elementos de acordo com um metamodelo que representa um programa orientado a aspectos. Como principais características, a AQL permite o uso de expressões aninhadas, funções pré-definidas e inferência de construções de junção. Este capítulo apresentou também os requisitos mínimos para um metamodelo para representar um programa orientado a aspectos no qual uma consulta AQL pode ser executada. O próximo capítulo trata de uma implementação de referência da linguagem AQL.

4 IMPLEMENTAÇÃO

Este capítulo apresenta detalhes da arquitetura e da implementação de referência da linguagem AQL. A versão atual de AQL é formada por dois subprojetos. O primeiro subprojeto é responsável pela análise do código fonte e subsequente mapeamento de seus elementos para um metamodelo. Esse projeto é denominado AOPJungle e, fundamentalmente, foi desenvolvido para a realização da extração, abstração e mapeamento de informações de programas escritos em AspectJ, a linguagem orientada a aspectos na qual o metamodelo foi construído nesta implementação de referência. O segundo subprojeto refere-se à implementação do compilador para a linguagem AQL. Juntos, os projetos somam cerca de 23 mil linhas de código em 210 classes, interfaces e aspectos, implementados nas linguagens Java, AspectJ e Xtend.

A abordagem adotada na implementação do compilador de AQL empregou a transformação fonte a fonte, ou seja, as instruções em AQL são transformadas para uma linguagem de consulta mais genérica, a qual realiza a busca no modelo e repassa os resultados a AQL para então serem retornados ao executor da consulta. A atual implementação de referência utiliza a linguagem HQL como linguagem alvo, responsável pela busca dos objetos. A HQL é uma linguagem de consulta a objetos persistidos em bancos de dados por meio do framework Hibernate (BAUER; KING, 2005), e emprega um processo similar de transformação de linguagens para a realização da consulta. De fato, o compilador de HQL transforma a consulta solicitada em uma consulta SQL, a partir de regras de mapeamento objeto/relacional (O/R).

A decisão pelo uso de HQL foi fundamentada pelos seguintes motivos: *(i)* HQL é uma linguagem próxima a SQL e oferece uma grande quantidade de recursos para a realização de consultas em objetos persistidos em banco de dados, *(ii)* as consultas baseadas em modelos relacionais são escaláveis e mais eficazes do que consultas baseadas diretamente em objetos, utilizando-se técnicas de força bruta, como filtros em listas de objetos (HAJIYEV; VERBAERE; MOOR, 2006), e *(iii)* a linguagem HQL apresenta algumas instruções com sintaxe e semântica equivalentes em AQL. Para essas instruções, o processo de transformação é realizado de modo uniforme (1 para 1). Além disso, uma consulta declarada em AQL sempre encontra uma consulta equivalente em HQL.

Por outro lado, a escolha de HQL introduz alguns aspectos negativos para o processo de transformação, tais como: *(i)* o uso de HQL vincula a implementação de referência da linguagem AQL com uma tecnologia em particular, o que a torna dependente da mesma, e *(ii)* uma

segunda transformação de linguagem pode onerar a execução da consulta. O processo de transformação de HQL para SQL é realizado de acordo com o mapeamento O/R realizado entre as entidades do sistema. Essa transformação nem sempre é realizada na relação um para um, ou seja, dependendo do mapeamento O/R e da complexidade da pesquisa, uma consulta HQL é resolvida com a execução de várias consultas SQL.

A implementação de AQL foi baseada em um metamodelo (apêndice C) construído para representar programas escritos em AspectJ em sua versão 5, em conformidade à especificação de Java 7. A escolha de AspectJ como linguagem de estudo foi baseada nos seguintes critérios: (i) maturidade da linguagem: a linguagem AspectJ possui um compilador e um combinador estáveis, sendo referência primária na comunidade de aspectos, (ii) disponibilidade de ferramentas: existem ferramentas e ambientes integrados para programação em AspectJ e (iii) código aberto à pesquisa: o código do compilador é aberto à pesquisa e consulta.

A avaliação da implementação de AQL foi realizada com um conjunto de projetos escritos em Java e AspectJ, nos quais foram submetidas consultas escritas em AQL. As consultas foram então comparadas com as saídas esperadas no processo de transformação para HQL e submetidas à execução, para que os resultados fossem então avaliados. Um estudo de caso (Capítulo 5) também foi realizado, no qual foi desenvolvida uma ferramenta para identificação de oportunidades de refatoração em código orientado a aspectos, utilizando os recursos de AQL. Ao final deste capítulo são apresentadas algumas métricas de desempenho do processo de transformação e um comparativo entre o tamanho das consultas em AQL e HQL.

4.1 A Arquitetura de AQL

A implementação dos projetos de AQL utilizou um conjunto de tecnologias relacionadas à plataforma Eclipse tanto para a reificacão do código fonte (instanciação do metamodelo), quanto para a implementação do compilador de AQL. Os projetos foram implementados como um conjunto de *plug-ins* para a plataforma Eclipse, possibilitando uma integração natural com outras ferramentas que venham a ser desenvolvidas em conjunto com o AQL nessa plataforma.

A plataforma Eclipse fornece uma interface comum com o usuário e pode ser executada em diferentes sistemas operacionais. Ela consiste de uma plataforma de execução (*runtime platform*) e um conjunto de *plug-ins* padrão implantado como parte do projeto básico. De modo geral, a plataforma de execução do Eclipse representa o núcleo da plataforma e é responsável por descobrir, carregar e executar os *plug-ins* disponíveis. A Figura 4.1 mostra o modelo de

AQL integrado à plataforma Eclipse.

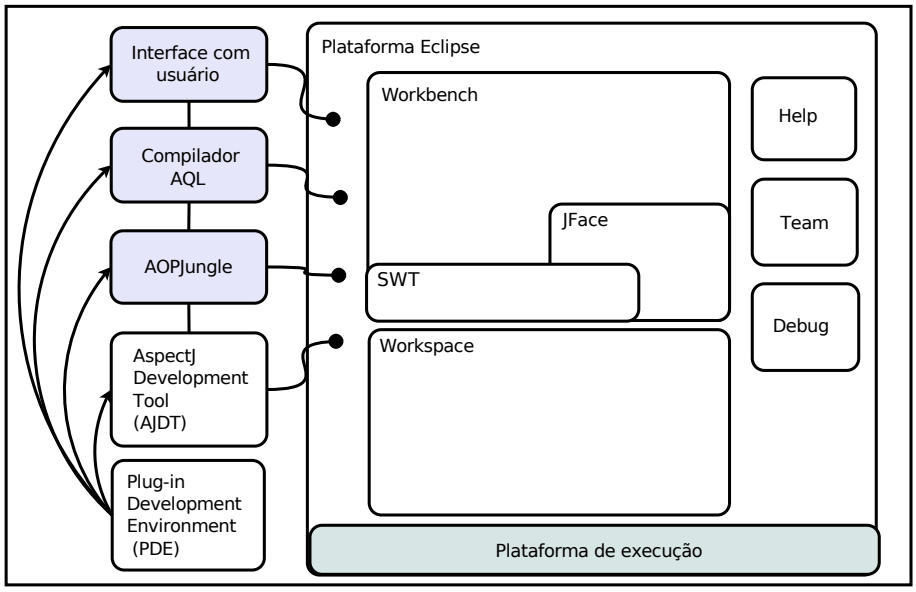


Figura 4.1 – Modelo de AQL integrado ao Eclipse IDE

O módulo de interface com o usuário foi idealizado para a escrita de consultas em AQL e para a apresentação dos resultados da consulta. Tanto o compilador de AQL quanto o editor de consultas foram construídos a partir da tecnologia Xtext (EYSHOLDT; BEHRENS, 2010). Uma vez que a consulta é escrita e submetida para execução, o compilador inicia o processo de análise e transformação. A saída da compilação de uma consulta em AQL é um código fonte em HQL. O código HQL obtido é então repassado ao framework AOPJungle, que tratará de sua execução e consequente retorno dos resultados, para serem então apresentados. A interface com o usuário para visualizar o resultado das consultas não foi desenvolvida neste trabalho, sendo orientada para futuras implementações. O framework AOPJungle possui integrações com a ferramenta AJDT e com a API do *Workspace* para obter informações do código e para obter informações sobre os recursos da plataforma, tais como projetos, arquivos, caminhos e configurações.

4.2 O Framework AOPJungle

O AOPJungle é um framework desenvolvido, fundamentalmente, para executar a instanciação do metamodelo de programas escritos em AspectJ. A partir da representação do programa fonte como um modelo de objetos, o AOPJungle admite receber e executar consultas sobre este modelo. A Figura 4.2 apresenta a arquitetura do AOPJungle.

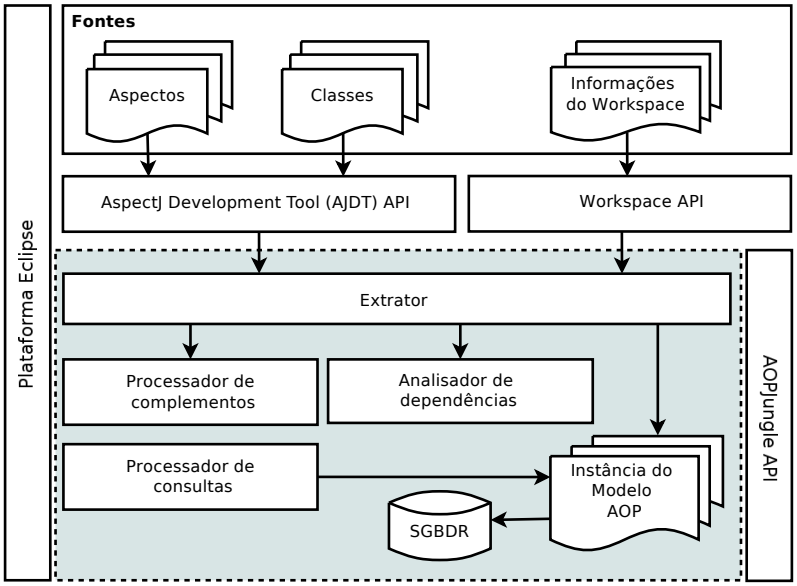


Figura 4.2 – Arquitetura do AOPJungle

4.2.1 Extrator

O módulo Extrator é o principal módulo do AOPJungle. Ele é responsável pela análise do código fonte de cada projeto do ambiente Eclipse e por instanciar o metamodelo. Para tanto, o Extrator utiliza duas fontes de informação: o *workspace* do Eclipse e a AST correspondente de cada unidade de compilação. No processo de análise de código fonte, o primeiro passo é identificar os projetos AspectJ/Java disponíveis no IDE. Isso é feito por meio da API do *workspace*, que fornece uma abstração do *workspace* raíz (`org.eclipse.core.resources.IWorkspaceRoot`) contendo uma lista dos recursos disponíveis, incluindo projetos, pastas e arquivos. A partir da lista de projetos, o Extrator utiliza a API do AJDT para obter os pacotes e as unidades de compilação. Cada unidade de compilação representa um arquivo contendo o código fonte do programa a ser analisado. Uma vez que uma unidade de compilação está disponível, é possível percorrer sua AST em busca das informações necessárias para a instanciação do metamodelo.

O módulo AJDT do Eclipse gera incrementalmente uma representação do programa em formato de AST, à medida que o programador está escrevendo seu código. Cada nó da AST fornece informações específicas sobre o objeto o qual está representando. Por exemplo, a classe `FieldDeclaration` possui informações sobre a declaração de atributos, a classe `MethodDeclaration` informações sobre métodos e a classe `SimpleName` representa qualquer String que não seja uma palavra reservada em Java/AspectJ. A Figura 4.3 ilustra um trecho

da AST gerada para o programa descrito na listagem 4.1. As setas indicam os nós e subnós correspondentes à declaração do atributo *field1*. Para o programa descrito, contendo a declaração de um atributo, e dois métodos (acesso e escrita do atributo), foram gerados quarenta e um nós na AST.

```
1  package br.ufsm.lab.core;
2  public class Class1 {
3        private String field1;
4
5        public String getField1() {
6              return field1;
7        }
8
9        public void setField1(String field1) {
10             this.field1 = field1;
11       }
12 }
```

Listagem 4.1 – Exemplo de código para ilustrar a geração da AST

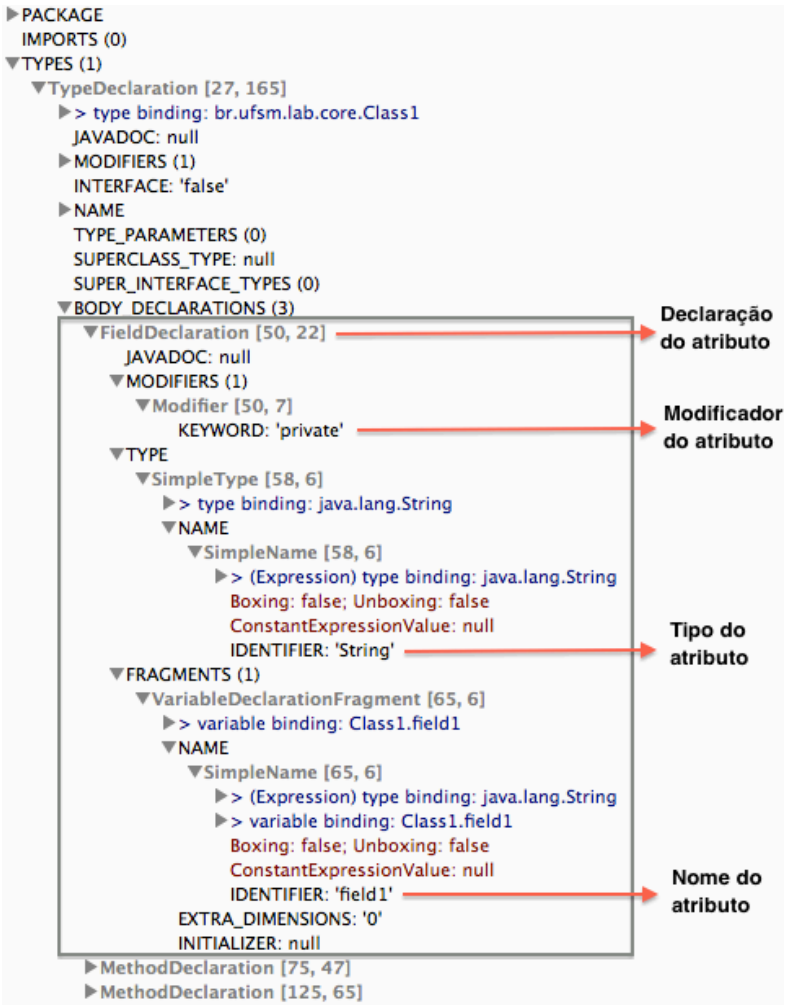


Figura 4.3 – Exemplo de AST

O AJDT fornece um mecanismo de acesso a ASTs para obter informações de cada nó

em uma árvore de maneira uniforme. Esse mecanismo é implementado por meio do padrão *Visitor* (VLISSIDES et al., 1995), o qual possibilita que a execução de uma determinada lógica seja realizada de forma separada dos elementos de uma estrutura de dados. Na prática, o padrão oferece um modo de adicionar novas operações sobre uma estrutura de objetos sem que essa estrutura seja alterada.

Para que a representação de uma AST seja acessada é necessário estender a classe abstrata `AjASTVisitor`, pertencente ao pacote `org.aspectj.org.eclipse.jdt.core.dom`. No AOPJungle, a classe responsável por essa implementação é a classe `AOJungleVisitor`, contida no pacote `br.ufsm.aopjungle.aspectj`. Os nós são visitados de cima para baixo, da esquerda para a direita, de acordo com sua hierarquia, até alcançar os nós folha. Ao visitar todos os nós filhos, o método `endVisit(...)` é invocado quando que não há mais nós filhos a serem visitados. A listagem 4.2 apresenta dois trechos de código que ilustram o início e fim da visita em um nó da AST. O método *visit* (linha 2) indica o início da visita ao nó representando a declaração de um adendo do tipo *before*. Na linha 8, o método *endVisit* é invocado quando não há mais nós filhos a visitar para o nó representando a declaração.

```
1   @Override
2   public boolean visit(BeforeAdviceDeclaration node) {
3       . . .
4       return super.visit(node);
5   }
6
7   @Override
8   public void endVisit(BeforeAdviceDeclaration node) {
9       . . .
10      super.endVisit(node);
11  }
```

Listagem 4.2 – Exemplo de código *Visitor*

Para garantir a correta contextualização dos elementos durante a instanciação do metamodelo, o Extrator implementa uma pilha para manter objetos que estão atualmente sendo visitados. O processo de adição e remoção de elementos da pilha ocorre durante a execução dos métodos `visit(...)` e `endVisit(...)` respectivamente.

4.2.2 Analisador de Dependências

O Analisador de dependências é o módulo responsável por construir as associações e as dependências entre os elementos do modelo. Uma vez que o módulo Extrator instanciou o metamodelo, o módulo de análise de dependências é chamado para acrescentar as informações de

ligação entre os elementos, tais como relações de adendos com os pontos de junção, declarações intertipos e modificações hierárquicas.

Uma característica importante durante a construção da AST é a possibilidade de incluir informações de ligação em seus nós, facilitando assim o entendimento das dependências entre os elementos do programa. Durante o desenvolvimento do analisador de dependências do AOPJungle, verificou-se que o analisador implementado pelo AJDT para AspectJ (`org.aspectj.org.eclipse.jdt.core.dom.ASTParser`) é bastante limitado em relação à resolução de ligações. De acordo com a documentação oficial do AspectJ em sua versão 1.7, o suporte a resolução de ligações de tipos na AST ainda é uma questão pendente[3]. Para que o modelo de ligações pudesse ser instanciado, o analisador de dependências utilizou recursos fornecidos diretamente das ASTs construídas pelo compilador de AspectJ. Dessa forma, o analisador de dependências instancia o modelo de ligação para cada elemento que afeta ou é afetado por outro elemento no programa.

4.2.3 Processador de Complementos

O processador de complementos é responsável por gerenciar os módulos que serão executados após a instanciação do metamodelo, incluindo, por exemplo, o cálculo de métricas e estatísticas de um programa escrito em AspectJ. Os complementos são módulos que oferecem serviços de pós processamento à instanciação do metamodelo, e podem ser adicionados por meio de seu registro no configurador de complementos do AOPJungle.

4.2.4 Processador de Consultas

Toda consulta realizada ao modelo criado pelo AOPJungle é recebida pelo módulo processador de consultas. O principal serviço disponível no processador de consultas é o serviço de execução de consultas. O módulo de processamento de consultas mantém um registro de executores de consultas identificado unicamente por um código denominado manipulador (*Handler*). Um cliente que deseja realizar uma consulta ao modelo deve indicar o manipulador do executor e o código fonte da consulta. Por exemplo para executar uma consulta em HQL ao modelo, a seguinte chamada é realizada:

```
1    aopjungle.query ('hql', 'consulta hql.... ');
```

[3] Referenciado em http://bugs.eclipse.org/bugs/show bug.cgi id=146528

4.3 O Compilador da AQL

O compilador da AQL foi implementado como um tradutor fonta a fonte, o qual recebe como entrada uma sequência de caracteres representando o código em AQL e como saída emite um conjunto de caracteres representando o código em HQL. O processo de compilação de AQL é realizado em 3 fases, sendo:

- Análise Estática: nessa etapa são realizadas as análises léxica, sintática e semântica da linguagem. O objetivo é identificar inconsistências no código AQL baseadas em suas regras sintáticas e semânticas.
- Tradução: refere-se à execução da transformação do modelo AQL em um modelo HQL.
- Emissão: representa a etapa final da compilação, a qual é realizada a transformação do modelo HQL em texto.

A principal ferramenta utilizada na construção do compilador foi o *workbench* Xtext. O Xtext possibilitou o gerenciamento de todo o ciclo de implementação do compilador, desde a fase de definição da sintaxe da AQL, passando pelo processo de tradução e, finalmente, pela geração do código em HQL. A Figura 4.4 apresenta a arquitetura do compilador AQL.

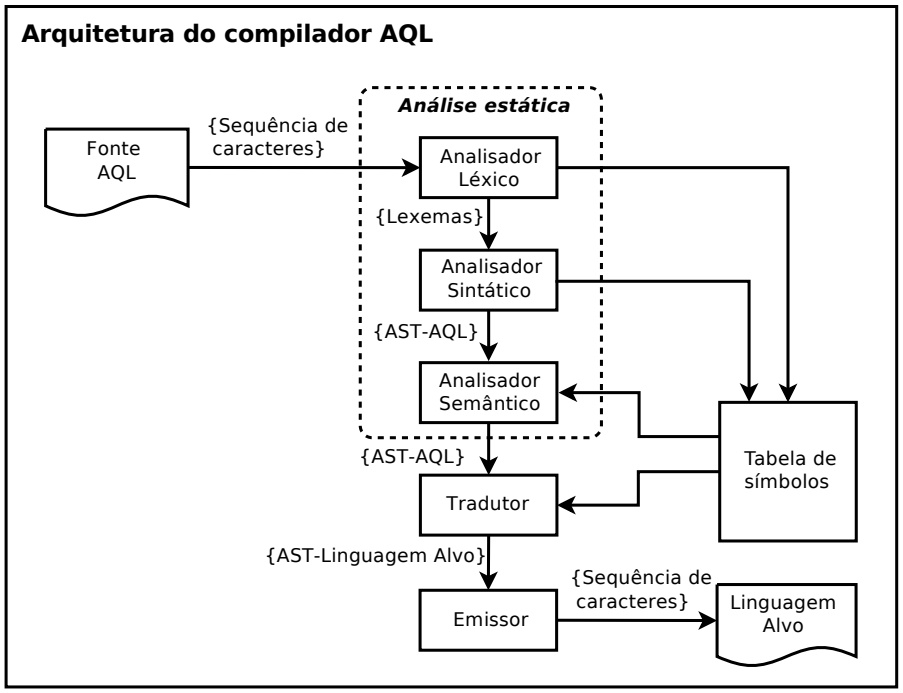


Figura 4.4 – Arquitetura do compilador AQL

Historicamente, o projeto do Xtext é estreitamente ligado ao projeto da linguagem Xtend, uma linguagem ortogonal a Java, desenvolvida com a proposta de facilitar a escrita de programas, por meio de novas funcionalidades, tais como extensões de métodos e expressões lambda. A linguagem Xtend foi a linguagem utilizada para a implementação do compilador AQL, principalmente por oferecer recursos que facilitam a geração de código, por meio de modelos de String (*String Templates*).

As seções seguintes descrevem os módulos do compilador AQL e apresenta discussões sobre decisões de implementação.

4.3.1 Definição da Sintaxe Concreta da AQL

Baseada na especificação proposta no Capítulo 3, a gramática da AQL foi criada utilizando uma metalinguagem fornecida pelo Xtext, baseada em regras EBNF. Tais regras são classificadas como regras terminais e regras não terminais[1].

As regras terminais representam símbolos atômicos utilizados na linguagem, como por exemplo a definição de um identificador, comentários ou um número inteiro, e são declaradas, em Xtext, com a palavra reservada `terminal`. Cada regra terminal retorna um valor atômico de um determinado tipo, baseado no metamodelo *Ecore* do *Eclipse Modeling Framework* (EMF) (GRONBACK, 2009). A listagem 4.3 apresenta as regras terminais descritas para a definição de identificadores (linha 1), números inteiros (linha 5) e números com ponto flutuante (linha 9). A gramática completa da AQL pode ser verificada no apêndice A.

```
1  terminal ID returns ecore::EString
2       : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_
           ')*
3       ;
4
5  terminal INT returns ecore::EInt
6       : '0'..'9'+
7       ;
8
9  terminal FLOAT returns ecore::EDouble
10      : ( ('0'..'9')+ ( '.' ('0'..'9')+ ( ('E'|'e') ('-')? ('0
           '..'9')+ )? )? )
11      ;
```

Listagem 4.3 – Definição de regras terminais em AQL

O Xtext deriva o modelo da AST da linguagem AQL a partir da descrição de sua gramá-

[1] As regras não terminais também são conhecidas como regras de análise ou produção (SEBESTA, 2009).

tica em EBNF[4]. À medida que a sequência de caracteres é consumida, o modelo é instanciado com os objetos representativos de cada elemento da linguagem.

Por padrão, o retorno de cada regra sintática é uma instância do tipo `ecore::EString` , podendo ser modificado por meio da cláusula `returns`. O reuso de regras terminais pode ser alcançado com o uso de fragmentos, os quais descrevem regras terminais que não devem ser consumidas pelo analisador léxico e devem ser utilizadas em outras regras terminais.

