Um Processo para o uso de Linguagens de Consulta em Código Fonte

Gustavo Stangherlin Cantarelli[1], Cristiano De Faveri[1], Eduardo Kessler Piveta[1]

[1]Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brasil

gus.cant@gmail.com, cristiano@defaveri.com.br, piveta.inf@ufsm.br

Abstract. *The search in the source code is gaining more and more space because of the increasing complexity of current software systems and also the need for improvements in source code. Although the paradigms of object-oriented programming and aspect-oriented programming have several features to improve code reuse and clarity when maintenance of code is required, developers tend to reduce productivity because of problems on locating the parts to be corrected or improved. Aiming maintenance activities, this paper presents a search code process that can be applied to source code repositories.*

Resumo. *As buscas em código-fonte estão ganhando cada vez mais espaço devido ao atual aumento da complexidade dos sistemas de software e também à necessidade de melhorias em código-fonte. Embora os paradigmas de programação orientada a objetos e de programação orientada a aspectos possuam diversos recursos para melhorar o reuso e a clareza de código, quando é necessária a manutenção de trechos de código, os programadores tendem a reduzir sua produtividade em função de problemas em localizar os trechos a serem corrigidos ou melhorados. Visando as atividades de manutenção, este trabalho apresenta um processo de consulta que pode ser aplicado em repositórios de código-fonte.*

1. Introdução

As tecnologias de consulta em código possuem um importante papel na análise de um sistema computacional. À medida que as aplicações se tornam cada vez maiores, com centenas de módulos, manter a qualidade do código é um constante desafio. A consulta em código é uma técnica a qual permite pesquisar desde elementos de um programa, suas relações e dependências, até cenários completos, catalogados como oportunidades de refatoração [Fowler 1999], [Moonen, et al. 2002], [Schumacher, et al. 2010]. A aplicação de técnicas de consulta em código pode ser encontrada em diferentes áreas, tais como análise de arquitetura [Feijs, et al. 1998], engenharia reversa [Feijs, et al. 1998], [Holt 1998], garantia de uso de convenções [Verbaere, et al. 2007], identificação de oportunidades de refatoração [Hovemeyer, et al. 2004],[Pmd 2012] e busca por interesses transversais [Marin, et al. 2007].

Em linguagens orientadas a aspectos (OA) [Kiczales, et al. 1997], a busca em código desempenha também um relevante papel. Se por um lado propriedades como transparência e quantificação [Filman, et al. 2000] promovem melhorias em temas como decisão tardia de projeto [Laddad 2003] e descrição concisa de interesses [Filman, et al. 2000], por outro permitem a interferência de forma sistêmica no comportamento de componentes de uma aplicação, tornando, muitas vezes, seu entendimento mais complexo [Pfeifer, et al. 2005],[Salom 2007]. Por exemplo, a simples tarefa de

identificar quais módulos são afetados por um determinado adendo pode envolver diversas inspeções, em diferentes códigos fontes. A ausência de ferramentas que auxiliem na busca e na visualização de pontos de junção afetados por aspectos pode levar ao programador a cometer erros durante a modularização de interesses transversais [Pfeifer, et al. 2005].

Dentre as tecnologias de consulta em código desenvolvidas durante os últimos anos, diferentes abordagens têm sido empregadas em relação à forma de extrair e armazenar os metadados de um programa, em relação à aplicação de operações e consultas sobre os metadados armazenados, e em relação à apresentação dos resultados. Algumas abordagens realizam o processo de extração dos metadados e os armazenam em repositórios com diferentes formatos, tais como relacionais ou XML. A consulta então é realizada com linguagens como SQL [Elmasri, et al. 2011] e XQuery [Walmsley 2007]. Outras abordagens implementam uma linguagem específica de domínio para a realização da consulta, como por exemplo as linguagens JQuery [McCormick, et al. 2004] e JTL [Cohen, et al. 2006].

Este trabalho propõe um processo de busca em código fonte, que possibilita tanto o uso de linguagens de consulta existentes quanto servir de apoio para a criação de novas linguagens de consulta. Para avaliação do uso do processo proposto, foram feitas duas instanciações como prova de conceito, sendo uma para o paradigma de Programação Orientada a Objetos (POO) e uma para o paradigma de Programação Orientada a Aspectos (POA). A primeira instanciação mostra como o processo pode ser usado para buscar construções em código orientado a objetos usando a linguagem de consulta SPARQL em uma ontologia. A segunda, por sua vez, mostra o processo usado para código orientado a aspectos, utilizando a linguagem de consulta XQuery em dados armazenados em arquivos XML. O objetivo destas escolhas de modelos de representação (Ontologias e XML) é mostrar diferentes visões de instanciação para validação do processo.

Para a instanciação OO, foi desenvolvida uma ferramenta implementada em Java (na plataforma Eclipse), chamada OopExtract, que realiza percursos em Árvores Sintáticas Abstratas (ASTs) para a extração de metadados em código-fonte Java. Para popular a ontologia utilizada na primeira instanciação, foi utilizada a API Jena [Asf 2012c].