As regras não terminais, ou de análise, produzem uma árvore de *tokens* terminais e não terminais. Diferentemente das regras terminais, as regras de análise produzem árvores de análise (também referenciadas no Xtext como modelo de nós) as quais são computadas em um grafo de sintaxe abstrata ligada, ou em termos gerais, uma AST. A Listagem 4.4 mostra as regras de análise para a definição da cláusula *find* da linguagem AQL.

```
1  FindClause
2       : clause='find' bindingObject+=BindingObject (',' 
           bindingObject+=BindingObject)*
3       ;
4
5  BindingObject
6       : type=ObjectType alias+=ID+
7       ;
8
9  ObjectType
10      : {Project} value='project'
11      | {Package} value='package'
12      | {Class} value='class'
13      | {Aspect} value='aspect'
14      | {Interface} value='interface'
15      | {Enum} value='enum'
16      ;
```

Listagem 4.4 – Definição de regras para a cláusula find em AQL

A primeira regra, denominada *FindClause* (linha 1), descreve a palavra reservada *find* e uma lista de objetos retornados da subregra *BindingObject*. Por sua vez, a regra *BindingObject* (linha 5) é descrita como a subregra *ObjectType* seguida de uma lista de identificadores (ID) contendo ao menos um elemento. Por fim, a regra *ObjectType* (linha 9) é descrita por uma das palavras reservadas indicando objetos de ordem superior na pesquisa (project, package, class, etc.).

[4] Alternativamente, o Xtext possibilita também que um modelo pré-definido representando a linguagem seja especificado.

4.3.2 Análise Estática

As análises léxicas e sintáticas da AQL são realizadas por analisadores gerados automaticamente pelo Xtext, a partir da gramática formalmente descrita. O Xtext delega a função de geração dos analisadores léxico e sintático ao ANTLR (PARR, 2007), um programa que possibilita a geração de analisadores para serem utilizados, principalmente, na construção de compiladores e interpretadores. O processo de tradução da gramática para o formato reconhecido pelo ANTLR é realizado durante a geração dos artefatos de Xtext. A partir da gramática traduzida, o ANTLR gera os analisadores léxico e sintático correspondentes.

4.3.2.1 Análise Léxica e Sintática

O analisador léxico é responsável por coletar caracteres e agrupá-los de forma lógica, associando códigos internos aos agrupamentos, de acordo com sua estrutura (SEBESTA, 2009; AHO et al., 2006). Esses agrupamentos lógicos são denominados lexemas, enquanto os códigos internos para as categorias desses agrupamentos são chamados de *tokens*. Para ilustrar o processo de análise léxica da AQL, a Figura 4.5 mostra exemplos de lexemas e *tokens* gerados a partir uma cadeia de caracteres representando uma consulta simples.

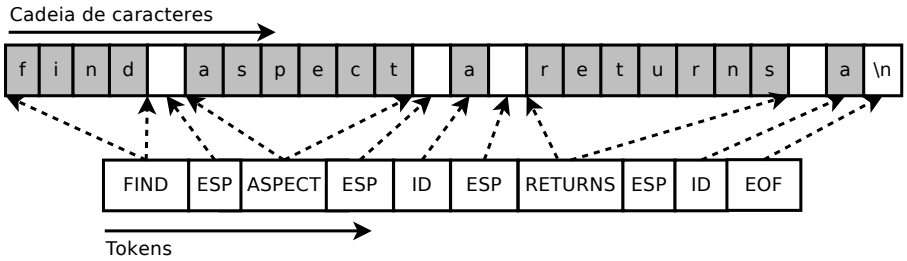


Figura 4.5 – Exemplos de lexemas e *tokens*

Considerando como entrada a cadeia de caracteres `find aspect a returns a`, o analisador léxico identifica cada um dos lexemas e o associa a seu respectivo *token*, de acordo com as regras gramaticais da linguagem. Os *tokens* são identificados por números inteiros, representados por constantes como *FIND* e *ID*, para facilitar a leitura e compreensão humana. A Tabela 4.1 apresenta os lexemas e *tokens* identificados.

Durante o processo de análise léxica, alguns lexemas são, em geral, ignorados, como por exemplo espaços em branco (ESP), comentários e marcas de final de arquivo (EOF), embora o analisador gerado pelo ANTLR possa mapear esses lexemas em diferentes estruturas,

Tabela 4.1 – Relação entre sequência de caracteres, lexemas e *tokens*

Token	Lexema	Descrição
FIND	find	Palavra chave
ASPECT	aspect	Palavra chave
RETURNS	returns	Palavra chave
ID	a, a	Identificador

denominadas canais reservados.

O ANTLR trabalha na geração de analisadores léxicos e sintáticos a partir da definição de uma única gramática descrevendo regras terminais e regras de análise. O processo de análise sintática tem como entrada a série de *tokens* produzida pelo analisador léxico. O objetivo principal do analisador sintático é garantir que a série de *tokens* está coerente com as regras da linguagem. À medida que a análise sintática ocorre, o modelo representando a AST é instanciado e permanece disponível para outras fases de análise, como a análise semântica e o processo de transformação. A Figura 4.6 mostra a AST instanciada para a consulta apresentada anteriormente, na Figura 4.5.

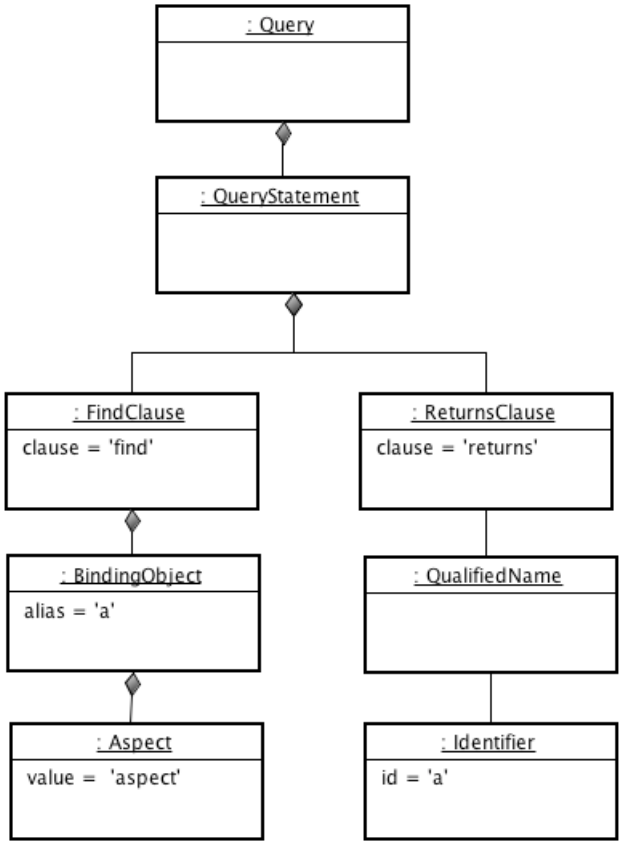


Figura 4.6 – AST instanciada de acordo com o modelo *Ecore*

Os objetos do tipo *Query* representam o início da AST enquanto os objetos do tipo

QueryStatement representam o início das instruções da consulta em AQL. Ambos pertencem ao grupo de tipos que não possuem atributos, representando por si só a semântica do nó na AST. À medida que as consultas se tornam maiores, com mais instruções, a AST, da mesma forma, se torna mais densa, tornando o processo de transformação gradualmente complexo.

4.3.2.2 Análise Semântica

Semanticamente, o código de AQL é verificado em uma fase subsequente à análise sintática, desde que nenhuma inconsistência seja identificada, o que ocasiona a interrupção no processo de compilação. A análise semântica é parte da estrutura de validações customizadas do Xtext, as quais podem ser criadas mediante a instanciação de uma classe descendente do tipo `AbstractDeclarativeValidator` do pacote `org.eclipse.xtext.validation`. O processo de análise semântica da AQL envolveu basicamente duas áreas: análise de referências cruzadas e análise de funções internas, conforme descritas a seguir:

- **Referências cruzadas:** A análise de referências cruzadas, ou ligação, é responsável por garantir que os identificadores sejam corretamente declarados antes de sua utilização. Mesmo a linguagem AQL sendo declarativa, as referências ao uso de pseudônimos da cláusula `find` necessitam ser verificadas, pois são obrigatórias para qualquer consulta. Por exemplo, um nome qualificado `a.modifier(...)` definido na cláusula `where` necessita do objeto definido por `a` declarado na cláusula `find`. Uma vez que uma referência não seja devidamente encontrada, o compilador aborta sua execução e acrescenta uma mensagem correspondente à lista de erros. A listagem 4.5 apresenta uma consulta na qual uma regra semântica de AQL é violada. Conforme pode-se observar, o pseudônimo `b` referenciado na cláusula `returns` não possui declaração na cláusula `find`.

```
1 find aspect a
2 where a.pointcut.name = 'Foo' and a.modifier(public)
3 returns b.name, b.pointcut.isAnonymous, b.pointcut.code
```

Listagem 4.5 – Exemplo de uma cláusula inválida em AQL

- **Funções internas:** Toda função contextualizada em uma consulta AQL é mapeada como uma função interna, a qual possui um conjunto de regras associadas que devem ser observadas pelo analisador semântico. Essas regras são mantidas em um arquivo descritor (XML), e são carregadas durante o processo de análise semântica, de acordo com a quantidade e o tipo dos parâmetros especificados. O analisador semântico também verifica se

a função está corretamente contextualizada, ou seja, se o objeto e a função associada são compatíveis. A listagem 4.6 mostra um trecho do arquivo descritor contendo a definição da função `modifier`.

```
1  <aql-function>
2     <internal>
3        <function name="modifier">
4           <contexts>
5              <context name=AOJAspectDeclaration/>
6              <context name=AOJClassDeclaration/>
7              <context name=AOJInterfaceDeclaration/>
8              <context name=AOJMethodDeclaration/>
9              . . .
10          </contexts>
11          <params>
12             <param name=kind type=List value=public, protected,
                  private, abstract, final, strictfp, native,
                  synchronized, transient, volatile></param>
13          </params>
14          <return type=boolean/>
15       </function>
16       . . .
17    </internal>
18 . . .
19 </aql-function>
```

Listagem 4.6 – Trecho do arquivo descritor de funções da AQL

Em relação à verificação estática de tipos, a atual implementação da AQL delega à linguagem alvo qualquer tarefa relativa à verificação de tipos em expressões contidas na consulta. Essa decisão não impede que futuras implementações possam considerar a verificação de tipos como parte do processo de validação semântica, principalmente se o código alvo não prover um sistema para tal função. Para a implementação atual da AQL, a consulta é traduzida e enviada ao motor de execução da linguagem alvo, a qual, ao detectar uma violação semântica na verificação de tipos, retorna o erro correspondente à AQL.

4.3.3 Processo de Transformação

O principal objetivo do processo de transformação é a criação de um modelo HQL (destino) a partir de um modelo AQL (fonte), obtido como resultado da fase de análise da consulta. Durante a fase de transformação, uma nova instância do modelo HQL é criada a partir do modelo AQL. Esse processo é realizado por meio da aplicação de regras de transformação entre os modelos, as quais definem padrões de entrada e saída. O mecanismo de transformação procura por padrões de entrada no modelo AQL e aplica os padrões de saída no modelo HQL. Ao final

Tabela 4.2 – Exemplo de código entre as fases do processo de transformação

Fase	Código

Entrada (AQL)	`find aspect a`
	`where a.modifier (public,protected)`
	`returns a.name, a.modifier.name`

Fase 1 (Intermediário)	`select a.name, a.modifier.name`
	`from AOJAspectDeclaration a`
	`where a.modifier(public,protected)`

Fase 2 (HQL)	`select a.name, aojmodifier_2`
	`from AOJAspectDeclaration a`
	`left join a.modifiers aojmodifier_1`
	`inner join aojmodifier_1.descriptors aojmodifier_2`
	`where aojmodifier_2 in ( 'PUBLIC', 'PROTECTED' )`

de todas transformações, a AST do código em HQL deve ser produzida.

Os modelos AQL e HQL são modelos instanciados a partir do metamodelo *Ecore*, embora sejam modelos distintos. Durante a transformação, algumas construções no modelo de origem possuem equivalência semântica direta no modelo destino. São transformações uniformes, ou seja, um padrão identificado na origem leva à criação de um padrão idêntico no destino. Por exemplo, a cláusula `where` tem equivalência semântica direta na cláusula `where` em HQL. Por outro lado, algumas construções demandam a criação de múltiplos padrões no destino. Por exemplo, a tradução de funções como `modifier(...)` e `kind(...)` implica na geração de múltiplas junções (construção `join`) na cláusula `from` para que a consulta em HQL seja realizada.

O processo de transformação é realizado em duas fases: na primeira fase, o mecanismo de transformação busca pelos padrões de origem no modelo AQL e instancia os padrões de destino no modelo HQL, de acordo com as regras de transformação especificadas. A segunda fase é responsável pelas transformações dos nomes qualificados. Observa-se, portanto, que dependendo das construções utilizadas na consulta, a fase um pode resultar em um modelo intermediário, o qual será interpretado e resolvido pela fase dois. Para ilustrar a transformação de modelos entre as fases, a Tabela 4.2 apresenta exemplos de código resultante em cada fase. Observa-se, porém, que os códigos mostrados nas fases 1 e 2 são ilustrativos, sendo que a produção dos mesmos ocorre somente na fase de emissão do código, conforme descrito na Subseção 4.3.8.

No exemplo mostrado na Tabela 4.2, o código de entrada em AQL é transformado na

fase 1 em um código intermediário, sem a resolução de nomes qualificados, como `a.name` e `a.modifier(public,protected)`. Conforme mencionado, essa transformação é realizada na segunda fase do processo.

As seções seguintes descrevem o mecanismo de transformação e detalha as regras aplicadas nas fases 1 e 2 do processo de transformação.

4.3.4 Mecanismo de Transformação

A transformação entre modelos baseia-se no princípio de que existam pré-condições, pós-condições e invariantes que devam ser satisfeitas antes e após o processo de transformação do modelo. Nesse sentido, um programa pode ser expresso como um grafo orientado atributivo tipado, no qual os vértices representam os elementos do programa e as arestas representam as relações entre esses elementos. Cada aresta pode ser dirigida e nomeada representando uma relação única entre dois elementos e os vértices podem conter atributos que os identificam unicamente.

Formalmente, o processo de transformação de modelos baseado em grafos é definido como um sistema de transformação de grafos atributivos $GTRS = (ATG, P)$, consistindo de um grafo atributivo tipado ATG e um conjunto de regras de transformações P, sendo $p = (ATG_S \overset{s}{\leftarrow} ATG \overset{t}{\rightarrow} ATG_T) \in P$. O grafo de sintaxe abstrata de um modelo fonte pode ser especificado por um conjunto de todos (ou um subconjunto) de grafos de instâncias sobre um grafo do tipo ATG_S. De forma análoga, o grafo de sintaxe abstrata de um modelo destino pode ser especificado por um conjunto de todos (ou um subconjunto) de grafos de instâncias sobre um grafo de tipo ATG_T. Ambos grafos ATG_S e ATG_T devem ser subgrafos do grafo atributivo tipado ATG.

O início da transformação com a instância do grafo AG_S tipado como ATG_S, também é tipado como ATG. Uma sequência de transformações $AG_S \overset{*}{\Rightarrow} AG_T$ entre as instâncias AG_S e AG_T denota que AG_S é isomórfico em relação a AG_T ou existe uma sequência de $n \geq 1$ de transformações diretas

$$AG_S = AG_0 \overset{p_1}{\Rightarrow} AG_1 \overset{p_2}{\Rightarrow} \ldots \overset{p_n}{\Rightarrow} AG_n = AG_T$$

Durante o processo de transformação, os grafos intermediários são tipados como ATG e, relações, atributos e tipos adicionais podem ser utilizados como recursos temporários à tradução. O grafo resultante é um grafo do tipo AG_T, ou seja, é um grafo sintaticamente correto.

Durante o processo de transformação os tipos de dados são não alterados. A Figura 4.7 ilustra o processo de transformação de modelos associados a tipos.

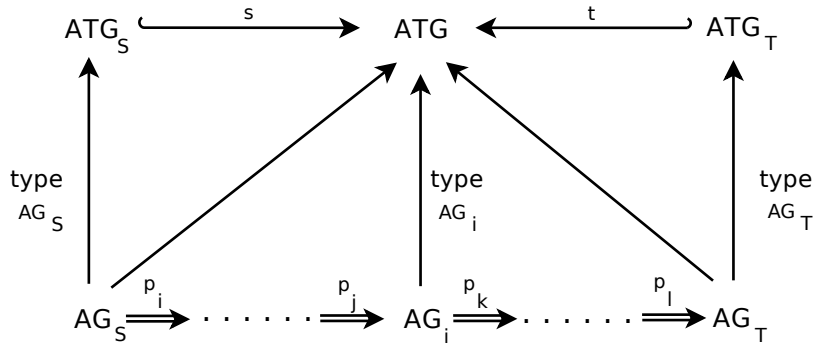


Figura 4.7 – Tipagens no processo de transformação de modelos

Uma regra de transformação de grafo $r : L \rightarrow R$ consiste de dois grafos L, R tipados em ATG, os quais a união $L \cup R$ é definida. O lado esquerdo L representa as pré-condições da regra, enquanto o lado direito R as pós-condições da regra. A união entre L e R forma novamente um grafo compatível com a fonte, o destino e a configuração de tipos. $L \cap R$ representa a parte do grafo que necessita existir para a transformação ser realizada, mas que não são alteradas. Já a relação $L \setminus (L \cap R)$ configura a parte a ser excluída e $R \setminus (L \cap R)$ a parte a ser criada. O primeiro passo no início da transformação do grafo é encontrar uma combinação m do lado esquerdo L na instância atual do grafo AG, a qual seja compatível com o tipo e mantenha sua estrutura preservada. Em seguida todos vértices e arestas as quais são combinados por $L \setminus (L \cap R)$ são removidos de AG. A estrutura restante $D := AG \setminus m(L \setminus (L \cap R))$ permanece um grafo válido, ou seja, sem arestas pendentes. Por fim, o grafo intermediário D é colado a $R \setminus (L \cap R)$ para obter um grafo derivado AG_n ou ainda um grafo final AG_T.

A partir da representação dos modelos AQL e HQL como grafos orientados, foi possível especificar as regras de transformação, as quais são alinhadas basicamente com os seguintes conceitos de reescrita de grafos:

- Pré-condição (*Left Hand Side Rule - LHS*),
- Pós-condição (*Right Hand Side Rule - RHS*),
- Condições de aplicação negativa (*Negative Application Condition* - NAC)
- Condições de aplicação positiva (*Positive Application Condition* - PAC).

A pré-condição (LHS) representa um subgrafo do grafo de origem que será utilizado para descrever os padrões candidatos a transformação. A pós-condição (RHS) são os resultados pretendidos, obtidos pela manutenção, exclusão ou geração de novos elementos. As NACs são restrições aos cenários pretendidos na transformação do grafo. Uma NAC representa uma condição a qual a transformação somente ocorre quando a mesma for falsa, mesmo na presença de uma padrão candidato a transformação. Já uma PAC é aplicada em uma pós condição, ou seja, somente ocorrerá a transformação, caso uma condição no modelo de saída estiver presente.

4.3.5 Transformações em EMorF

Para apoiar a implementação das transformações entre os modelos, uma formalização das regras de transformação foi realizada com o uso do framework EMorF (KLASSEN; WAGNER, 2012), o qual baseia-se na transformação algébrica de grafos (ROZENBERG; EHRIG, 1999) e possui recursos para a transformação de modelos EMF.

Em EMorF, os vértices são representados por objetos tipados e as arestas são representadas por referências orientadas entre os vértices. As pré-condições e pós-condições são representadas por rótulos nos objetos, os quais são pré-definidos como: «bind», «delete» e «create». O rótulo «bind» realiza o casamento do objeto e preserva sua instância durante o processo de transformação. O rótulo «create» cria um novo objeto e o associa a outros objetos. O rótulo «delete» remove um objeto existente ou uma referência. Os vértices e arestas da LHS podem ser identificados pelos rótulos «bind» ou «delete». Na porção RHS, os rótulos «bind» ou «create» podem ser utilizados. As NACs e PACs são descritas por expressões OCL (*Object Constraint Language*) rotuladas como «constraint». Por fim, as atribuições são realizadas também por meio de expressões OCL, porém identificadas com o rótulo «assignment».

O framework EMorF permite a especificação de regras de transformação tanto entre modelos distintos com rastreabilidade (abordagem DPO - *double-pushout approach* (ROZENBERG; EHRIG, 1999)), quanto de um modelo único (abordagem SPO - *single-pushout approach* (ROZENBERG; EHRIG, 1999)). Na abordagem DPO, o grafo é apresentado em três colunas. A coluna da esquerda, nomeada *source-domain*, representa os vértices e arestas do modelo origem (AQL). A coluna da esquerda, nomeada *target-domain*, representa os vértices e arestas do modelo destino (HQL). A coluna central (*corr-domain*) contém elementos de rastreabilidade para as regras de transformação, e são de uso da linguagem EMorF. Já na abordagem SPO, não há divisão entre origem e destino e as regras de reescrita são descritas conforme apresentadas

anteriormente («bind», «delete» e «create»).

4.3.6 Regras de Transformação

Dadas as instâncias AG_S e AG_T, de grafos tipados em AQL[5] e HQL[6], respectivamente, sendo AQL o tipo origem e HQL o tipo destino, o conjunto de regras de transformações P que gera o grafo AG_T a partir do grafo AG_S é particionado em duas categorias, pelo mecanismo de transformação: transformações diretas $(1:1)$ e transformações múltiplas $(1:N)$.

4.3.6.1 Regras de Transformação $1:1$

A linguagem AQL apresenta elementos que possuem sintaxe e semântica próximas ou idênticas em HQL, nos quais o processo de transformação ocorre na proporção $1:1$, ou seja, a transformação é realizada de forma direta, com a criação do elemento correspondente. Essas regras são aplicadas na fase 1 do processo de transformação e correspondem a cláusula *where*, *group by*, *order by*, *having* e operadores pertencentes às expressões. A Figura 4.8 ilustra a transformação da cláusula *where* utilizando o framework EMorF. Na porção RHS, os vértices *s-where* e *s-wheren1* tipados em *WhereClause* e *Expression* de AQL, respectivamente, dão origem ao vértice *t-where* (LHS) tipado em *WhereClause* em HQL. A Tabela 4.3 apresenta as categorias e instruções de origem (AQL) e de destino (HQL) em cada transformação $1:1$ aplicada pelo compilador AQL.

4.3.6.2 Regras de Transformação $1:N$

As regras de transformação $1:N$ resultam em múltiplas construções (vértices e arestas) no destino a partir de uma construção na origem. São aplicadas nas cláusulas *find* e *returns*, e em nomes qualificados de uma consulta AQL. Conforme já mencionado, a transformação das cláusulas *find* e *returns* ocorre ainda na fase um do processo de transformação, enquanto que os nomes qualificados são resolvidos na fase dois.

A especificação da transformação da cláusula *find* é realizada a partir da combinação de três vértices tipados como *FindClause*, *BindingObject* e *SymbolTable*, todos com rótulos *bind*, conforme pode ser observado no grafo representado na Figura 4.9. Observa-se

[5] O modelo AQL é gerado a partir da definição da gramática da linguagem em EBNF

[6] O modelo HQL utilizado foi baseado na versão 3.6 do Hibernate e está disponível em http://sourceforge.net/projects/hibernate/files/hibernate3/3.6.10.Final/

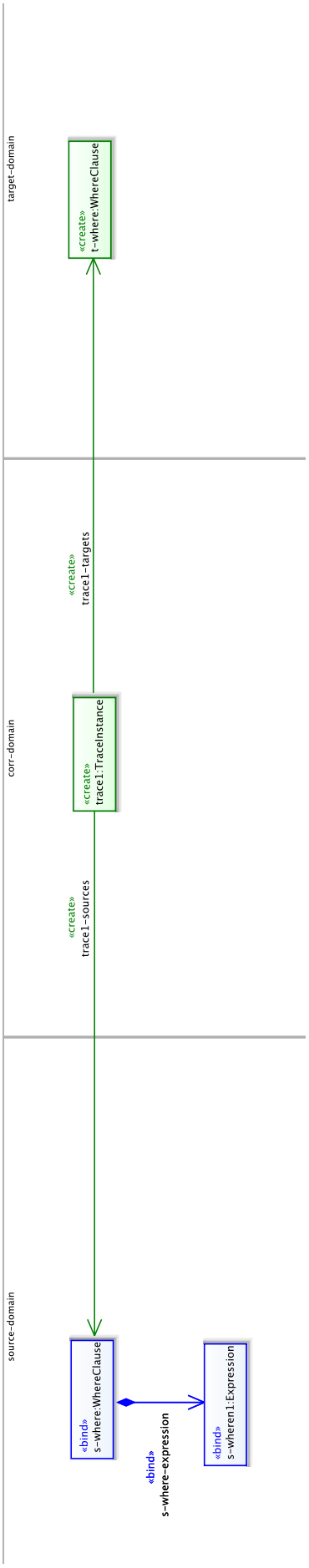


Figura 4.8 – Grafo de transformação da cláusula *where*

Tabela 4.3 – Transformações 1:1 (AQL para HQL)

Categoria	Instrução AQL	Instrução HQL
Operadores aritméticos	$+$	$+$
	$-$	$-$
	$*$	$*$
	$/$	$/$
Operadores relacionais	$>$	$>$
	$<$	$<$
	$>=$	$>=$
	$<=$	$<=$
	$=$	$=$
	$!=$	$<>$
	$like$	$like$
Operadores lógicos	and	and
	or	or
	not	not
Agrupamento	( )	( )
Quantificados	in	in
	$not\ in$	$not\ in$
Cláusula where	$where$	$where$
Cláusula order by	$order\ by$	$order\ by$
Cláusula group by	$group\ by$	$group\ by$
Cláusula having	$having$	$having$
Cláusula asc	asc	asc
Cláusula desc	$desc$	$desc$

que o vértice *from* é criado a partir do vértice *find*, e, para cada *findn1* do tipo *BindingObject* é criado um vértice *fromn1* de tipo *ObjectSource*. Essa condição é descrita na atribuição `fromn1.alias ->asSet()= findn1.alias ->asSet()`, na qual o conjunto descrito em `fromn1.alias -> asSet()` é atribuído ao conjunto `findn1.alias -> asSet()`. Já na atribuição `from1.className= symbolTable.getType(findn1.type.value)`, resolve-se o nome da classe de acordo com o tipo associado a cláusula *find* (*aspect, class, enum, interface, etc*) a partir do método `getType()` contido na tabela de símbolos instanciada.

A especificação da transformação da cláusula *returns* é realizada a partir da combinação da cláusula de tipo `ReturnsClause`, uma lista de tipo `StringLiteral` e uma lista de descendentes de tipo `Expression`. A cláusula *returns* dá origem a cláusula de tipo `SelectClause` no modelo destino, e é associada a uma lista de expressões e a uma lista de pseudônimos (*alias*). A Figura 4.10 apresenta as regras de transformação da cláusula *returns*. As expressões são transformadas conforme as regras $1:1$ apresentadas na Seção 4.3.6.1.

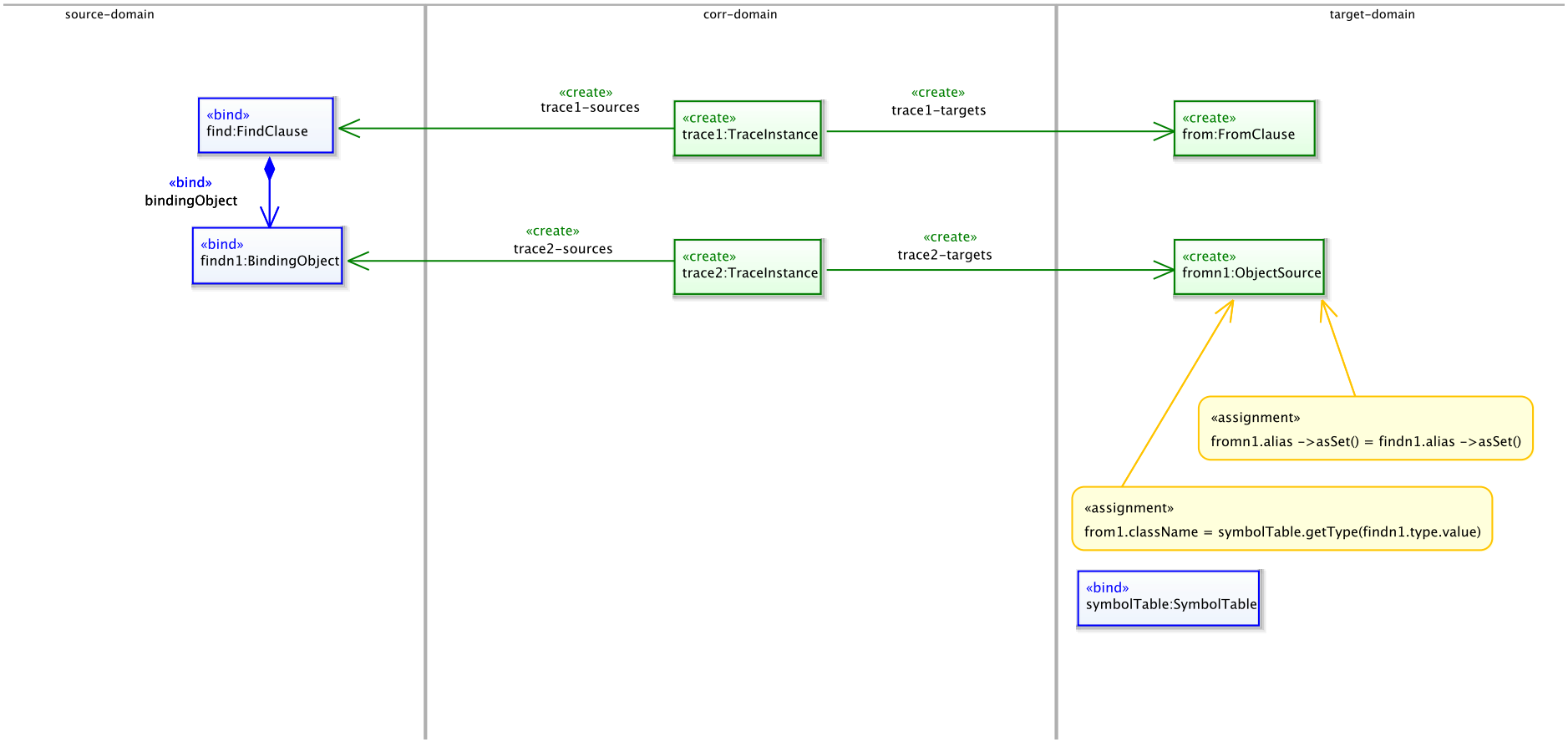


Figura 4.9 – Grafo de transformação da cláusula *find*

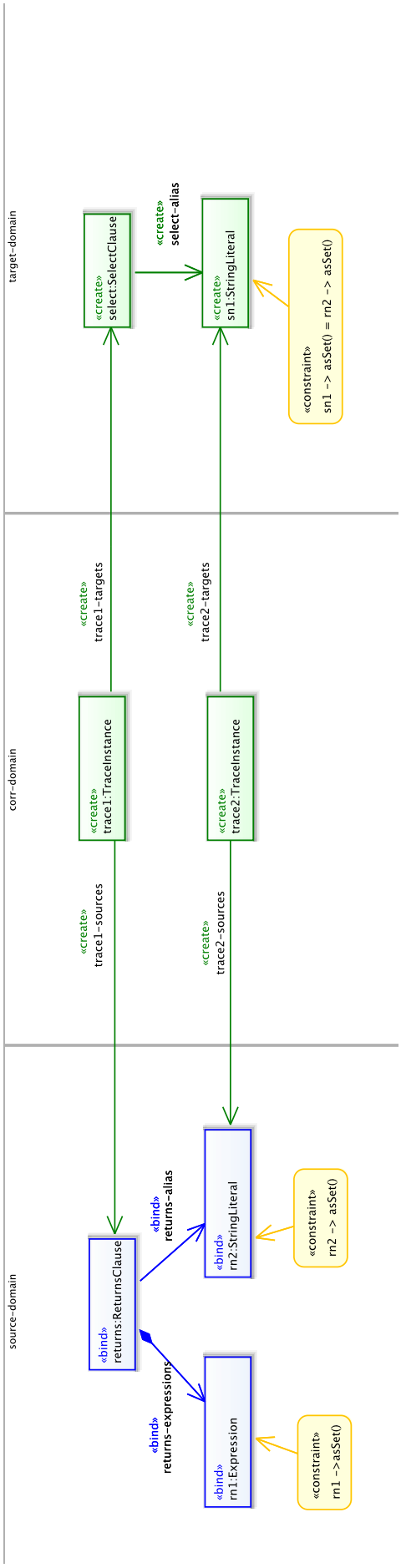


Figura 4.10 – Grafo de transformação da cláusula *returns*

4.3.7 Regras de Transformação de Nomes Qualificados

Os nomes qualificados são resolvidos durante a segunda fase do processo de transformação, sobre o grafo intermediário tipado em HQL, gerado durante a fase um do processo de transformação. O mecanismo de transformação processa os nomes qualificados a partir da invocação dinâmica de regras de transformação, de acordo com um mapa de módulos de transformação, consultado durante o processo de compilação.

À medida que os objetos e as funções de um nome qualificado são analisados, o compilador procura por um módulo para a execução da transformação baseado no contexto e no nome do objeto ou da função analisada. O exemplo a seguir ilustra a transformação de um nome qualificado. A consulta apresentada na Listagem 4.7 retorna o nome dos aspectos e seus respectivos códigos de pontos de corte, para pontos de corte nomeados que contenham primitivas do tipo *call*. Na Listagem 4.8 é apresentado um trecho do arquivo de mapeamentos de módulos de transformação.

```
1 find aspect a
2 where a.pointcut.kind(call)
3 returns a.name, a.pointcut.code
```

Listagem 4.7 – Exemplo de consulta AQL

```
1  <aojql-mapping>
2     <bindings>
3        ...
4
5        <bind in="project" out="AOJProject"/>
6        <bind in="aspect" out="AOJAspectDeclaration"/>
7        <bind in="class" out="AOJClassDeclaration"/>
8        <bind in="package" out="AOJPackageDeclaration"/>
9        <bind in="interface" out="AOJInterfaceDeclaration"/>
10       <bind in="enum" out="AOJEnumDeclaration"/>
11
12       ...
13
14       <bind in="AOJAspectDeclaration.pointcut" out="br.ufsm.hql.
            transformation.HqlPointcutFunction"/>
15       <bind in="AOJAspectDeclaration.pointcut.kind" out="br.ufsm.hql.
            transformation.HqlPointcutKindFunction"/>
16
17    </bindings>
18 </aojql-mapping>
```

Listagem 4.8 – Trecho do arquivo de mapeamento de módulos de transformação

A cláusula *where* da consulta em AQL (linha 2 da Listagem 4.7) apresenta o nome

qualificado *a.pointcut.kind(call)*.O elemento *a* é resolvido como um símbolo pela tabela de símbolos (cláusula `find aspect a`). O objeto do tipo *aspect* é resolvido para a classe *AOJAspectDeclaration*, conforme os atributos *in - out* do mapeamento (linha 6 da Listagem 4.8). O elemento *pointcut* é resolvido para *br.ufsm.hql.transformation.HqlPointcutFunction* (linha 14). Essa resolução é feita pela busca pelo contexto *AOJAspectDeclaration.pointcut* no arquivo de mapeamento (linha 14). Por fim, o elemento *kind* é uma função resolvida para o módulo *br.ufsm.hql.transformation.HqlPointcutKindFunction* (linha 15).

A atual versão de AQL implementa cerca de cinquenta transformações entre funções e objetos qualificados, de acordo com o metamodelo de AspectJ proposto. A seguir, são apresentadas as regras de transformações de duas funções (*Affects* e *Kind*), escolhidas aleatoriamente, e de seus objetos, como referência às regras de transformação aplicadas. Será dada maior ênfase nas transformações das cláusulas *from* e *where*, visto que as outras cláusulas foram resolvidas na primeira fase da transformação.

4.3.7.1 Função *Affects(...)*

As Figuras 4.11 e 4.12 mostram, respectivamente, as regras de transformação das cláusulas *from* e *where* para a função *affects(...)*.

Na Figura 4.11, o LHS, identificado pelos vértices de rótulo «bind» e «delete», inclui o vértice tipado em *Function* de nome *affects*, o qual é assegurado pelo PAC *wn11.name = 'affects'*. O vértice tipado em *AliasManager* é um tipo auxiliar para gerar pseudônimos únicos, e é utilizado em todas as transformações de nomes qualificados. Os vértices *fn11* e *fn12*, tipados em *ObjectSource* representam o contexto e o parâmetro da função *affects* respectivamente, na cláusula *from*. Por exemplo, no fragmento de consulta

```
... where a.affects(c)...,
```

o objeto *a* seria representado por *fn11* e *c* por *fn12*. O RHS da transformação é representado por dois vértices (*fn2* e *fn21*) e três arestas (*from-ObjectList2*, *fn2-left* e *fn2-right*).

O vértice *fn21* possui duas atribuições, sendo que o atributo *alias* recebe um novo pseudônimo dado por *fn21.alias = aliasMgm.getNextAlias('affects')* e o atributo *className* recebe uma string de ligação, dada por *fn21.className = fn11.alias + '.bindingModel.outBinding'*. A string de ligação é uma string que conecta objetos de acordo com o metamodelo de AspectJ empregado.

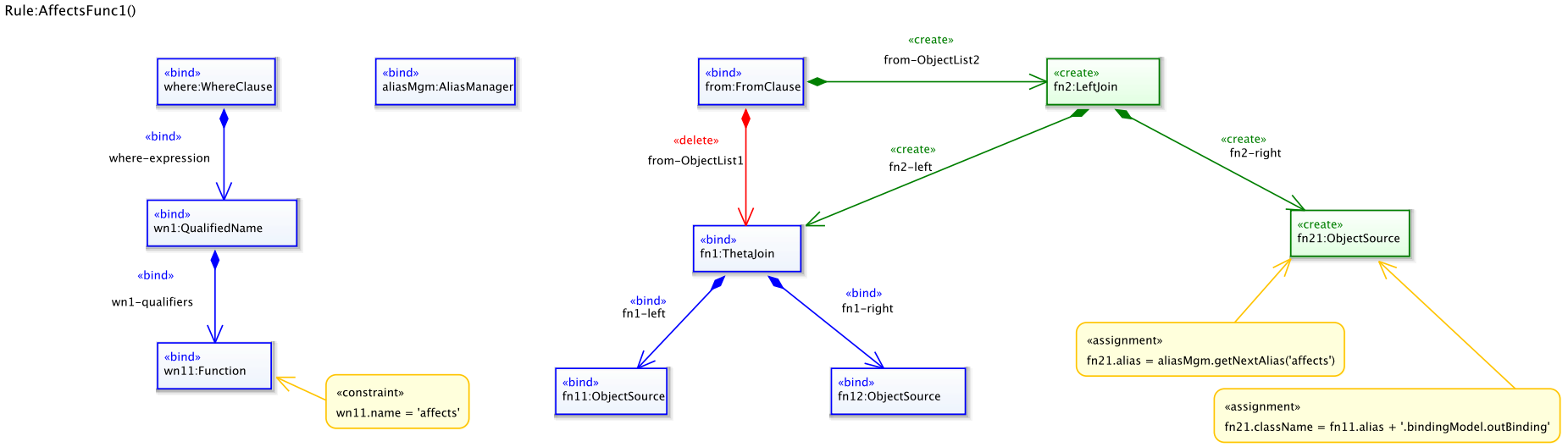


Figura 4.11 – Grafo de transformação da cláusula *from* da função *affects(. . .)*

Na Figura 4.12, é apresentado o conjunto de regras de transformação para a cláusula *where* da função *affects*. A condição de combinação é a mesma criada para a transformação da cláusula *from*, contendo um vértice tipado em *Function* e um PAC que assegura o nome da função sendo *affects*. O vértice tipado em *Equals* (*wn2*) é criado para representar uma expressão binária de igualdade, e, a partir dele, os vértices tipados em *QualifiedName* (*wn21*, *wn22*) e *Identifier* (*wn211*, *wn212*, *wn221*, *wn222*) são construídos.

Os identificadores *wn211* e *wn221* recebem, no atributo *id*, dois valores passados como parâmetros para o conjunto de regras: *bindingLink* e *targetAlias*. Esses parâmetros podem ser vistos no canto superior esquerdo da Figura 4.12, juntamente com o nome da regra de transformação (*AffectsFunc2*). O parâmetro *bindingLink* refere-se ao pseudônimo criado na junção *left* da cláusula *from* (*fn21.alias*) na etapa anterior (Figura 4.11), e o parâmetro *targetAlias* refere-se ao pseudônimo do objeto de ligação passado como parâmetro na função *affects*.

Por exemplo, considerando-se o fragmento de consulta anterior (*... where a.affects(c) ...*), tendo *aojbinding_1* como o valor hipotético do parâmetro *bindingLink* e *c* como o valor do parâmetro *targetAlias*, o trecho da cláusula *where* transformada resultante será:

```
... where aojbinding_1.id = c.id ...,
```

Na fase final da transformação da cláusula *where*, os vértices e arestas correspondentes a função *affects* são eliminados, visto que não fazem sentido na linguagem alvo. Dessa forma, o grafo resultante contém elementos que resultam em uma consulta equivalente na linguagem alvo, de acordo com um metamodelo proposto nessa implementação.

4.3.7.2 Função *Kind(...)*

A seguir são apresentadas as regras de transformação para a função *kind* no contexto de um adendo. As Figuras 4.13 e 4.14 apresentam, respectivamente, as transformações relativas às cláusulas *from* e *where*.

Na Figura 4.13, o PAC *wn11.name = 'kind'* garante a existência da função *kind* na cláusula *where*. A função *kind* necessita de duas junções (*fn2* e *fn21*) na resolução de equivalência HQL , e cada uma delas possuem associações com vértice tipados em *ObjectSource*. Os vértices *fn22* e *fn212* recebem nos atributos *alias* pseudônimos gerados por *aliasMgm*. O atributo *className* de *fn212* recebe o valor *AOJAdviceDeclaration* de acordo com associações do metamodelo e de forma similar, *fn22.className* recebe o valor *fn1.alias + '.members.allAdvices'*.

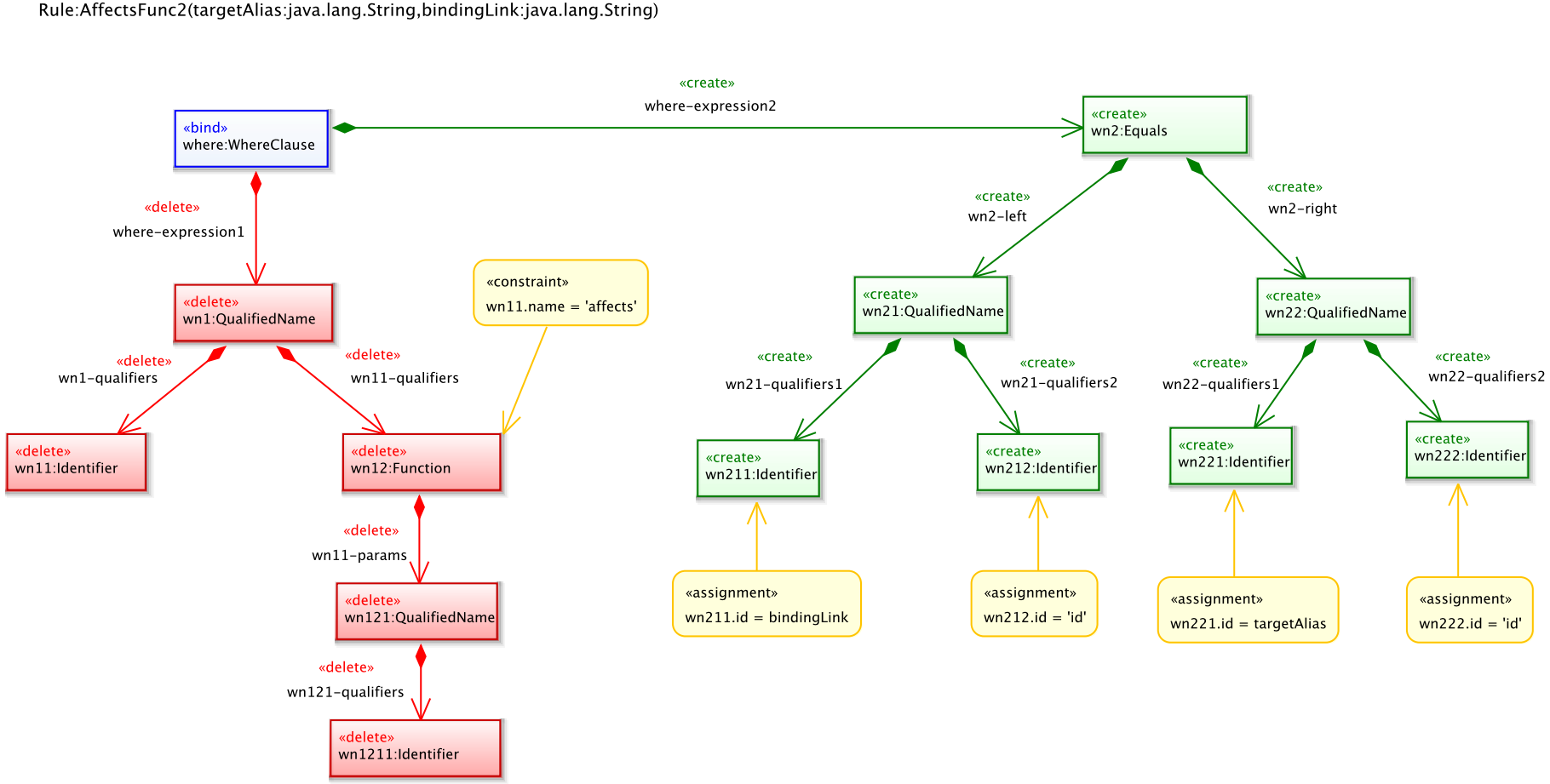


Figura 4.12 – Grafo de transformação da função *affects(. . .)* - *Cláusula where*

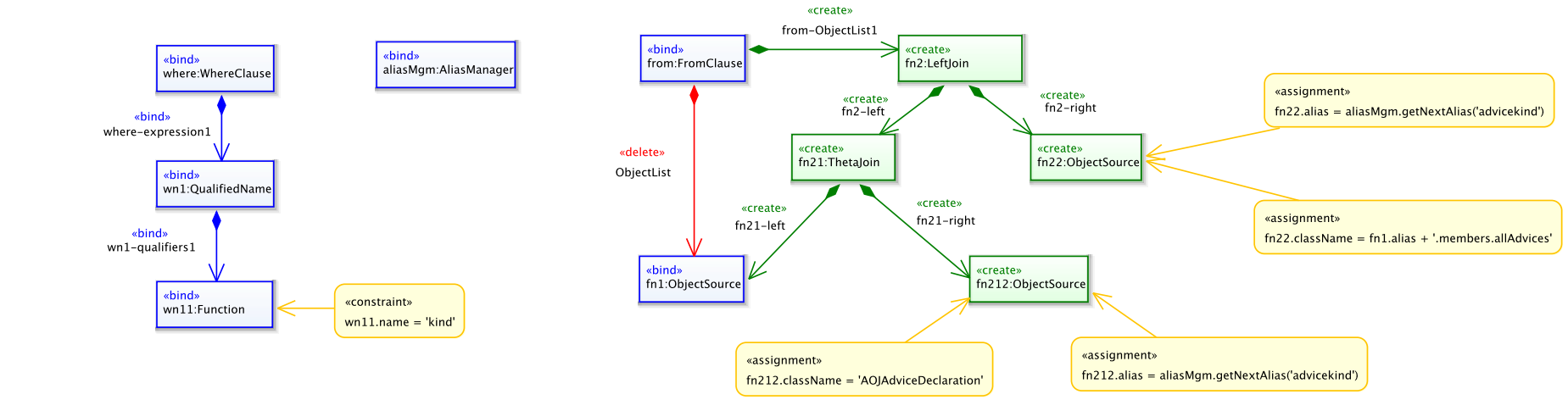


Figura 4.13 – Grafo de transformação da cláusula *from* da função *kind(. . .)*

A associação inicial *ObjectList* do vértice *from* com *fn1* é excluída e uma nova associação é criada com o vértice *fn21* tipado em *ThetaJoin*.

Para exemplificar a transformação da cláusula *from* na função *kind*, no contexto de adendos, o seguinte trecho de código é utilizado:

```
... where a.advice.kind(before)...,
```

No trecho de código anterior, o objeto *a* representa um pseudônimo para um aspecto declarado na cláusula *find* e é representado pelo vértice *fn1*, criado na fase 1 do processo de transformação. Nesse caso, o valor de *fn22.className* é *a.members.allAdvices*. Supondo-se que *aliasMgm* tenha gerado *aojadvice_1* e *aojadvice_2* para *fn212.alias* e *fn22.alias* respectivamente, o fragmento de código correspondente ao trecho de código anterior é o seguinte:

```
... from AOJAspectDeclaration a, AOJAdviceDeclaration
aojadvice_1 LEFT JOIN a.members.allAdvices aojadvice_2 ...,
```

A Figura 4.14 apresenta o conjunto de regras de transformação para a cláusula *where* da função *kind*. A condição de combinação é a mesma criada para a transformação da cláusula *from*, anteriormente apresentada. A função *kind* é basicamente substituída na cláusula *where* por um conectivo lógico *And*, sendo que o lado esquerdo da expressão é formado por uma expressão de igualdade (vértice tipado *Equals*) e o lado direito por uma expressão do tipo *in* (vértice tipado *InClause*). O vértice tipado em *ParsExpression* representa expressões com parênteses.

O conjunto de regras de transformação para a cláusula *where* da função *kind* recebe dois parâmetros de entrada: *adviceKindLink1* e *adviceKindLink2*, sendo utilizados nos vértices *wn2111*, *wn2211* e *wn2121*, como valor do atributo *id*. Ao final da transformação, os vértices e arestas relativos à função *kind* são eliminados do grafo, conforme assinalados com o rótulo «delete».

Para ilustrar a transformação da cláusula *where* da função *kind*, o mesmo exemplo trecho de código anterior aplicado será utilizado (`... where a.advice.kind(before)...`). Considerando-se que os parâmetros recebidos em *adviceKindLink1* e *adviceKindLink2* sejam os valores hipotéticos *aojadvice_1* e *aojadvice_2*, tem-se *wn2111.id = 'aojadvice_1'*, *wn2211.id = 'aojadvice_1'* e *wn2121.id = 'aojadvice_2'*. O atributo *wn2221.value* assume o valor do parâmetro passado na função *kind* (*before*, no atual exemplo). Após a aplicação das associações e das regras de transformação, o seguinte resultado é esperado na geração do código:

Rule:KindFunc2(adviceKindLink1:java.lang.String,adviceKindLink2:java.lang.String)

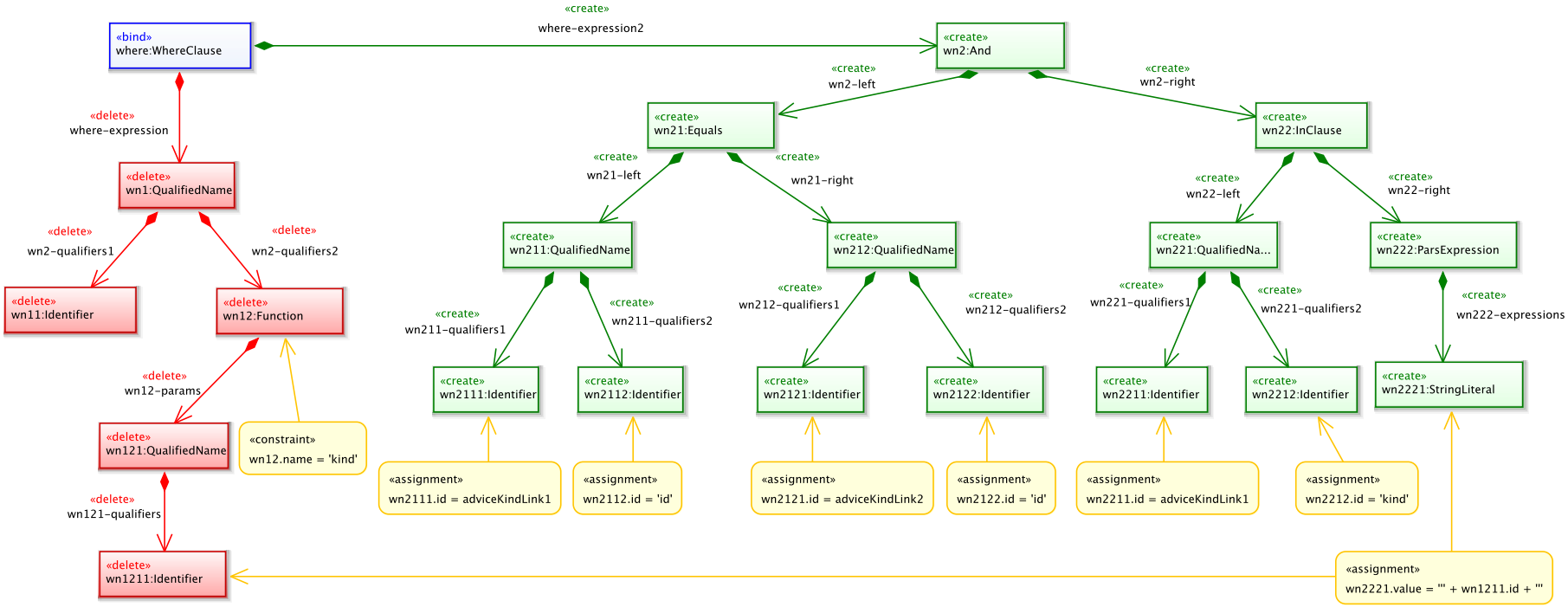


Figura 4.14 – Grafo de transformação da cláusula *where* da função *kind(. . .)*

```
... where aojadvice_1.id = aojadvice_2.id and aojadvice_1.kind
                        in ('before')...
```

4.3.8 Emissão de Código

A emissão do código é a última etapa realizada pelo compilador e consiste na produção do código da linguagem alvo a partir de sua AST. Esse processo é relativamente simples com o uso do Xtext e plenamente apoiado pela linguagem Xtend, com o uso de expressões de modelos. Utilizando-se a técnica de expressões de modelo é possível implementar diferentes saídas de código para a mesma AST, simplesmente por meio da substituição de um modelo por outro. O ANTLR utiliza técnica similar para a geração dos analisadores léxicos e sintáticos, a qual pode ser realizada em diferentes linguagens, tais como as linguagens C, Java e C#, a partir de uma configuração apropriada.

O módulo responsável pela geração do código é o emissor. O emissor recebe, como entrada, a AST da linguagem alvo e percorre seus nós e subnós, identificando cada tipo e seu modelo correspondente dentre as expressões de modelo. A inspeção da AST é iniciada pelo primeiro nó da árvore (raiz), sendo percorrida de cima para baixo, da esquerda para a direita, até o último nó folha. A inspeção de um nó pode determinar a emissão de um trecho de código ou pode somente apontar pela inspeção de seus subnós. Por exemplo, os nós do tipo `Query` e `QueryStatement` são nós que não resultam em geração de código, pois indicam, conforme citado anteriormente, o início da AST e da consulta respectivamente.

A Listagem 4.9 mostra o método que inicia o processo de inspeção da AST no emissor. O emissor utiliza o recurso de invocação polimórfica de métodos do Xtend para direcionar a geração correta de código, de acordo com os tipos de nós da AST. O início e fim de um modelo em Xtend é marcado por três símbolos de apóstrofes simples (”’). A interpolação de texto estático com código executável é realizada por meio dos caracteres *guillemets* (« »).

```
1  def dispatch compile(hql.QueryStatement query) '''
2       «query.selectClause.compile»
3       «query.fromClause.compile»
4       «IF (query.whereClause != null)»
5          «query.whereClause.compile»
6       «ENDIF»
7       «IF (query.orderByClause != null)»
8          «query.orderByClause.compile»
9       «ENDIF»
10
11      ...
12 '''
```

Listagem 4.9 – Inspeção do nó `hql.QueryStatement` pelo emissor

A listagem 4.10 apresenta o código para emissão da cláusula `SELECT` de HQL. O uso de expressões interpoladas pode ser observado na linha 7, onde o texto representando a cláusula `as` é mesclado à lógica envolvida na emissão da cláusula.

```
1   def dispatch compile(hql.SelectClause select) '''
2      «select.clause.toUpperCase»
3      «clear_i»
4      «FOR b:select.expressions SEPARATOR ','»
5         «b.compile»
6         «IF (i < select.alias.size)»
7            as «select.alias.get(i)»
8            «inc_i»
9         «ENDIF»
10     «ENDFOR»
11    '''
```

Listagem 4.10 – Emissão da cláusula SELECT da HQL

4.3.9 Avaliação

A fim de avaliar a implementação da linguagem AQL, duas abordagens foram exploradas: *(i)* a execução de um conjunto de testes funcionais em diferentes sistemas escritos em AspectJ. Esses testes visaram encontrar defeitos no compilador, seja durante as fases de análise sintática/semântica ou durante processo de transformação, e *(ii)* um estudo de caso com o uso da linguagem AQL em uma ferramenta de identificação de oportunidades de refatoração em sistemas orientado a aspectos. Esse estudo de caso é apresentado no Capítulo 5.

A execução dos testes funcionais foi agrupada de acordo com as construções apresentadas pela linguagem AQL, tais como cláusulas básicas (*find, where, returns, order by, group by*, expressões, etc.) e testes de transformação para nomes qualificados. Para cada teste executado, o resultado da transformação, ou seja, o código HQL, foi comparado ao resultado esperado e,

então, executado em um conjunto de sistemas escritos em AspectJ. Por meio de inspeção visual, os dados obtidos resultantes da consulta foram então comparados com os dados extraídos de cada programa. Os resultados foram considerados corretos quando o código gerado pela transformação correspondeu ao código esperado e consequentemente pôde ser executado como uma consulta HQL válida e equivalente.

Além dos testes com cenários positivos, um conjunto de testes para exercitar o fluxo de exceção (testes negativos) também foi executado. Os testes com cenários negativos foram configurados tendo consultas inválidas em AQL como entrada, e, como saída, foi avaliada a interrupção do processo de compilação com a sinalização de exceção correspondente. Os testes com consultas inválidas foram realizados a fim de encontrar possíveis defeitos durante as análises sintáticas e semânticas do compilador. Para apoiar o processo de teste, o framework JUnit (HUNT; THOMAS, 2003) foi utilizado. O conjunto completo de casos de teste está disponíveis para consulta na página do projeto (AQL, 2013).

Outras duas análises permitiram avaliar o desempenho do processo de compilação e o tamanho das consultas AQL em comparação ao código correlato em HQL. Em relação ao desempenho, a compilação de consultas é executada na ordem de milissegundos[7], sendo a menor consulta contendo 5 lexemas[8] e a maior contendo 34 lexemas. Já a análise de tamanho das consultas foi realizada comparando-se a quantidade de lexemas entre cada consulta em AQL e sua correspondente em AQL. Em média, as consultas em AQL apresentaram uma redução de 38% na quantidade de lexemas em relação à mesma consulta em HQL, para uma amostra de 115 casos de teste. A Figura 4.15 apresenta a distribuição de lexemas AQL em comparação a HQL.

O aumento de lexemas resultantes em HQL pode ser observado, principalmente, pela ausência de construções de junção na linguagem AQL, as quais são inferidas pelo compilador durante o processo de transformação entre as linguagens. Contudo, uma consulta simples, em AQL, tal como `find aspect a returns a` possui equivalência numérica de lexemas em relação a sua correspondente em HQL (`SELECT a from AOJAspectDeclaration a`). De fato, essa consulta pode ser ainda simplificada, resultando na seguinte construção: `from AOJAspectDeclaration`.

Dessa forma, observa-se que em consultas as quais a inferência de junção não se faz necessária, os códigos AQL e HQL se mostram similares em relação à quantidade de lexemas.

[7] Os testes foram realizados em notebooks MacPro, Intel i5 com 4GB RAM

[8] A contagem de lexemas não considerou pontuações, tais como vírgulas, parênteses, aspas, etc.

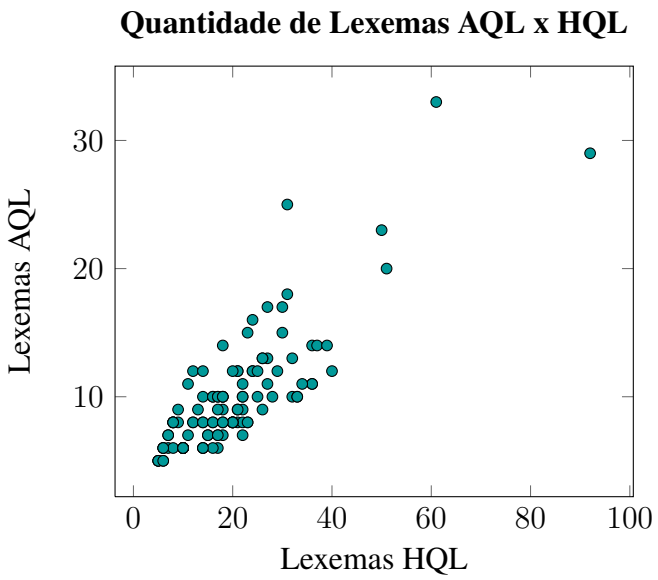


Figura 4.15 – Distribuição da quantidade de lexemas AQL x HQL

Por outro lado, em consultas as quais a inferência de junção ocorre, uma consulta em AQL mostra-se mais concisa. Por exemplo, a consulta em AQL apresentada na Listagem 4.11 é gerada com 29 lexemas, enquanto a sua correspondente em HQL, apresentada na Listagem 4.12 necessita 62 lexemas.

```
1  find aspect a1 a2
2     where (a1.advice.pointcut.code = a2.advice.pointcut.code)
3     and (a1.advice.position < a2.advice.position)
4     and (a1.fullQualifiedName = a2.fullQualifiedName)
5     and (a1.pointcut.isAnonymous = false)
6     and (a1.advice.kind = a2.advice.kind)
7     returns a1.fullQualifiedName, a1.advice.position, a2.advice.position,
         a1.advice.pointcut.code
```

Listagem 4.11 – Consulta AQL

```
1  SELECT a1.fullQualifiedName, aojadvice_1.position, aojadvice_3.position,
       aojadvicepointcut_1.code
2  FROM AOJAspectDeclaration a1, AOJAspectDeclaration a2,
       AOJAdviceDeclaration aojadvice_1
3  LEFT JOIN a1.members.allAdvices aojadvice_2, AOJAdviceDeclaration
       aojadvice_3
4  LEFT JOIN a2.members.allAdvices aojadvice_4
5  LEFT JOIN aojadvice_1.pointcutExpression aojadvicepointcut_1
6  LEFT JOIN aojadvice_3.pointcutExpression aojadvicepointcut_2
7  LEFT JOIN a1.members.pointcuts aojpointcuts_1
8  WHERE aojadvice_3.id = aojadvice_4.id
9  AND aojadvice_1.id = aojadvice_2.id
10 AND ( aojadvicepointcut_1.code = aojadvicepointcut_2.code )
11 AND ( aojadvice_1.position < aojadvice_3.position )
12 AND ( a1.fullQualifiedName = a2.fullQualifiedName )
13 AND ( aojpointcuts_1.isAnonymous = false )
14 AND ( aojadvice_1.kind = aojadvice_3.kind )
```

Listagem 4.12 – Consulta HQL

A Figura 4.16 apresenta o histograma representando o porcentual de redução de lexemas em consultas AQL em relação a consultas HQL. Conforme pode-se observar, para uma amostragem de 115 consultas, a maioria apresenta uma redução de lexemas na ordem de 40% a 60%.

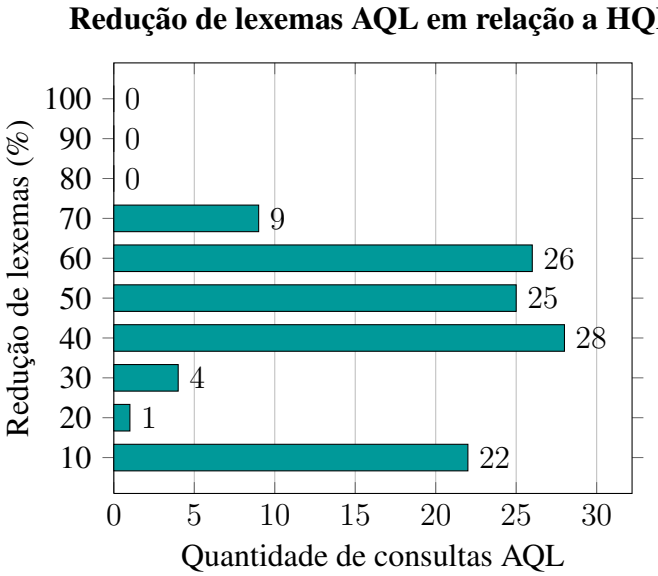


Figura 4.16 – Histograma de redução de lexemas AQL em relação a HQL

4.4 Considerações Finais

Este capítulo apresentou uma implementação de referência da linguagem AQL. A abordagem adotada implementou um modelo de representação de programas escritos em AspectJ e um compilador responsável por traduzir o código AQL para a linguagem HQL. Para obter informações de um programa, foi desenvolvido um framework chamado AOPJungle, que obtém referências estruturais e informações de dependências entre elementos do programa. O compilador foi implementado com apoio do *workbench* Xtext, e cada módulo de sua arquitetura foi descrito, desde a análise léxica, passando pelo processo de transformação de modelos, até a emissão do código. O próximo capítulo apresenta um estudo de caso utilizando a AQL para identificação de oportunidades de refatoração em código orientado a aspectos.

5 ESTUDO DE CASO

Este capítulo apresenta um estudo de caso envolvendo o uso da linguagem AQL como apoio no processo de identificação de oportunidades de refatoração em código orientado a aspectos. O estudo foi conduzido a partir de dados empíricos coletados em sete projetos de código aberto, os quais foram analisados de acordo com cenários candidatos à refatoração, catalogados na literatura. Para automatizar a identificação desses cenários, a ferramenta ASTOR (*Aspect Tool for refactoring*) foi implementada. Este capítulo está organizado da seguinte forma: a Seção 5.1 apresenta a metodologia aplicada e os projetos selecionados para o estudo de caso. A Seção 5.2 descreve os detalhes da implementação e da operação da ferramenta ASTOR. A Seção 5.3 apresenta cada cenário candidato à refatoração e as correspondentes consultas AQL de apoio à identificação desses cenários. Por fim, a Seção 5.4 apresenta uma discussão acerca do uso da linguagem AQL em identificação de oportunidades de refatoração.

5.1 Metodologia e Projetos Selecionados

A fim de investigar o uso da linguagem AQL integrada a uma ferramenta de análise estática, foram analisadas uma variedade de linhas de código fonte escritas de AspectJ. Foram selecionados dados empíricos a partir de sete projetos coletados em repositórios de código aberto, considerando o número de usuários e os diferentes domínios de aplicação. A coleta de projetos heterogêneos, com diferentes tamanhos e aplicações, proporcionou uma maior variedade nas construções em AspectJ e, portanto, uma maior cobertura na avaliação proposta. A Tabela 5.1 apresenta um resumo contendo nome, descrição, tamanho e URL dos projetos selecionados.

A obtenção da métrica *locc*, bem como a extração das estatísticas de cada projeto, foi realizada com instruções AQL, a partir do framework AOPJungle, o qual mantém essas informações em diferentes níveis de agrupamento. Por exemplo, a métrica *locc* é mantida desde o nível global (soma de todos projetos analisados), até o nível mais inferior, representado por um método ou adendo. O cálculo de *locc* é realizado de acordo com as seguintes diretivas (PIVETA, 2009):

- Comentários, javadocs e linhas em branco não são contadas;
- Cabeçalhos de classes, aspectos, interfaces, enumerações, métodos e adendos são contados como uma linha;

Nome do projeto	Descrição	Versão (locc)
1.JMule	Um cliente para redes eDonkey2000	V 0.5.8 (61365)
http://sourceforge.net/projects/jmule/		
2.Spacewar (AspectJ Examples)	Uma versão orientada a aspectos do Jogo Spacewar (Pacote de exemplos do AspectJ)	V AJ5 (1475)
http://www.eclipse.org/aspectj/		
3.AOPJungle	Framework de análise estática em programas AspectJ	V 0.1 (6329)
http://www.ufsm.br/ppgi		
4.AOPTetris	Uma versão orientada a aspectos do Jogo Tetris	V 0.1 (786)
http://www.guzzzt.com/coding/aspect.shtml		
5.AJHotDraw	Uma versão orientada a aspectos do framework gráfico JHotDraw	V 0.4 (20001)
http://sourceforge.net/projects/ajhotdraw/		
6.JLDAPLib	Uma biblioteca Java para acesso LDAP	V 1.0.1 (3635)
http://sourceforge.net/projects/jldaplib/		
7.XPlanner AspectJ	Uma versão orientada a aspectos da aplicação de gerenciamento de projetos XPlanner	V AspectJ (53427)
http://www.ohloh.net/p/xplanneraspectj		

Tabela 5.1 – Resumo dos projetos selecionados

- Uma nova linha é criada para o símbolo de fechamento de escopo (});
- Constantes String são contadas como uma linha simples.

A Listagem 5.1 apresenta a consulta para obter a métrica *locc* dos projetos selecionados.

```
find project p
returns p.name, p.metrics.numberOfLines
```

Listagem 5.1 – Consulta AQL para obter a métrica *locc* em projetos AspectJ

A Tabela 5.2 apresenta as informações para orientação acerca do tamanho dos projetos selecionados. As colunas #NCLA, #NASP, #NINT, #NENU, %NASP correspondem respectivamente ao número de declarações de classes, de aspectos, de interfaces, de enumerações e o percentual de declarações de aspectos em relações a todos tipos declarados.

De forma similar à obtenção da métrica *locc*, as informações estatísticas de cada projeto foram extraídas a partir de uma consulta AQL, conforme apresentada na Listagem 5.2.

Nome do projeto	# NCLA	# NASP	# NINT	# NENU	% NASP
JMule	482	29	117	50	4.28%
Spacewar	22	9	3	0	26.47%
AOPJungle	108	7	16	6	5.11%
AOPTetris	8	12	3	1	50.00%
AJHotDraw	253	31	52	0	9.23%
JLDAPLib	64	9	5	3	11.11%
XPlanner	686	10	47	0	1.35%

Tabela 5.2 – Módulos dos projetos selecionados

```
find project p
returns p.name as ProjectName, p.metrics.numberOfClasses as NCLA,
        p.metrics.numberOfAspects as NASP,
        p.metrics.numberOfInterfaces as NINT,
        p.metrics.numberOfEnums as NENU,
        cast( ((p.metrics.numberOfAspects /
        (p.metrics.numberOfClasses + p.metrics.numberOfAspects +
        p.metrics.numberOfInterfaces +
        p.metrics.numberOfEnums)) * 100),
        big_decimal) as PercASP
```

Listagem 5.2 – Consulta AQL para obter estatísticas de projetos AspectJ

5.2 A Ferramenta ASTOR (*Aspect Tool for Refactoring*)

A identificação de oportunidades de refatoração tem sido amplamente explorada em sistemas orientados a objetos (FOWLER, 1999; EMDEN; MOONEN, 2002; OLBRICH et al., 2009; SCHUMACHER et al., 2010) e, mais recentemente, em sistemas orientados a aspectos (PIVETA et al., 2006; MACIA BERTRAN; GARCIA; STAA, 2011; HUANG et al., 2011). Tais estudos resultaram em catálogos de refatorações, nos quais cada oportunidade é associada a um nome, a uma descrição de quando ocorre, a um exemplo e a um ou mais padrões de transformação (KERIEVSKY, 2004). As oportunidades de refatoração representam pontos em um código onde possivelmente encontram-se problemas de manutenção mais profundos (FOWLER, 1999).

O cenário *Lazy Aspect*, por exemplo, pode ser identificado quando aspectos não carregam o peso total de suas responsabilidades e delegam essa carga a classes por meio de declarações intertipos. Outro sintoma é a ocorrência de muitos campos e métodos, embora haja poucos módulos afetados por adendos no aspecto. Muitos dos cenários encontrados em programas orientados a aspectos são exemplos recorrentes de cenários encontrados em programas orientados a objetos, com algumas adaptações. Contudo, alguns cenários mais recentes (MON-

TEIRO; FERNANDES, 2005; MACIA BERTRAN; GARCIA; STAA, 2011) foram catalogados em função das novas construções encontradas em aspectos.

Para a realização do estudo de caso com uso da linguagem AQL, dez cenários representando oportunidades de refatoração (MONTEIRO; FERNANDES, 2005; PIVETA, 2009) (*Abstract Method Inter-Type Declaration*, *Anonymous Pointcut Definition*, *Code Duplication*, *Divergent Changes*, *Feature Envy*, *Large Aspect*, *Large Pointcut Definition*, *Lazy Aspect*, *Speculative Generality*, *Double Personality*) foram considerados para serem identificados. A fim de avaliar a capacidade da linguagem AQL em auxiliar no reconhecimento desses cenários, foi proposta a implementação da ferramenta ASTOR (*Aspect Tool for Refactoring*). A ferramenta foi desenvolvida como um *plug-in* do ambiente Eclipse e possibilita identificar cenários positivos para refatoração em código escrito em AspectJ, ou seja, oportunidades as quais a necessidade por refatorações seja mais provável. O processo de identificação de oportunidades é executado em três fases pela ferramenta ASTOR: *(i)* busca de cenários ou elementos do programa, *(ii)* processamento dos elementos do programa e *(iii)* apresentação dos resultados.

A Figura 5.1 ilustra a execução da ferramenta ASTOR por meio de duas visões. A primeira (visão 1), apresenta a estrutura do programa e suas dependências de forma hierárquica. Essa visão é basicamente a visão do modelo mantido em memória pelo AOPJungle. A segunda visão (visão 2) apresenta o resultado da análise do código em relação às oportunidades abordadas.

5.3 Identificando Oportunidades de Refatoração com AQL

Esta seção apresenta as consultas realizadas para auxiliar na identificação de cada uma das dez oportunidades de refatoração selecionadas para análise. Para cada cenário, são apresentadas as possíveis formas de identificá-lo no código, a consulta proposta para auxiliar em sua identificação e o resultado obtido pela ferramenta ASTOR para cada projeto analisado. Alguns cálculos dependem de parâmetros que são informados à ferramenta antes de sua execução. Foram considerados parâmetros e limiares com valores hipotéticos para avaliar o uso da ferramenta, sendo que alterações nessa configuração podem ocasionar um conjunto diferente de resultados. Esses parâmetros e limiares são apresentados em cada cenário analisado, onde forem requeridos.

1. *Abstract Method Inter-type Declaration*: Esse cenário pode ser reconhecido pela simples

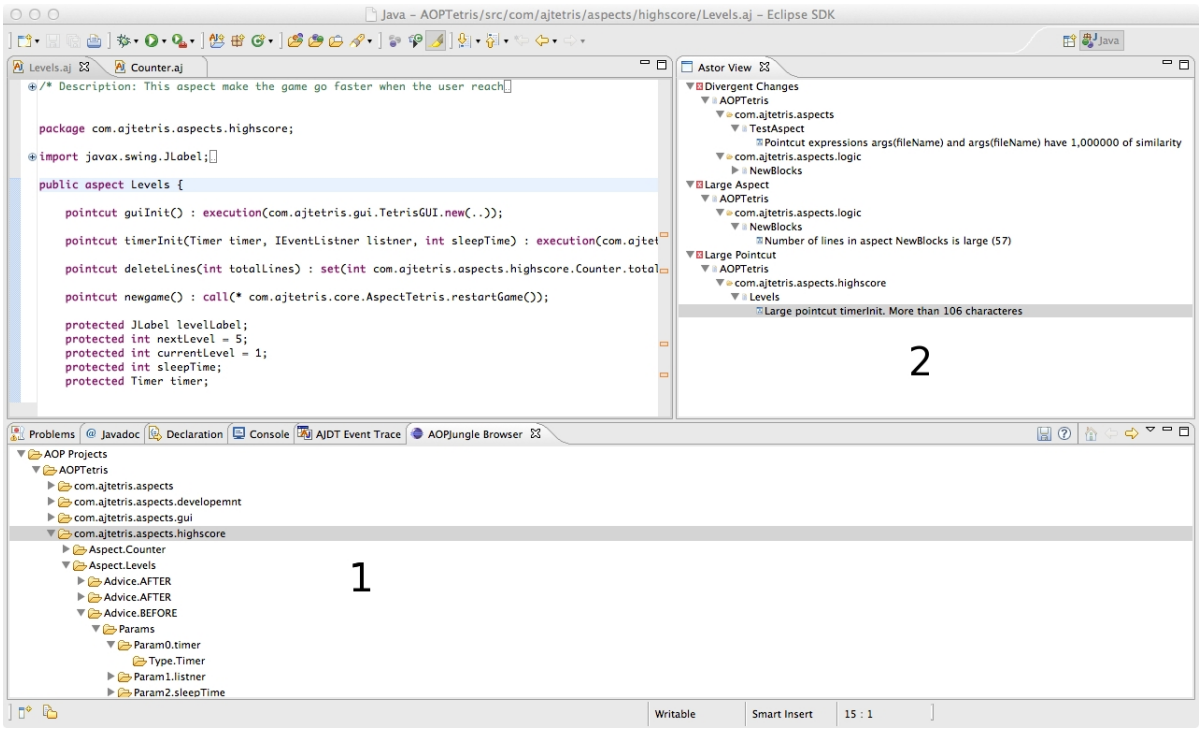


Figura 5.1 – Ferramenta ASTOR

existência de declarações intertipos de métodos abstratos no código. A introdução de um método abstrato implica na sua implementação em cada subclasse de classes afetadas pelo aspecto. A identificação deste cenário pode ser realizada por uma única consulta AQL, a qual busca pela introdução de métodos com modificador *abstract*, retornando o nome qualificado do aspecto, o nome do método introduzido e seus modificadores declarados. A Figura 5.2 apresenta os resultados para a identificação do cenário *Abstract Method Inter-type Declaration* nos projetos analisados.

```
1       find aspect a
2       where a.intermethod.modifier(abstract)
3       returns a.fullQualifiedName, a.intermethod.name,
4             a.intermethod.modifier.name
```

2. *Anonymous Pointcut Definition* : A identificação desse cenário ocorre quando existem pontos de corte anônimos associados a adendos. A associação de pontos de corte nomeados à adendos é uma prática recomendável para aumentar a legibilidade do código, visto que a declaração de um adendo não possui nome. Associar um ponto de corte nomeado à um adendo possibilita definir claramente a intenção deste ponto de corte. A consulta AQL para identificação desse cenário é uma consulta simples que busca por pontos de corte associados à adendos e que possuem o campo *isAnonymous* com valor verdadeiro. Esse

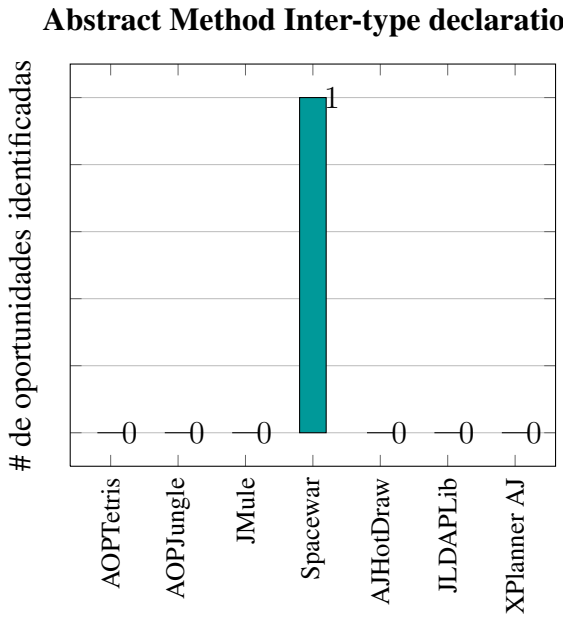


Figura 5.2 – Identificação de *Abstract Method Inter-type declaration*

campo é calculado pelo AOPJungle durante o processo de instanciação do metamodelo AspectJ. A consulta retorna o nome qualificado do aspecto, a posição do adendo dentro do aspecto e o código do ponte de corte associado. A Figura 5.3 apresenta os resultados para a identificação do cenário *Anonymous Pointcut Definition* nos projetos analisados.

```
1        find aspect a
2        where a.advice.pointcut.isAnonymous
3        returns a.fullQualifiedName, a.advice.position,
4           a.pointcut.code
```

3. *Code Duplication* : Uma das motivações do desenvolvimento orientado a aspectos é a redução da duplicidade de código. A partir de mecanismos para modularização de interesses transversais, há uma tendência em reduzir a duplicidade de código, visto que diferentes interesses, antes espalhados em vários módulos pelo sistema, agora estão encapsulados em um único módulo, o aspecto. Contudo, a duplicidade de código pode ocorrer entre adendos em um aspecto, seja por má codificação ou falha em sua documentação (PIVETA, 2009).

   A identificação de cenários de código duplicado pode ser realizada, basicamente, por meio de duas abordagens: léxica e semântica. Na abordagem léxica, a comparação entre dois blocos de código ocorre de forma textual, dentro de uma heurística determinada para reduzir falsos-positivos. Na abordagem semântica, busca-se determinar se dois blocos

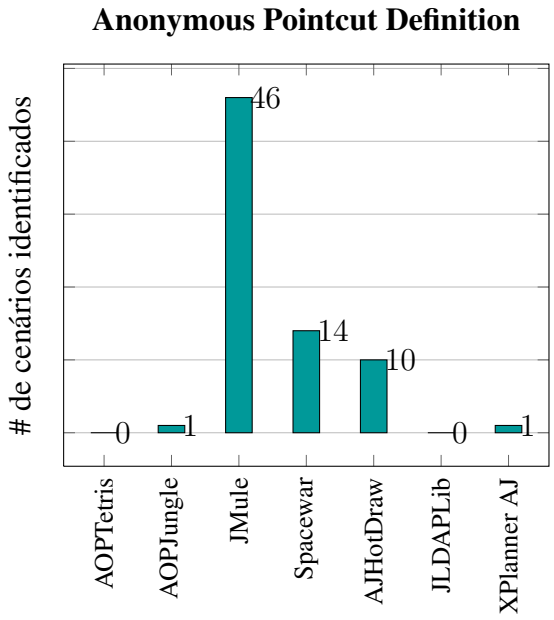


Figura 5.3 – Identificação de *Anonymous Pointcut Definition*

de código, mesmo não contendo textos idênticos resultam em uma mesma intenção. A linguagem AQL oferece apoio a ambas abordagens, porém em diferentes níveis. Na abordagem léxica, é possível comparar blocos de código diretamente em uma consulta AQL, retornando informações sobre os objetos participantes da consulta. Por exemplo, código duplicado em adendos pode ser identificado quando existem dois ou mais adendos do mesmo tipo, os quais possuem o mesmo código implementado, porém com pontos de corte diferentes associados.

A consulta AQL a seguir ilustra como identificar adendos com código duplicado.

```
1        find aspect a1 a2
2        where (a1.advice.pointcut.code = a2.advice.pointcut.code)
3           and (a1.advice.position < a2.advice.position)
4           and (a1.id = a2.id)
5           and (a1.pointcut.isAnonymous = false)
6           and (a1.advice.kind = a2.advice.kind)
7        returns a1.fullQualifiedName,
8           a1.advice.position, a2.advice.position,
9           a1.advice.pointcut.code
```

A consulta implica no uso de dois objetos do tipo *aspect* para comparar diferentes adendos dentro de um mesmo aspecto. A fim de garantir que a comparação é realizada dentro de um mesmo aspecto, o filtro de igualdade de identificadores (id) foi adicionado (linha 4). Na linha 2, o código dos pontos de corte são comparados para verificar se são iguais. Para

garantir que a comparação de um adendo não está sendo realizado com ele mesmo, foi adicionado o filtro de comparação de posição do adendo (linha 3). Somente os pontos de corte nomeados são considerados, o que é explicitado na linha 5. Por fim, os tipos de adendos são comparados para garantir que são iguais (linha 6).

A abordagem semântica para identificação de duplicidade de código é mais complexa que a abordagem léxica. Essa abordagem demanda uma análise mais profunda das estruturas do código a fim de determinar se dois blocos são semanticamente iguais. Para esses casos, a linguagem AQL oferece consultas diretas aos elementos e blocos de código do programa, a qual, então, podem ser analisados por ferramentas de comparação semântica.

Os resultados fornecidos pela ferramenta ASTOR a seguir, foram obtidos pela técnica de comparação léxica, utilizando uma taxa de similaridade de 90 %. A Figura 5.4 apresenta os resultados para a identificação do cenário *Code Duplication* nos projetos analisados.

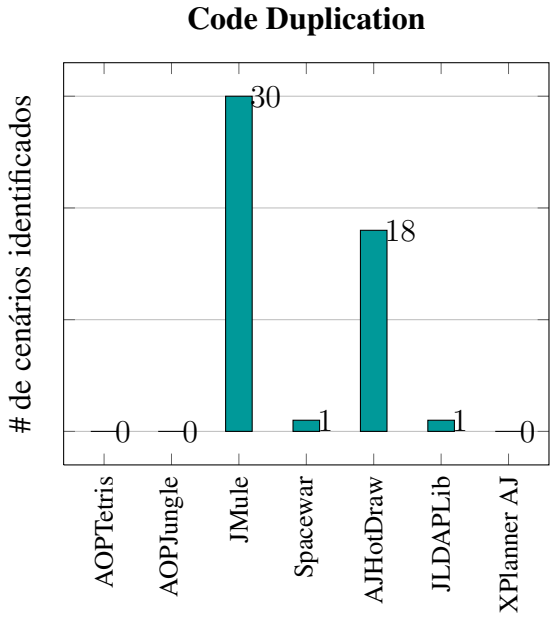


Figura 5.4 – Identificação de *Code Duplication*

4. *Divergent Changes*: Esse cenário é reconhecido quando há duas definições de pontos de corte que possuem grande similaridade entre si, variando apenas em seus modificadores ou em pequenas partes de seu predicado. Assim como a limitação *Code Duplication*, uma vez que uma parte da definição do ponto de corte é alterada, todos os pontos de corte que possuem as mesmas primitivas necessitam ser alterados, onerando a manutenção do código. A consulta AQL para identificação desse cenário considera a comparação entre

fragmentos de primitivas definidos nos pontos de corte, conforme mostrado no código a seguir. Na linha 4 ocorre a comparação de fragmentos entre dois pontos de corte. Os filtros contidos nas linhas 2 e 3 garantem que a comparação é feita em pontos de corte no mesmo aspecto, porém diferentes entre si.

```
1        find aspect a1 a2
2        where (a1.id = a2.id)
3           and (a1.pointcut.id != a2.pointcut.id)
4           and (a1.pointcut.fragment = a2.pointcut.fragment)
5        returns a1.fullQualifiedName, a1.pointcut.name,
6           a2.pointcut.name
```

A obtenção dos resultados pela ferramenta ASTOR considerou similaridade maior ou igual a 90% entre fragmentos de primitivas. A Figura 5.5 apresenta os resultados para a identificação do cenário *Divergent Changes* nos projetos analisados.

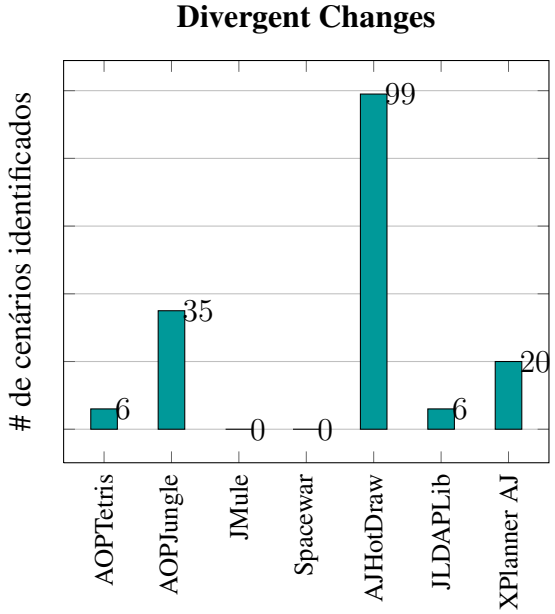


Figura 5.5 – Identificação de *Divergent Changes*

5. *Feature Envy*: Em AspectJ, os pontos de corte podem ser definidos tanto em classes quanto em aspectos. Uma vez que um ponto de corte seja definido em uma classe e seja utilizado em somente um aspecto, é interessante que esse ponto de corte seja movido para o aspecto. A identificação desse cenário não pode ser realizada de forma direta com uma consulta de AQL, contudo é possível fornecer subsídios para que a ferramenta ASTOR identifique o cenário. A seguinte consulta resulta no nome qualificado de classes, aspectos e pontos de corte (linha 4), os quais o ponto de corte definido em

uma classe seja utilizado por um adendo em um aspecto (linha 2). A Figura 5.6 apresenta os resultados para a identificação do cenário *Feature Envy* nos projetos analisados.

```
1   find aspect a, class c
2   where a.advice.pointcut.code like
3      concat ('%', ' ', c.pointcut.name, ' ', '%')
4   returns count (a.fullQualifiedName), c.fullQualifiedName,
5      a.fullQualifiedName, c.pointcut.name
6   group by c.fullQualifiedName, a.fullQualifiedName,
7      c.pointcut.name
```

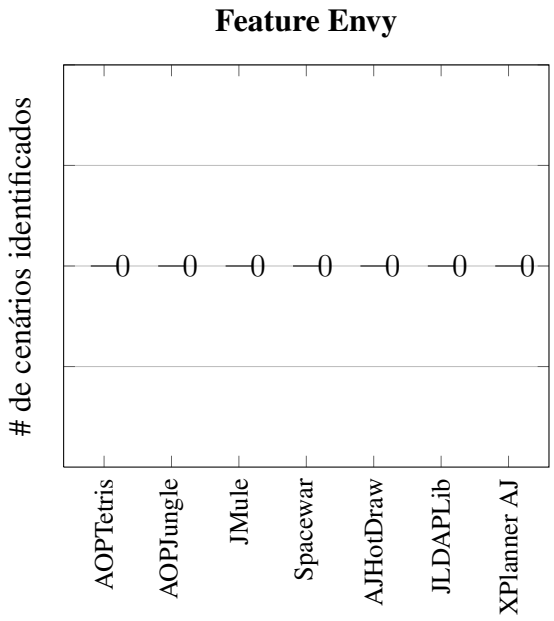


Figura 5.6 – Identificação de *Feature Envy*

6. *Large Aspect*: Esse cenário é de simples identificação, sendo possível reconhecê-lo com uma comparação entre o número de linhas (*locc*) de um aspecto e um limiar definido para que um aspecto seja considerado grande. Tanto classes quanto aspectos contendo uma grande quantidade de linhas podem onerar a manutenção do código, tornando difícil sua leitura, compreensão e reparo. A consulta AQL para identificar esse cenário é descrita a seguir.

```
1   find aspect a
2   where a.metrics.numberOfLines > [maxLinesAspect]
3   returns a.fullQualifiedName, a.metrics.numberOfLines
```

A métrica *numbeOfLines* (linha 2) é calculada pelo AOPJungle durante o processo de instanciação do metamodelo AspectJ. Foi utilizado o limiar *locc* igual a 250 para identifi-

cação desse cenário. A Figura 5.7 apresenta os resultados para a identificação do cenário *Large Aspect* nos projetos analisados.

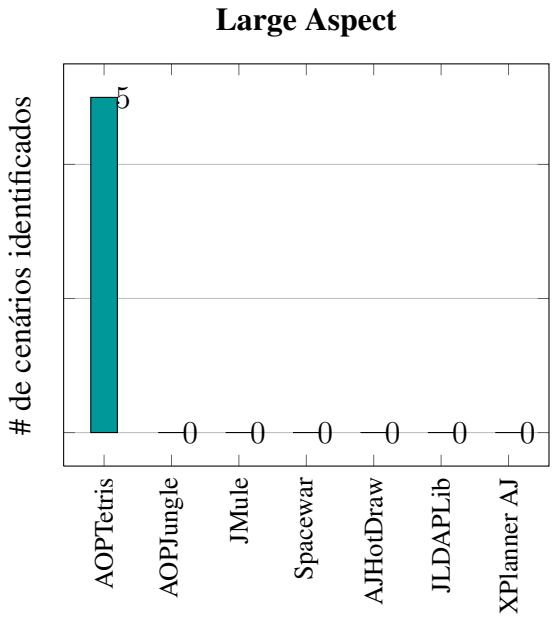


Figura 5.7 – Identificação de *Large Aspect*

7. *Large Pointcut Definition*: A definição de um ponto de corte grande pode levar a uma taxa baixa de reuso, uma vez que não é possível a recomposição de primitivas na definição de outros pontos de junção. A identificação desse cenário é realizada pela simples comparação do tamanho do código (léxico) na definição do ponto de corte em relação a um limiar definido. A consulta AQL para reconhecer esse cenário é apresentada a seguir. Para identificar o tamanho do código de definição do ponto de corte, a função *length* (linha 2) foi utilizada.

```
1       find aspect a
2       where length(a.pointcut.code) > [maxSizePointcut]
3       returns a.fullQualifiedName, a.pointcut.name, a.pointcut.code
```

Outra maneira de identificar esse cenário seria comparar a quantidade de fragmentos da definição do ponto de corte com um limiar. A consulta a seguir apresenta essa alternativa.

```
1       find aspect a
2       returns count(a.pointcut.fragment), a.fullQualifiedName, a.
           pointcut.name
3       group by a.fullQualifiedName, a.pointcut.name
4       having count(a.pointcut.fragment) > [maxSizePointcutFragment]
```

A identificação do cenário *Large Pointcut Definition* foi realizada considerando-se uma definição de ponto de corte com 200 caracteres ou mais. A Figura 5.8 apresenta os resultados para a identificação do cenário *Large Pointcut Definition* nos projetos analisados.

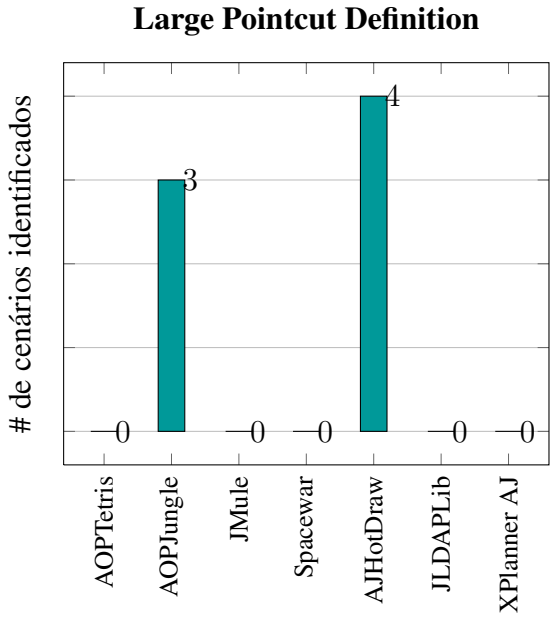


Figura 5.8 – Identificação de *Large Pointcut Definition*

8. *Lazy Aspect*: Esta limitação, conforme já ilustrado anteriormente, ocorre quando um aspecto é pequeno o suficiente a fim de não justificar sua existência. Outro caso da ocorrência do cenário *Lazy Aspect* pode ser detectada quando existem aspectos com muitos campos e métodos declarados, porém, poucos módulos são afetados por adendos deste aspecto. A consulta AQL para identificação desse cenário utiliza métricas computadas pelo AOPJungle e um parâmetro para indicar limiares para número de métodos e campos (linha 2), além do número de módulos afetados por um aspecto (linha 3).

```
1        find aspect a
2        where (a.metrics.numberOfMethods + a.metrics.numberOfFields) >
             [maxNumMembers]
3           and a.metrics.numberOfOutAffects < [minAffects]
4        returns a.fullQualifiedName
```

Para identificação desse cenário, a ferramenta ASTOR foi parametrizada para identificar aspectos que afetem até três elementos no programa. A Figura 5.9 apresenta os resultados para a identificação do cenário *Lazy Aspect* nos projetos analisados.

9. *Speculative Generality*: Esse cenário é reconhecido quando existem aspectos abstratos

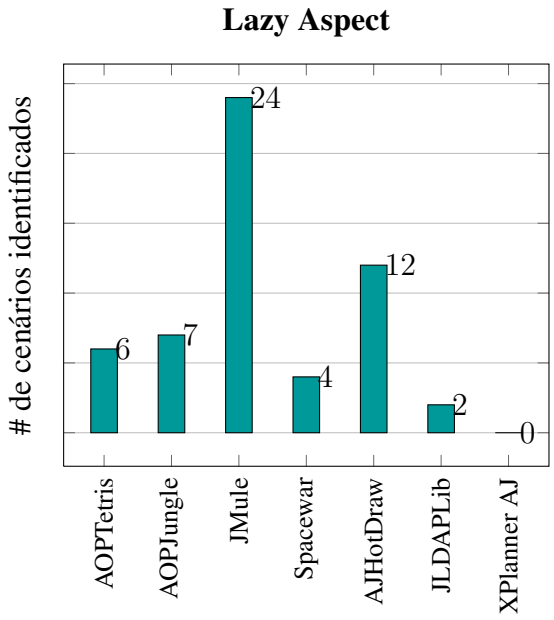


Figura 5.9 – Identificação de *Lazy Aspect*

que não afetam nenhum módulo e não possuem nenhum outro aspecto descendente. Esse cenário é característico quando aspectos e classes são criados para lidar com requisitos futuros, os quais podem nunca se materializar. A consulta AQL para identificar esse cenário utiliza as métricas de número de módulos afetados pelo aspecto (linha 2) e de número de aspectos descendentes (linha 3), além de identificar que o aspecto possui o modificador de aspecto abstrato (linha 4). A Figura 5.10 apresenta os resultados para a identificação do cenário *Speculative Generality* nos projetos analisados.

```
1    find aspect a
2    where a.metrics.numberOfOutAffects = 0
3       and a.metrics.numberOfChildren = 0
4       and a.modifier(abstract)
5    returns a.fullQualifiedName
```

10. *Double Personality*: Essa limitação é reconhecida quando há vários membros, tais como campos, métodos e declarações intertipos, sem relação em um mesmo aspecto. Múltiplos interesses em um aspecto pode resultar no problema de entrelaçamento, levando à diminuição da legibilidade e de reusabilidade do código (ELRAD; FILMAN; BADER, 2001). Não há descrito na literatura um processo automatizado para a identificação de interesses manipulados por um aspecto. A linguagem AQL, novamente, pode fornecer informações a respeito dos módulos e blocos de código para que sejam analisados visualmente ou parcialmente automatizado por alguma ferramenta de mapeamento de interesses, tais como

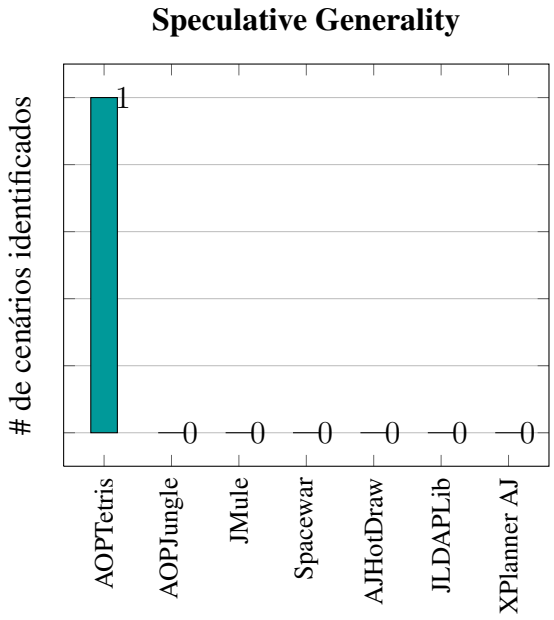


Figura 5.10 – Identificação de *Speculative Generality*

FEAT ou ConcernMapper, desde que adaptadas para programas orientados a aspectos. A ferramenta ASTOR não oferece suporte a detecção deste cenário e portanto não houve resultados alcançados.

5.4 Discussão

No estudo apresentado, a linguagem AQL apresentou-se consistente no propósito de fornecer recursos às ferramentas de análise estática em programas orientados a aspectos. Dos dez cenários propostos para serem identificados pela ferramenta ASTOR, oito puderam ser resolvidos por uma única consulta realizada em cada cenário, de acordo com as heurísticas propostas na literatura. Dentre os cenários não identificados com uma única consulta AQL, o cenário *Feature Envy* pôde ser analisado a partir de uma consulta mais genérica, para posterior processamento pela ferramenta ASTOR. Já o cenário *Double Personality* não foi automatizado pela ferramenta ASTOR, porém, da mesma forma que o cenário *Feature Envy*, uma consulta genérica pode ser gerada como subsídio para ferramentas externas de mapeamento de interesses.

O foco da linguagem AQL é fornecer informações sobre código orientado a aspectos, portanto, uma maior ênfase aos elementos que compõe esse paradigma foi dada no estudo de caso. Contudo, algumas outras avaliações em relação ao seu uso são necessárias para ampliar sua validação, principalmente em ferramentas de análise estática que demandem informações

em diferentes níveis, como por exemplo, informações intramodulares e micropadrões (GIL; MAMAN, 2005). A capacidade da linguagem AQL em fornecer dados sobre um programa está diretamente ligada a abrangência do seu metamodelo. Quanto mais enriquecido for o metamodelo, maior sua capacidade em fornecer informações a respeito de um programa. Nesse sentido, dois componentes da arquitetura de AQL são fundamentais para que a linguagem seja melhor aproveitada em um determinado contexto: o framework AOPJungle e o mecanismo de transformação de nomes qualificados. O framework AOPJungle possibilita que o metamodelo seja adaptado de acordo com as necessidades de cada ferramenta, no qual novos elementos são criados e novas computações são realizadas. Já o mecanismo de transformação pode implementar novos módulos para os recém criados elementos do metamodelo, ou mesmo criar novas funções que facilitem o uso da linguagem pelos usuários. As extensões do metamodelo e do mecanismo de transformação da AQL são discutidas no Seção 6.1 do Capítulo 6.

5.5 Considerações Finais

Este capítulo apresentou um estudo de caso envolvendo o uso da linguagem AQL como ferramenta de auxílio à identificação de oportunidades de refatoração em código orientado a aspectos. Foram escolhidos sete projetos a partir de repositórios de código aberto para análise. A ferramenta ASTOR foi implementada para realizar o processamento e apresentação das oportunidades de refatoração. Para tanto, a ferramenta utilizou os recursos da linguagem AQL para busca de cenários de acordo com um catálogo de dez oportunidades de refatoração em código orientado a aspectos, descritos na literatura. Por fim, uma discussão em relação às vantagens e às limitações apresentadas pela linguagem AQL foi conduzida. O próximo capítulo apresenta as considerações finais e referências a trabalhos futuros.

6 CONCLUSÃO

Esta dissertação apresentou uma linguagem específica de domínio para busca em código orientado a aspectos. A linguagem AQL é uma DSL textual, declarativa e externa, com sintaxe próxima a linguagens de busca em objetos, tais como JPQL e HQL. Uma implementação de referência foi conduzida, na qual a construção de um compilador que traduz código AQL para HQL foi realizada. Além disso, a reificação do modelo de um programa em AspectJ foi realizada por meio da ferramenta AOPJungle, a qual extrai informações do código e instancia o metamodelo correspondente. A fim de validar o uso da linguagem AQL foi construída uma ferramenta de análise estática (ASTOR), que usa AQL para buscar por oportunidades de refatoração em código orientado a aspectos, baseado em catálogos de oportunidades descritos na literatura.

Durante a conceitualização e o desenvolvimento da linguagem AQL, muitas decisões foram tomadas. Algumas dessas decisões estão no âmbito da engenharia aplicada na construção do compilador e do framework AOPJungle e não possuem direta interferência na forma como um usuário utilizará a linguagem. Outras decisões, porém, inevitavelmente influenciam no uso da linguagem e na sua abrangência de atuação. Essas decisões estão concentradas principalmente na expressividade da linguagem AQL e nas tecnologias envolvidas durante o processo de engenharia do compilador. Neste capítulo são abertas algumas discussões sobre as decisões que envolveram a concepção e implementação da linguagem AQL, os principais resultados obtidos durante o desenvolvimento desse trabalho, além de algumas sugestões trabalhos futuros.

O primeiro grande desafio encontrado durante o projeto foi explorar uma sintaxe que mantivesse o paradigma OO, porém fosse mais concisa que linguagens similares baseadas em OQL, sem onerar, contudo, seu poder computacional em relação às mesmas. Para tanto, alguns comandos da linguagem AQL foram mantidos com sintaxe e semântica similares às linguagens de busca largamente utilizadas, tais como SQL e HQL. Em comparação à linguagem HQL, a escrita de comandos equivalentes em AQL mostrou-se em média 38% menores para uma amostra de 115 consultas.

O segundo grande desafio encontrado durante o desenvolvimento desse trabalho envolveu a implementação do compilador. A decisão pela abordagem de traduzir a linguagem AQL para outra linguagem de consulta não foi a primeira escolha durante as fases preliminares de projeto. De fato, buscou-se realizar a consulta diretamente em objetos na memória, sem inter-

venção de nenhum outro mecanismo. Para que esta abordagem fosse viável, a pesquisa deveria ser escalável, o que requereria um mecanismo de indexação robusto para a realização de buscas consistentes. As iniciativas propostas para a execução de pesquisas de objetos em memória se mostraram ainda imaturas, e sem os recursos necessários para as principais construções de AQL. A decisão pela transformação fonte a fonte foi, então, a decisão mais viável no momento da construção do compilador, ainda que transformações subsequentes fossem necessárias, como no caso de HQL para SQL. De fato, tanto as transformações AQL para HQL quanto de HQL para SQL não representam substancial sobrecarga no processo de compilação, sendo, em geral, executadas em milésimos de segundo.

Embora existam diversas ferramentas para auxiliar no processo de construção de um compilador, escolher dentre as várias tecnologias e ambientes foi um trabalho que exigiu pesquisa, não somente no campo teórico de compiladores, como também em recursos oferecidos pelas ferramentas e técnicas disponíveis como o estado da arte em construção de linguagens. Com o auxílio do *workbench* Xtext e do ANTLR, o processo de criação do compilador pôde ser estabelecido mais facilmente do que se feito totalmente de forma manual. A partir da definição formal da linguagem, os analisadores léxicos e sintáticos puderam ser automaticamente gerados. O processo de transformação modelo a modelo utilizando a tecnologia EMF possibilitou a geração automática do código Java representando o metamodelo alvo a partir de sua definição em ferramentas para tal propósito, na plataforma Eclipse. Por fim, a linguagem Xtend forneceu facilidades tanto para o processo de transformação quanto para o processo de emissão do código alvo, devido a sua estrutura de modelos de String.

Em relação aos objetivo desse trabalho, tanto o framework AOPJungle quanto a linguagem AQL mostraram-se consistentes como um meio para obter informações de um programa orientado a aspectos. No estudo de caso apresentado, a linguagem AQL permitiu identificar oportunidades de refatoração, de acordo com as heurísticas propostas, com apenas uma única consulta. Nas exceções encontradas, a linguagem mostrou-se apta a fornecer informações sobre o código que puderam, então, ser processadas pela ferramenta ASTOR.

O framework AOPJungle pode ser enriquecido com mais informações sobre um programa orientado a aspectos. A proposta do projeto da linguagem AQL permitiu responder à evolução do modelo sem modificações em sua sintaxe. Outras construções podem agregar formas diferentes de utilização e mitigar limitações, que possam diminuir sua adoção pela comunidade de programadores de linguagens orientadas a aspectos. Essas melhorias são sugeridas

na seção seguinte, como forma de contribuir para a evolução do projeto.

6.1 Trabalhos Futuros

- **Validação em projetos de larga escala**. A validação da linguagem AQL em outros projetos de ferramenta de análise estática é um estudo interessante para a ampliação do metamodelo e dos serviços fornecidos pelo framework AOPJungle. Além disso, a capacidade da linguagem AQL de realizar pesquisas em múltiplos projetos possibilita extensões na solução para a inclusão de análise de repositórios de programas remotos, como por exemplo repositórios de código aberto.

- **Validação de uso em campo**. A condução de estudos empíricos voltados a aceitação e uso em maior escala da linguagem AQL é um campo de pesquisa que possibilitará compreender possíveis melhorias e o direcionamento da evolução da linguagem. As ferramentas de apoio à linguagem devem evoluir para a condução desses estudos junto aos usuários. Isso inclui aprimoramento dos editores de escrita e resultado de consultas e a criação de visões e de perspectivas no Eclipse IDE.

- **Estender a linguagem para a criação de novas funções**. Diferentes funções podem ser criadas para facilitar a utilização da linguagem, à medida que o metamodelo orientado a aspectos seja estendido. A evolução do metamodelo pode conter informações interessantes sobre um programa, como por exemplo, os elementos que perfazem um determinado padrão de projeto, o uso de determinada convenção ou ainda um modelo evoluído de métricas.

- **Alternativas de representação do metamodelo orientado a aspectos**. Alternativas ao modelo de implementação do framework AOPJungle podem ser testadas, como por exemplo, o uso de EMF na representação do metamodelo de um programa orientado a aspectos. Isso permitiria explorar outras formas de consulta diretas ao modelo reificado, assim que tecnologias maduras para tal propósito sejam implementadas.

- **Alternativas de transformação de modelos**. Outros modelos de transformação de modelos utilizando-se a tecnologia EMF podem ser aplicados, incluindo linguagens de transformação, como por exemplo a linguagem ATL (JOUAULT et al., 2006). Além de outros mecanismos de transformação, o compilador de AQL pode gerar modelos e emissores

para diferentes linguagens alvo, a fim de avaliar outras possibilidades de uso da linguagem.

REFERÊNCIAS

AHO, A. et al. **Compilers**: principles, techniques, and tools (2nd edition) principles, techniques, and tools (2nd edition). 2.ed. [S.l.]: Prentice Hall, 2006.

AKKI et al. Patterns for Evaluating Usability of Domain-Specific Languages. In: PATTERN LANGUAGES OF PROGRAMS CONFERENCE 2012. **Anais...** ACM, 2012.

AQL. **Página do projeto AQL**. Disponível em http://www.defaveri.com.br/ufsm/aql>. Acesso em: março 2013.

BACKUS, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In: IFIP CONGRESS. **Anais...** [S.l.: s.n.], 1959. p.125–131.

BARRENECHEA, E. S. **A Query-based Approach for the Analysis of Aspect-oriented Systems**. 2007. Dissertação (Mestrado em Ciência da Computação) — University of Waterloo.

BAUER, C.; KING, G. **Hibernate in action**. [S.l.]: Manning Greenwich, 2005.

BERGIN, T. J.; GIBSON, R. G. **History of Programming languages II**. [S.l.]: ACM Press New York, 1996.

BRYANT, B. R. et al. Grammar inference technology applications in software engineering: domain-specific development. In: GRAMMATICAL INFERENCE: THEORETICAL RESULTS AND APPLICATIONS, 10., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.276–279. (ICGI'10).

BURN, O. **CheckStyle Tool**. Disponível em: <http://checkstyle.sourceforge.net/>. Acesso em: outubro de 2012.

CATTELL, R. G. G.; BARRY, D. K. **The Object Data Standard**: odmg 3.0. [S.l.]: Morgan Kaufmann, 2000.

CHAMBERLIN, D. D. et al. **SEQUEL 2**: a unified approach to data definition, manipulation, and control. Riverton, NJ, USA: IBM Corp., 1976. 560–575p. v.20, n.6.

CLEMENT, A. Aspect-Oriented Programming with AJDT. In: IN ECOOP WORKSHOP ON ANALYSIS OF ASPECT-ORIENTED SOFTWARE. **Anais...** Springer, 2003.

CODD, E. F. Relational completeness of data base sublanguages. In: DATABASE SYSTEMS. **Anais...** Prentice-Hall, 1972. p.65–98.

COELHO, W.; MURPHY, G. C. **ActiveAspect**: presenting crosscutting structure. 2005. 1-4p. v.30, n.4.

COHEN, T.; GIL, J. Y.; MAMAN, I. JTL: the java tools language. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 21., New York, NY, USA. **Proceedings...** ACM, 2006. p.89–108. (OOPSLA '06).

COPELAND, T. **PMD applied**. [S.l.]: Centennial Books San Francisco, 2005.

CORPORATION, M. **Language-Integrated Query (LINQ)**. 2008.

CUNNINGHAM, H. A little language for surveys: constructing an internal dsl in ruby. In: ACM-SE 46: PROCEEDINGS OF THE 46TH ANNUAL SOUTHEAST REGIONAL CONFERENCE ON XX, New York, NY, USA. **Anais...** ACM, 2008.

DEURSEN, A. van; KLINT, P. **Little languages**: little maintenance ? Amsterdam, The Netherlands, The Netherlands: [s.n.], 1997.

DEURSEN, A. van; KLINT, P.; VISSER, J. **Domain-specific languages**: an annotated bibliography. New York, NY, USA: ACM, 2000. 26–36p. v.35, n.6.

EBERT, J. et al. **Using Difference Information to Reuse Software Cases**. 2007. v.27, n.2.

ELLIOTT, C. An embedded modeling language approach to interactive 3D and multimedia animation. **Software Engineering, IEEE Transactions on**, [S.l.], v.25, n.3, p.291–308, 1999.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: introduction. **Communications of the ACM**, [S.l.], v.44, n.10, p.29–32, 2001.

EMDEN, E. V.; MOONEN, L. Java Quality Assurance by Detecting Code Smells. **WCRE'02**, Los Alamitos, CA, USA, v.0, p.97, 2002.

EYSHOLDT, M.; BEHRENS, H. Xtext: implement your language faster than the quick and dirty way. In: ACM INTERNATIONAL CONFERENCE COMPANION ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS COMPANION, New York, NY, USA. **Proceedings...** ACM, 2010. p.307–309. (SPLASH '10).

FAITH, R.; NYLAND, L. S.; PRINS, J. F. Khepera: a system for rapid implementation of domain specific languages. In: IN PROCEEDINGS USENIX CONFERENCE ON DOMAIN-SPECI LANGUAGES. **Anais...** [S.l.: s.n.], 1997. p.243–255.

FEIJS, L.; KRIKHAAR, R.; VAN OMMERING, R. **A relational approach to support software architecture analysis**. New York, NY, USA: John Wiley & Sons, Inc., 1998. 371–400p. v.28, n.4.

FILMAN, R. E.; FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS, OOPSLA. **Anais...** [S.l.: s.n.], 2000. v.2000.

FISHER, K.; GRUBER, R. PADS: a domain-specific language for processing ad hoc data. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2005. v.40, n.6, p.295–304.

FOKAEFS, M.; TSANTALIS, N.; CHATZIGEORGIOU, A. Jdeodorant: identification and removal of feature envy bad smells. In: SOFTWARE MAINTENANCE, 2007. ICSM 2007. IEEE INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2007. p.519–520.

FOWLER, M. **Refactoring**: improving the design of existing code. Boston, MA, USA: Addison-Wesley, 1999.

FOWLER, M. **Domain Specific Languages**. [S.l.]: Prentice Hall, 2010. (The Addison-Wesley Signature Series).

FRIEDMAN, D. P.; WAND, M. Reification: reflection without metaphysics. In: ACM SYMPOSIUM ON LISP AND FUNCTIONAL PROGRAMMING, 1984., New York, NY, USA. **Proceedings...** ACM, 1984. p.348–355. (LFP '84).

GABRIEL, P.; GOULÃO, M.; AMARAL, V. Do Software Languages Engineers Evaluate their Languages? **arXiv preprint arXiv:1109.6794**, [S.l.], 2011.

GHOSH, D. **DSLs in Action**. Pap/Psc.ed. [S.l.]: Manning Publications, 2010.

GIL, J. Y.; MAMAN, I. Micro patterns in Java code. **ACM SIGPLAN Notices**, [S.l.], v.40, n.10, p.97–116, 2005.

GRONBACK, R. C. **Eclipse Modeling Project**: a domain-specific language (dsl) toolkit. [S.l.]: Addison-Wesley Professional, 2009.

HAJIYEV, E.; VERBAERE, M.; MOOR, O. de. CodeQuest: scalable source code queries with datalog. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 20., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2006. p.2–27. (ECOOP'06).

HANNEMANN, J.; KICZALES, G. Overcoming the Prevalent Decomposition of Legacy Code. In: WORKSHOP ON ADVACED SEPARATION OF CONCERNS. **Anais...** [S.l.: s.n.], 2001.

HOVEMEYER, D.; PUGH, W. Finding bugs is easy. **ACM Sigplan Notices**, [S.l.], v.39, n.12, p.92–106, 2004.

HUANG, J. et al. EXTRACTOR: an extensible framework for identifying aspect-oriented refactoring opportunities. **International Conference on System Science, Engineering Design and Manufacturing Informatization**, [S.l.], v.2, p.222–226, 2011.

HUDAK, P. **Building Domain-Specific Embedded Languages**. 1996. v.28.

HUDAK, P. **Domain-Specific Languages - Handbook of Programming Languages Vol. II**: little languages and tools. 1st.ed. [S.l.]: Macmillan Technical Publishing, 1998.

HUNT, A.; THOMAS, D. **Pragmatic Unit Testing in Java with JUnit**. [S.l.]: The Pragmatic Programmers, 2003.

JANZEN, D.; DE VOLDER, K. Navigating and querying code without getting lost. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2., New York, NY, USA. **Proceedings...** ACM, 2003. p.178–187. (AOSD '03).

JENNINGS, J.; BEUSCHER, E. Verischemelog: verilog embedded in scheme. **ACM SIGPLAN Notices**, [S.l.], v.35, n.1, p.123–134, 2000.

JETBRAINS. **JetBrains Meta Programming System**. 2012.

JOHNSON, S. C. **Yacc**: yet another compiler compiler. New York, NY, USA: Holt, Rinehart, and Winston, 1979. v.2.

JOUAULT, F. et al. ATL: a qvt-like transformation language. In: COMPANION TO THE 21ST ACM SIGPLAN SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, New York, NY, USA. **Anais...** ACM, 2006. p.719–720. (OOPSLA '06).

KABANOV, J.; RAUDJÄRV, R. Embedded typesafe domain specific languages for java. In: IN PPPJ '08: PROCEEDINGS OF THE 6TH INTERNATIONAL SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA. **Anais...** ACM, 2008. p.189–197.

KAMIN, S.; HYATT, D. A special-purpose language for picture-drawing. In: CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES. **Proceedings...** [S.l.: s.n.], 1997. p.297–310.

KANG, K. C. et al. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. [S.l.]: Carnegie-Mellon University Software Engineering Institute, 1990.

KEITH, M.; SCHINCARIOL, M. **Pro EJB 3**: java persistence api. [S.l.]: Apress, 2006.

KERIEVSKY, J. **Refactoring to Patterns**. [S.l.]: Pearson Higher Education, 2004.

KERSTEN, M.; MURPHY, G. C. Mylar: a degree-of-interest model for ides. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 4., New York, NY, USA. **Proceedings...** ACM, 2005. p.159–168. (AOSD '05).

KICZALES, G. et al. Aspect-Oriented Programming. In: AKSIT, M.; MATSUOKA, S. (Ed.). **Proceedings European Conference on Object-Oriented Programming**. Berlin, Heidelberg, and New York: Springer-Verlag, 1997. v.1241, p.220–242.

KICZALES, G. et al. An Overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 15., London, UK, UK. **Proceedings...** Springer-Verlag, 2001. p.327–353. (ECOOP '01).

KIEBURTZ, R. B. et al. A software engineering experiment in software component generation. In: SOFTWARE ENGINEERING, 18., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1996. p.542–552. (ICSE '96).

KLARLUND, N.; SCHWARTZBACH, M. I. A domain-specific language for regular sets of strings and trees. **Software Engineering, IEEE Transactions on**, [S.l.], v.25, n.3, p.378–386, 1999.

KLASSEN, L.; WAGNER, R. EMorF-A tool for model transformations. **Electronic Communications of the EASST**, [S.l.], v.54, 2012.

KULLBACH, B. et al. Querying as an Enabling Technology in Software Reengineering. In: IN PROCEEDINGS OF THE 3RD EUROMICRO CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Anais...** IEEE Computer Society, 1999. p.42–50.

LADD, D. A.; RAMMING, C. J. Two Application languages in software production. In: VHLLS'94: PROCEEDINGS OF THE USENIX 1994 VERY HIGH LEVEL LANGUAGES SYMPOSIUM PROCEEDINGS ON USENIX 1994 VERY HIGH LEVEL LANGUAGES SYMPOSIUM PROCEEDINGS, Berkeley, CA, USA. **Anais...** USENIX Association, 1994. p.10.

LADD, D. A.; RAMMING, J. C. Two application languages in software production. In: USENIX VERY HIGH LEVEL LANGUAGES SYMPOSIUM PROCEEDINGS. **Anais...** [S.l.: s.n.], 1994. p.169–178.

LAMPORT, L. **LaTeX: a document preparation system (2nd edition)**. 2.ed. [S.l.]: Addison-Wesley Professional, 1994.

LESK, M. E. et al. **Lex - A lexical analyzer generator**. Philadelphia, PA, USA: W. B. Saunders Company, 1990. 375–387p.

MACIA BERTRAN, I.; GARCIA, A.; STAA, A. von. An exploratory study of code smells in evolving aspect-oriented systems. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, New York, NY, USA. **Proceedings...** ACM, 2011. p.203–214. (AOSD '11).

MARTIN, J. **Fourth-generation languages. Volume 1. Principles**. [S.l.]: Prentice Hall Inc., Old Tappan, NJ, 1985.

MCCORMICK, E.; DE VOLDER, K. JQuery: finding your way through tangled code. In: COMPANION TO THE 19TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, New York, NY, USA. **Anais...** ACM, 2004. p.9–10. (OOPSLA '04).

MENS, T.; TOURWÉ, T. **A Survey of Software Refactoring**. Piscataway, NJ, USA: IEEE Press, 2004. 126–139p. v.30, n.2.

MERNIK, M. et al. **When And How To Develop Domain-Specific Languages**. 2003.

MOHA, N. et al. DECOR: a method for the specification and detection of code and design smells. **Software Engineering, IEEE Transactions on**, [S.l.], v.36, n.1, p.20–36, 2010.

MONTEIRO, M. P.; FERNANDES, J. a. M. Towards a catalog of aspect-oriented refactorings. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 4., New York, NY, USA. **Proceedings...** ACM, 2005. p.111–122. (AOSD '05).

NARDI, B. A. **A small matter of programming**: perspectives on end user computing. [S.l.]: The MIT Press, 1993.

NIPKOW, T.; VON OHEIMB, D. Java light is type-safe—definitely. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 25. **Proceedings...** [S.l.: s.n.], 1998. p.161–170.

OLBRICH, S. et al. The evolution and impact of code smells: a case study of two open source systems. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, 2009., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.390–400. (ESEM '09).

OPDYKE, W. F. **Refactoring object-oriented frameworks**. 1992. Tese (Doutorado em Ciência da Computação) — , Champaign, IL, USA. UMI Order No. GAX93-05645.

PARR, T. **The definitive ANTLR reference** : building domain-specific languages. [S.l.]: Pragmatic Bookshelf, 2007.

PAUL, R. **Designing and Implementing a Domain-Specific Language**. 2005. v.135.

PFEIFFER, J.-H.; SARDOS, A.; GURD, J. R. Complex code querying and navigation for AspectJ. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 2005., New York, NY, USA. **Proceedings...** ACM, 2005. p.60–64. (eclipse '05).

PIVETA, E. K. et al. Detecting bad smells in aspectj. **JOURNAL OF UNIVERSAL COMPUTER SCIENCE**, [S.l.], 2006.

PIVETA, K. E. **Improving the Search for Refactoring Opportunities on Object-Oriented and Aspect-Oriented Software**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

PRESSMAN, R. **Software Engineering**: a practitioner's approach. 6.ed. New York, NY, USA: McGraw-Hill, Inc., 2005.

RAHIEN, A. **Building Domain-Specific Languages in Boo**. [S.l.]: Manning, 2008.

RÉVEILLÈRE, L. et al. A DSL approach to improve productivity and safety in device drivers development. In: International Conference on Automated Software Engineering, France. **Anais...** [S.l.: s.n.], 2000. p.101–109.

ROBILLARD, M. P.; MURPHY, G. C. Representing concerns in source code. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, [S.l.], v.16, n.1, p.3, 2007.

ROBILLARD, M. P.; WEIGAND-WARR, F. ConcernMapper: simple view-based separation of scattered concerns. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.65–69.

ROMAN, S. **Writing Excel macros with VBA**. [S.l.]: O'Reilly Media, 2008.

ROZENBERG, G.; EHRIG, H. **Handbook of graph grammars and computing by graph transformation**. [S.l.]: World Scientific London, 1999. v.1.

SAMMET, J. E. Programming languages: history and future. **Commun. ACM**, New York, NY, USA, v.15, n.7, p.601–610, July 1972.

SAVITCH, W. J. **Pascal, an introduction to the art and science of programming**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986.

SCHUMACHER, J. et al. Building empirical support for automated code smell detection. **ESEM' 10**, [S.l.], p.1, 2010.

SEBESTA, R. **Concepts of programming languages**. [S.l.]: Addison-Wesley, 2009.

SIMOS, M. A. **Organization domain modeling (ODM)**: formalizing the core domain modeling life cycle. New York, NY, USA: ACM, 1995. 196–205p. v.20, n.SI.

SOARES, S. C. B. **An Aspect-Oriented Implementation Method**. 2004. Tese (Doutorado em Ciência da Computação) — Universidade Federal de Pernambuco.

SOMMERVILLE, I. **Software Engineering**. 9.ed. Harlow, England: Addison-Wesley, 2010.

SPINELLIS, D. **Notable design patterns for domain-specific languages**. 2001. 91-99p. v.56, n.1.

STERLING, L.; SHAPIRO, E. **The Art of Prolog**. Cambridge (MA): MIT Press, 1986.

STICHNOTH, J. M.; GROSS, T. Code Composition as an Implementation Language for Compilers. In: IN USENIX CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES. **Anais...** [S.l.: s.n.], 1997. p.119–132.

SYSTEM, G. O. **Bison - GNU Parser Generator**. 2012.

TARSKI, A. **On the calculus of relations**. 1941. 73–89p. v.6, n.3.

TAYLOR, R. N.; TRACZ, W.; COGLIANESE, L. **Software development using domain-specific software architectures**. New York, NY, USA: ACM, 1995. 27–38p. v.20, n.5.

VANBRABANT, R. **Google Guice**: agile lightweight dependency injection framework. [S.l.]: Apress, 2008.

VERBAERE, M.; HAJIYEV, E.; DE MOOR, O. Improve software quality with SemmleCode: an eclipse plugin for semantic code search. In: COMPANION TO THE 22ND ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS AND APPLICATIONS COMPANION, New York, NY, USA. **Anais...** ACM, 2007. p.880–881. (OOPSLA '07).

VLISSIDES, J. et al. **Design patterns**: elements of reusable object-oriented software. [S.l.: s.n.], 1995. v.49.

VOLDER, K. D. **Type-Oriented Logic Meta Programming**. 1998. Tese (Doutorado em Ciência da Computação) — Department of Computer Science, Vrije Universiteit Brussel, Belgium.

W3C. **Hypertext Markup Language (HTML)**. 2012.

WA, R.; SIMONYI, C.; SIMONYI, C. **The Death Of Computer Languages, The Birth Of Intentional Programming**. [S.l.]: University of Newcastle upon Tyne, Department of Computing Science, 1995.

WALMSLEY, P. **XQuery**. [S.l.]: O'Reilly Media, 2009.

WEISS, D. M.; LAI, C. T. R. **Software product-line engineering**: a family-based software development process. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

WEXELBLAT, R. L. (Ed.). **History of programming languages I**. New York, NY, USA: ACM, 1981.

WILE, D. S. Supporting the DSL spectrum. **Journal of Computing and Information Technology**, [S.l.], v.9, n.4, p.263–287, 2001.

ZLOOF, M. M. Query-by-example: the invocation and definition of tables and forms. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 1., New York, NY, USA. **Proceedings...** ACM, 1975. p.1–24. (VLDB '75).

APÊNDICES

APÊNDICE A – Sintaxe de AQL em EBNF

```
Query hidden (WS, SL_COMMENT, ML_COMMENT) : query=
   QueryStatement;

QueryStatement : (find=FindClause) (where=WhereClause)? (
   return=ReturnsClause) (orderby=OrderByClause)? (groupby=
   GroupByClause)? ;

FindClause : clause='find' bindingObject+=BindingObject (','
   bindingObject+=BindingObject)* ;

BindingObject : type=ObjectType (alias+=ID)+ ;

ObjectType : {Project} value='project' | {Package} value='
   package' | {Class} value='class' | {Aspect} value='aspect'
    | {Interface} value='interface' | {Enum} value='enum' ;

WhereClause : clause='where' expression=Expression ;

ReturnsClause : clause='returns' expressions+=Expression ('as
   ' resultAlias+=ID)? ( ',' expressions+=Expression ('as'
   resultAlias+=ID)?)* ;

GroupByClause : clause='group by' expressions+=
   SimpleQualifiedName (',' expressions+=SimpleQualifiedName)
   * ;

OrderByClause : clause='order by' expressions+=
   SimpleQualifiedName (',' expressions+=SimpleQualifiedName)
   * option=('asc' | 'desc')? ('having' havingExpression=
   Expression)? ;

SimpleQualifiedName returns QualifiedName : qualifiers+=
   SimpleQualified ('.' qualifiers+=SimpleQualified)* ;

SimpleQualified returns QualifiedElement : Identifier ;

Expression hidden (WS, SL_COMMENT, ML_COMMENT) : Or ;

Or returns Expression : And ( ('or'{Or.left=current}) right=
   And)* ;

And returns Expression : Relation ( ('and'{And.left=current}
   right=Relation))* ;

Relation returns Expression : Add ( ( ('='{Equals.left=
```

```
   current}) | ('>'{Greater.left=current}) | ('>='{
   GreaterEqual.left=current}) | ('<'{Less.left=current}) |
   ('<='{LessEqual.left=current}) | ('like'{Like.left=current
   }) | ('!='{NotEquals.left=current} )) right=Add ) * ;

Add returns Expression : Mult ( ( ('+'{Add.left=current} ) |
   ('-'{Minus.left=current} )) right=Mult) * ;

Mult returns Expression : In ( ( ('*'{Mult.left=current}) |
   ('/'{Div.left=current}) | ('%'{Mod.left=current})) right=
   In) * ;

In returns Expression : Unary (('in' {In.left=current}) right
   =Unary) * ;

Unary returns Expression : Exponential | ('not' {Not} exp=
   Unary) | ('-' {UnaryMinus} expr=Unary) ;

Exponential returns Expression : Atom ('^' {Exponential.left=
   current} right=Exponential)? ;

Atom returns Expression : Literal | QualifiedName |
   ParsExpression ;

ParsExpression returns Expression : {ParsExpression} '(' expr
   +=Expression (',' expr+=Expression)* ')' ;

Literal : {StringLiteral} value=STRING | {FloatLiteral} value
   =FLOAT | {IntegerLiteral} value=INT | {Null} value='null'
   | {True} value='true' | {False} value='false' ;

QualifiedName returns QualifiedName : granular=('distinct'|'
   all')? qualifiers+=Qualified ('.' qualifiers+=Qualified)*
   ;

Qualified returns QualifiedElement : QualifiedElement ;

QualifiedElement returns QualifiedElement : Identifier |
   Function ;

Function returns Function : name=ID '(' params+=Expression
   (',' params+=Expression)* ')' ;

Identifier returns Identifier : id=ID ;

terminal ID returns ecore::EString : ('a'..'z'|'A'..'Z'|'_')
   ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
```

```
terminal INT returns ecore::EInt : '0'..'9'+ ;

terminal FLOAT returns ecore::EDouble : ( ('0'..'9')+ ( '.'
   ('0'..'9')+ ( ('E'|'e') ('-')? ('0'..'9')+ )? )? ) ;

terminal STRING : '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u
   '|'"'|"'"|'\\') | !('\\'|'"') )* '"' | "'" ( '\\' ('b'|'t
   '|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'" ;

terminal ML_COMMENT : '/*' -> '*/' ;

terminal SL_COMMENT : '//' !('\n'|'\r')* ('\r'? '\n')? ;

terminal WS : (' '|'\t'|'\r'|'\n')+ ;
```

APÊNDICE B – Palavras Reservadas de AQL

```
all
and
aspect
class
distinct
enum
false
find
group by
having
in
interface
like
not
null
or
order by
package
project
returns
true
where
```

APÊNDICE C – Metamodelo de AspectJ

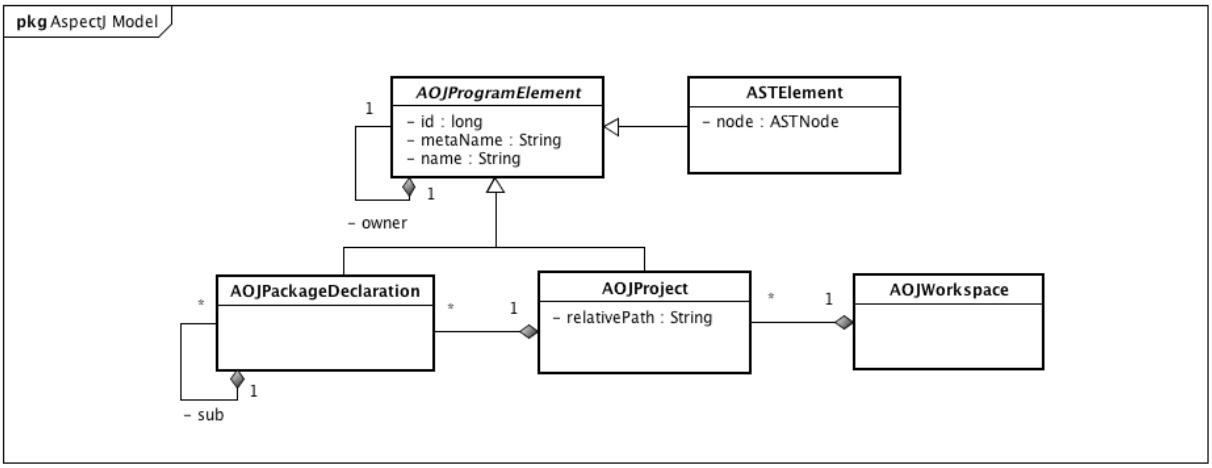


Figura C.1 – Modelo de projetos

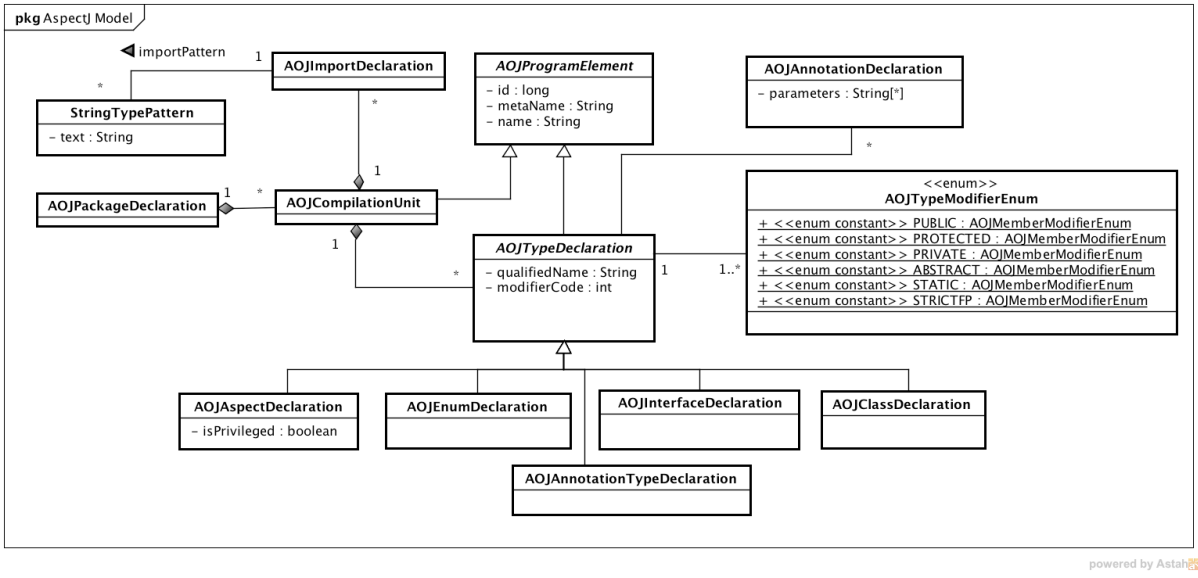


Figura C.2 – Modelo de tipos em AspectJ

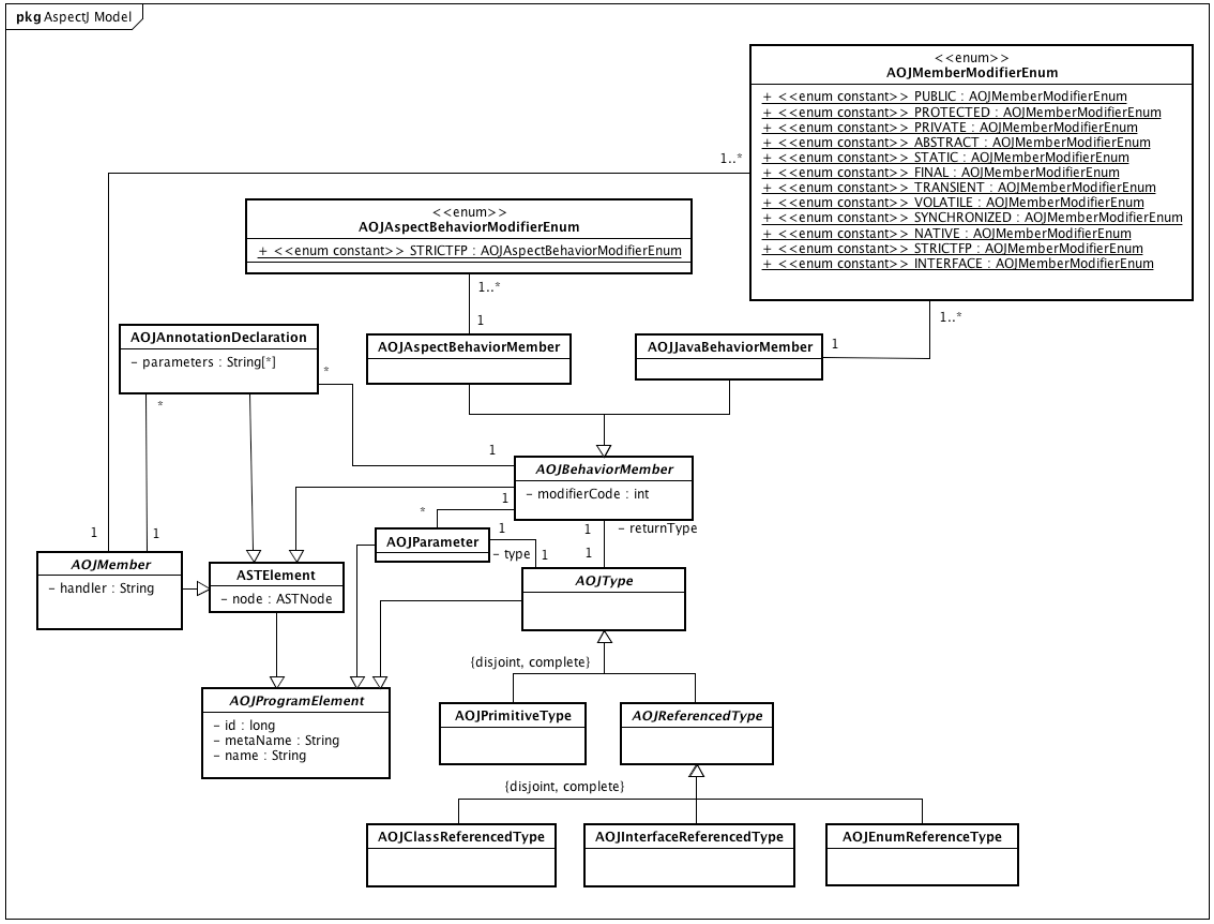


Figura C.3 – Modelo de membros dos tipos de AspectJ

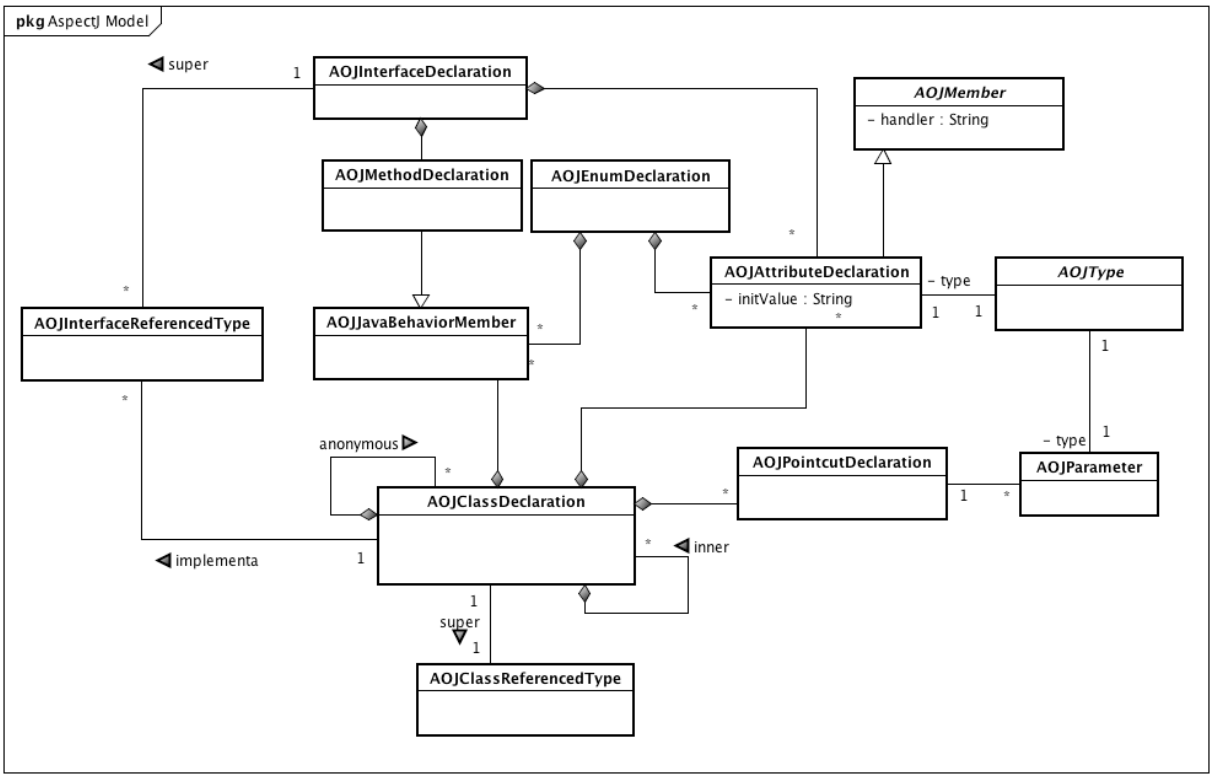


Figura C.4 – Modelo de classe

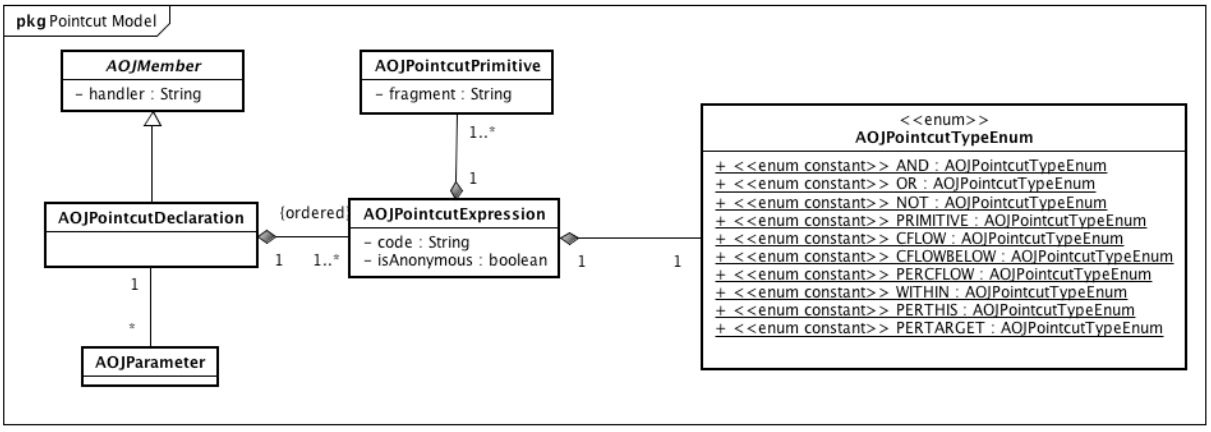


Figura C.5 – Modelo de ponto de corte

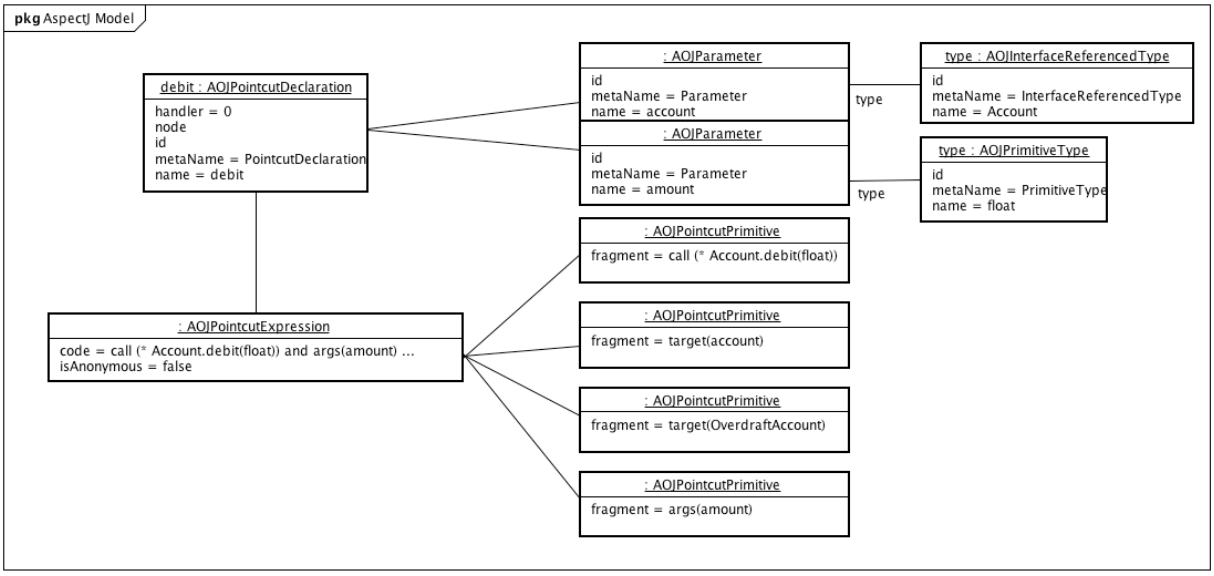


Figura C.6 – Modelo de objetos da declaração de um ponto de corte

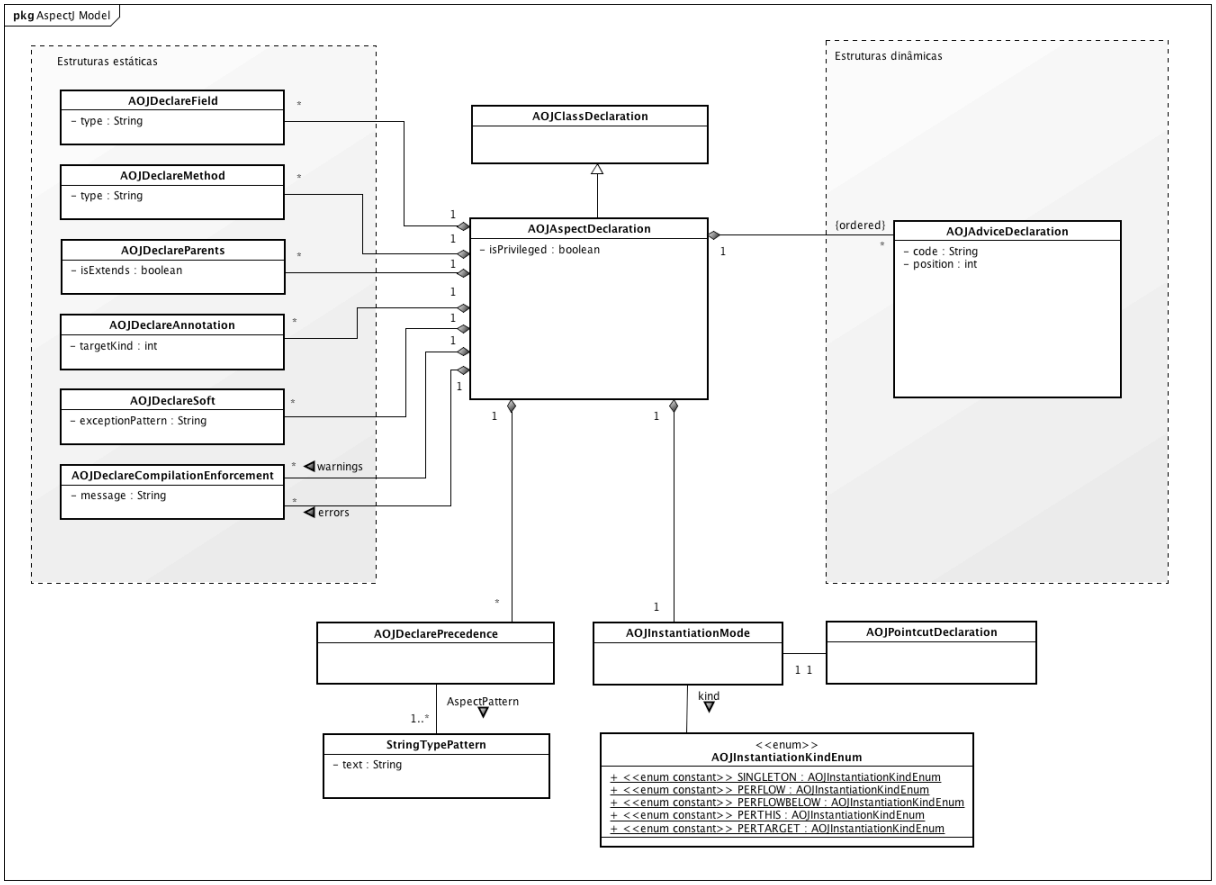


Figura C.7 – Modelo de aspecto

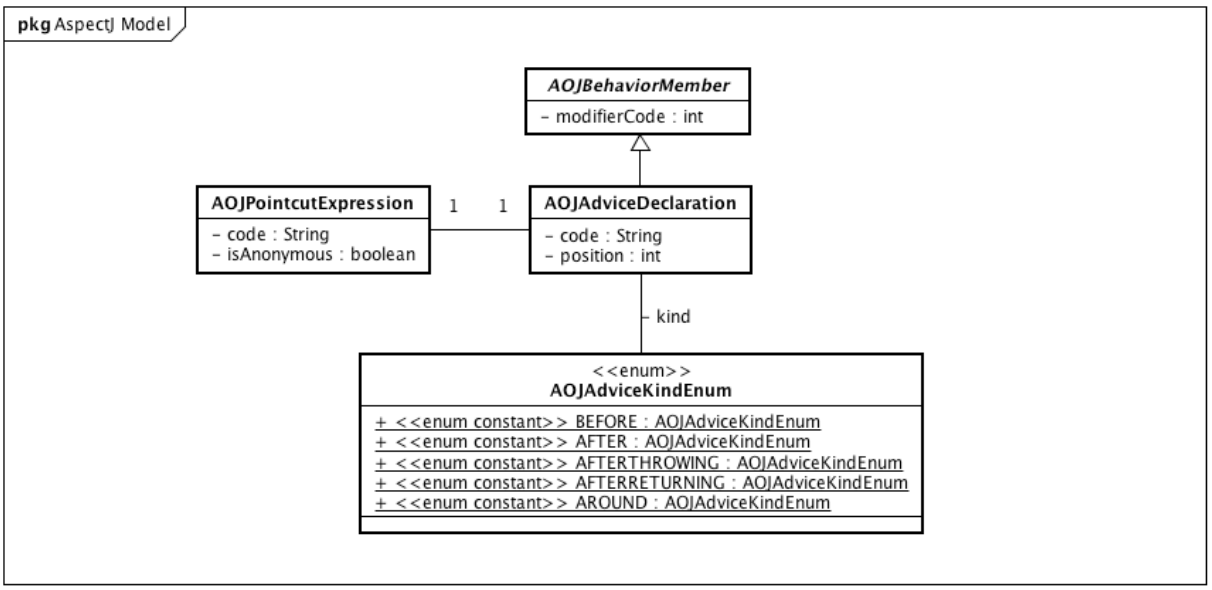


Figura C.8 – Modelo de adendo

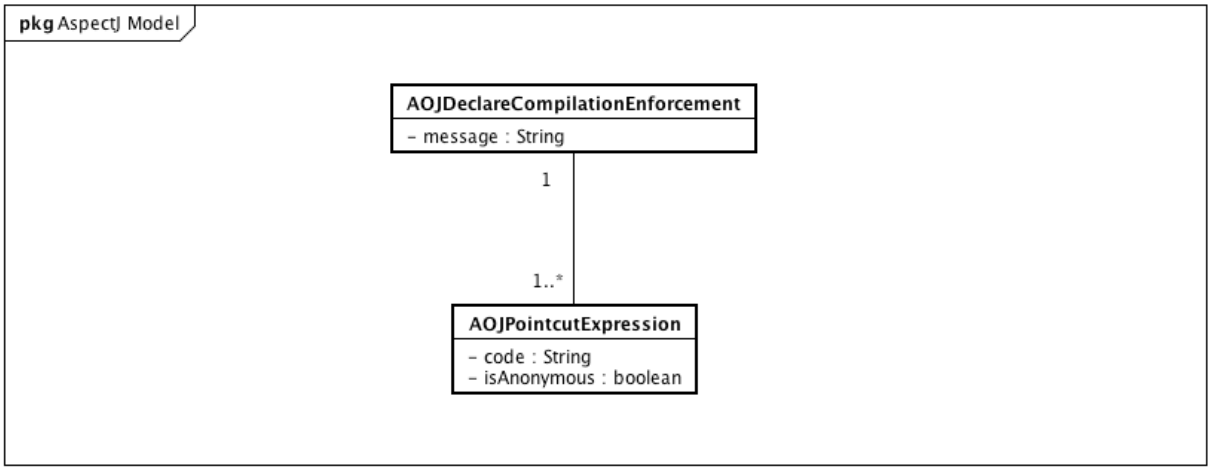
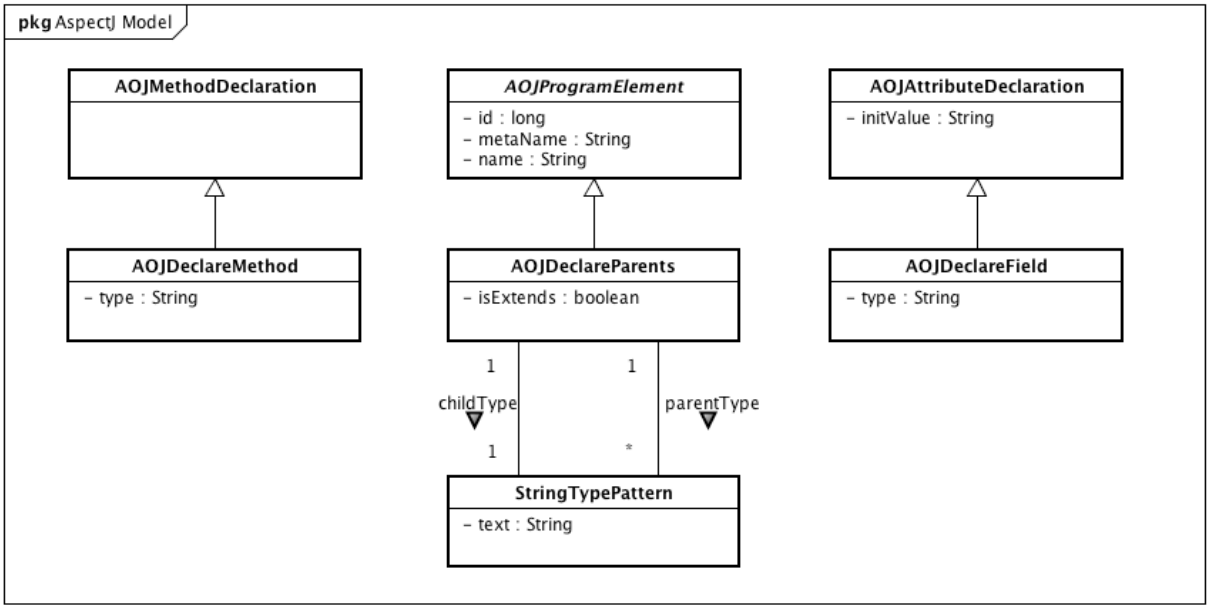


Figura C.9 – Modelo de regras de combinação



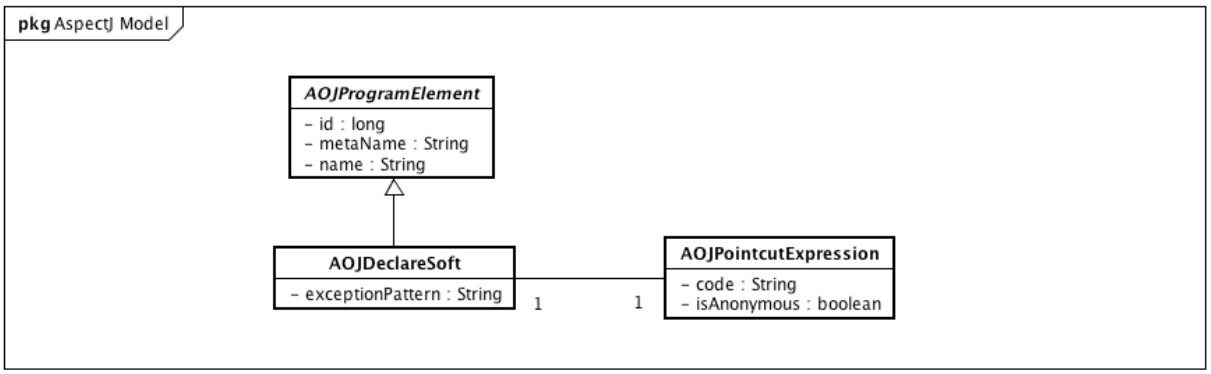


Figura C.10 – Modelo de suavização de exceções

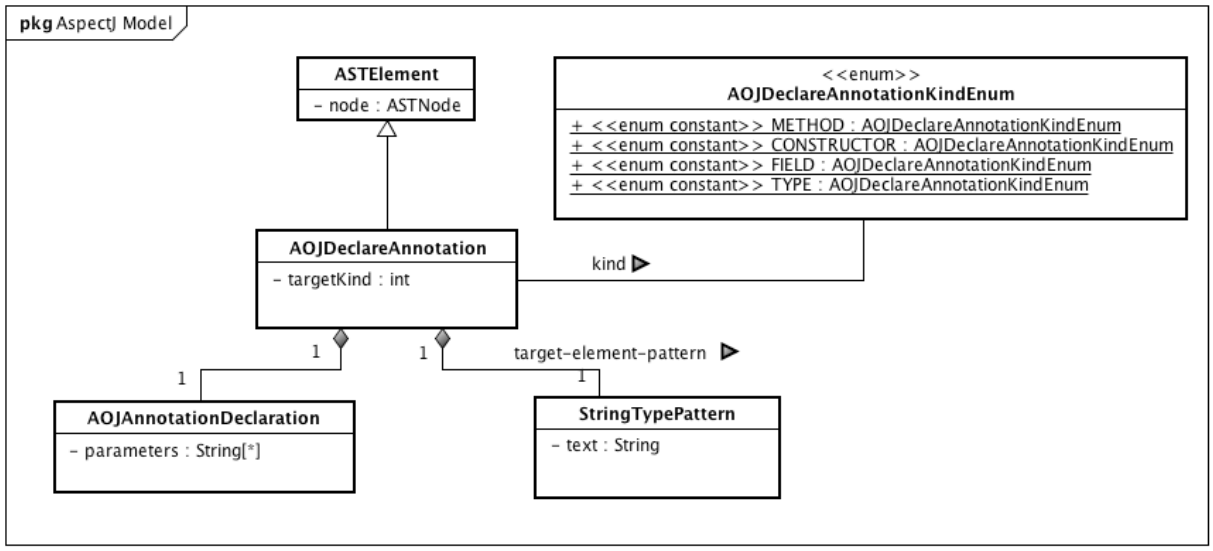


Figura C.11 – Modelo de introdução de anotação

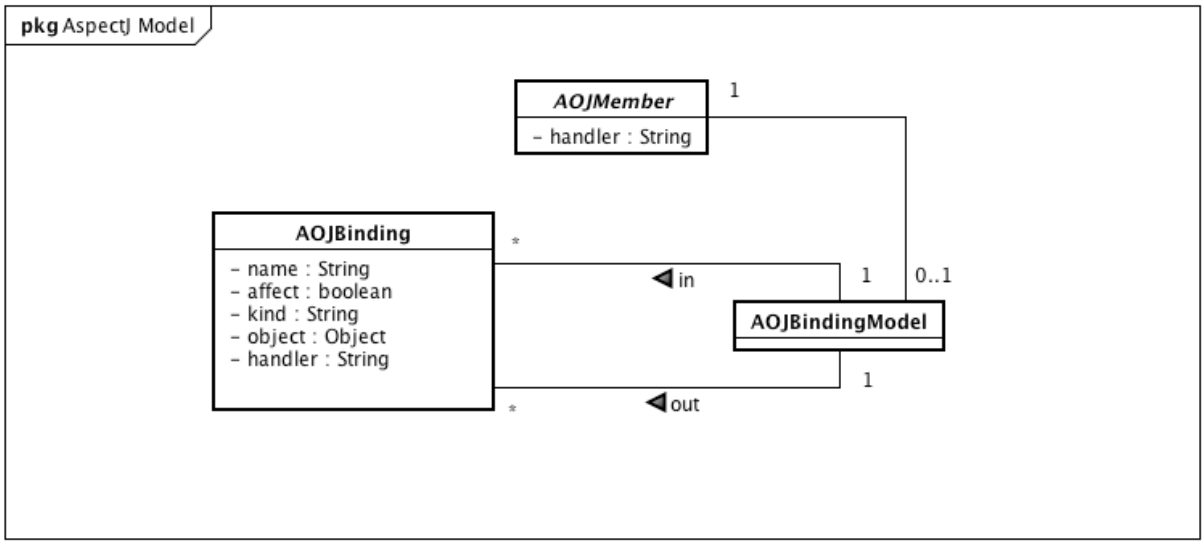


Figura C.12 – Modelo de ligação

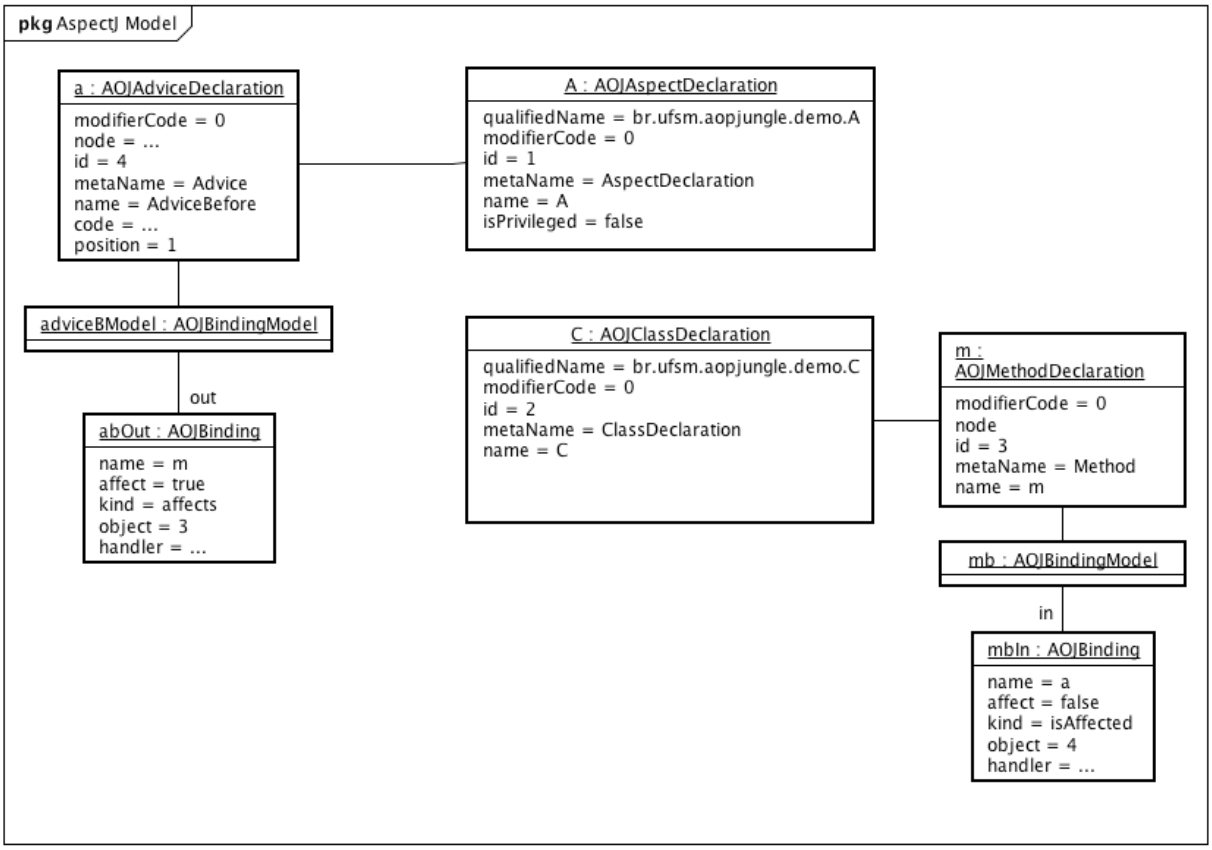


Figura C.13 – Exemplo de instância do modelo de ligação

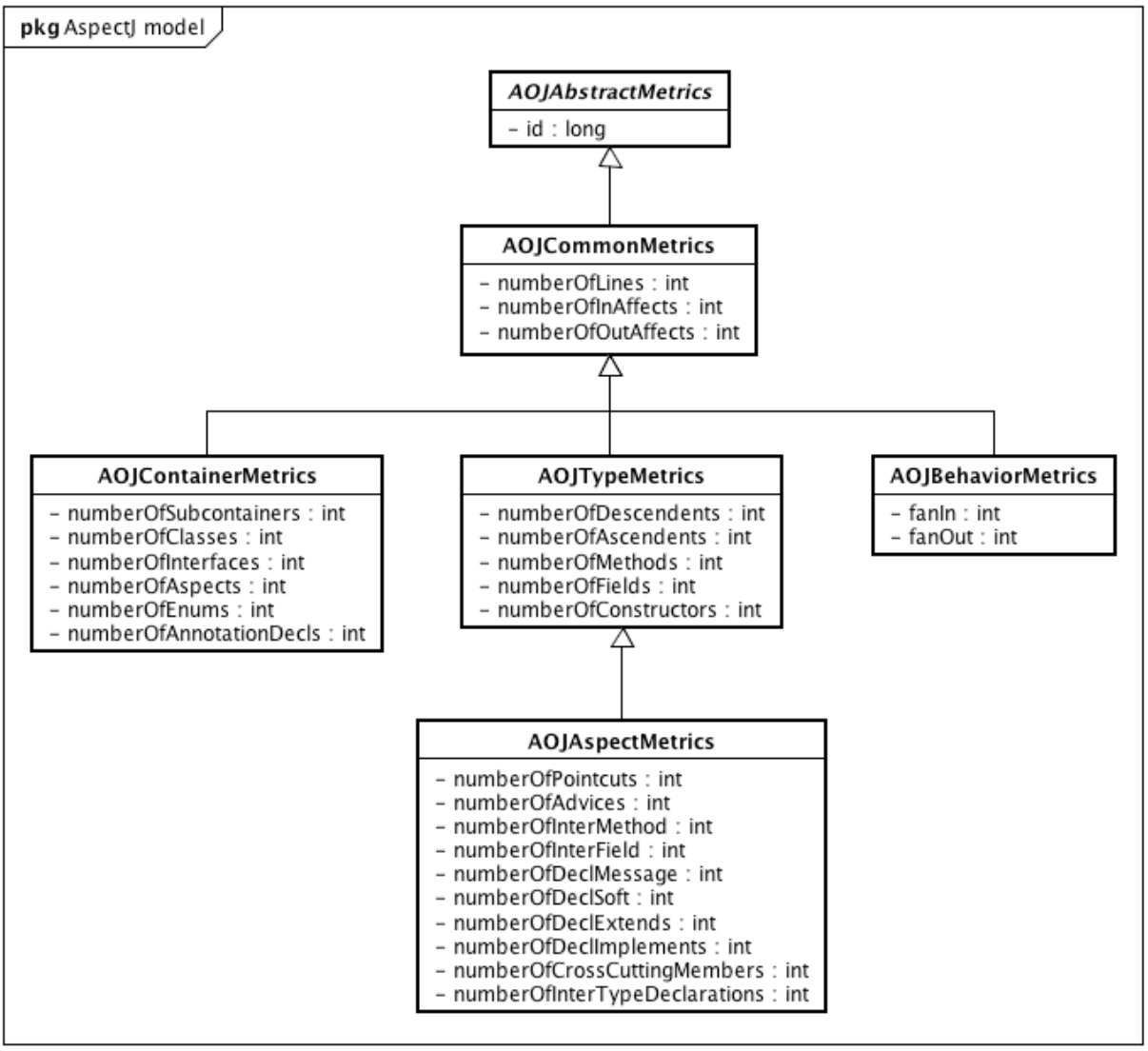


Figura C.14 – Modelo de métricas para AspectJ