A instanciação Orientação a Aspectos (OA) usou os resultados da ferramenta AOPJungle, a qual foi desenvolvida pelo Grupo de Linguagens de Programação e Bancos de Dados da Universidade Federal de Santa Maria (UFSM).

2. Trabalhos Relacionados

Linguagem para busca em código é uma linguagem de computador usada para realizar consultas em arquivo de código fonte, escrito em uma linguagem de programação definida, representada através de uma estrutura apropriada [Fowler, et al. 2010].

Existem diversas linguagens de consulta para códigos, dentre as quais se destacam, principalmente, JQuery, Lost [Pfeifer, et al. 2005] e JTL, sendo JQuery e JTL linguagem voltadas a código OO e Lost para programas orientados a aspectos e JQuery e JTL .

JQuery é um *plugin* da plataforma Eclipse para a manipulação de código OO, usando um banco de dados a partir do código-fonte, fornecendo meios para consultar os

dados constantes em tal banco. Após a execução da consulta, é apresentada uma hierarquia em formato árvore com nós e sub nós que representam as classes e seus métodos. Essas consultas são construídas através de predicados, tendo por base uma linguagem de programação lógica chamada TyRuBa [Volder 1998], implementada em Java. A Figura 1 mostra o código de uma consulta, executada em JQuery, que busca todos os pacotes, cujas classes possuem um método que comece com "d".

```
Package(?P), child(?P, ?CU), child(?CU, ?M), child(?M, ?T), type(?T),
name (?M,?name), re_match(/^d,?name)
```

Figura 1. Exemplo de consulta utilizando JQuery (PFEIFER, SARDOS e GURD, 2005).

A ferramenta Lost é um *plugin* da plataforma Eclipse a qual possibilita buscas em código OO e código OA e visa obter maior precisão na consulta para código e navegação nos resultados por meio de árvore [Pfeifer, et al. 2005]. A proposta de Lost é utilizar os benefícios da JQuery, tais como navegador universal de código e consultas iterativas ao programa OA, utilizando AspectJ. A Figura 2 mostra uma consulta em Lost para buscar todos os métodos cujos nomes contenham "set" ou "put" e estejam sendo afetados por adendos do aspecto "formatChecker" porém, que não sejam afetados (via adendos) pelo aspecto "policyEnforcer".

```
Select Method where Method.hasName("set|put")
where Method.isAdvisedBy(Aspect)
where !Method.isAdvisedBy(Aspect[2])
where Aspect.hasName("formatChecker") &&
Aspect[2].hasName("policyEnforcer")
```

Figura 2. Exemplo de consulta OA utilizando Lost (PFEIFER, SARDOS e GURD, 2005).

A linguagem Java Tools Language (JTL) é uma linguagem cuja finalidade é efetuar buscas em trechos de código para Java, baseando-se em um banco de dados relacional como representação (ao invés de uma AST). Em particular, os autores citam o futuro uso da JTL em buscas por pontos de junção (POA), podendo se tornar uma alternativa à linguagem de seleção de pontos de junção de AspectJ, e outros elementos genéricos utilizados em códigos derivados da linguagem Java.

A Figura 3 mostra um exemplo de consulta que verifica se uma classe é considerada "clássica", ou seja, é concreta, possui pelo menos um campo e alguns métodos de leitura e métodos de escrita distintos.

```
classical := class members: {
   has field;
   many method;
   no static;
   method => public;
   field => private;
   disjoint setter, getter;
}
```

Figura 3. Exemplo de consulta em classes utilizando JTL (COHEN e YOSSI, 2006).

3. Um Processo para o Uso de Linguagens de Consulta em Código Fonte

Visando a auxiliar o desenvolvedor na busca em códigos fonte em paradigmas OO e AO este trabalho consiste na especificação de um processo genérico para a criação ou para o uso de linguagens de consulta para a busca em código-fonte.

O processo consiste de duas etapas: Definição e Instanciação. O processo inicia na etapa de Definição, na qual são selecionados os elementos a serem mapeados e os modelos de representação, e finaliza com a etapa de Instanciação, na qual é criada uma linguagem de consulta ou utilizada uma ferramenta existente de extração de metadados e consultas para a obtenção dos resultados desejados.

Os pré-requisitos para a definição do processo são:

- Possuir um repositório de código;
- Conhecer quais resultados se deseja alcançar;
- Possuir conhecimento da existência de ferramentas para a definição do processo ou possuir conhecimento para a criação de uma ferramenta, e
- Estar em conformidade com o SPEM[1] (*Software Process Engineering Metamodel*).

Nas Figuras 4 e 5 são apresentados o fluxo das atividades do processo que ocorre nas fases de Definição e Instanciação e uma visão geral do processo, respectivamente. Na visão geral do processo são enfatizados o executor das tarefas e os artefatos consumidos e produzidos durante e após a execução de cada tarefa.

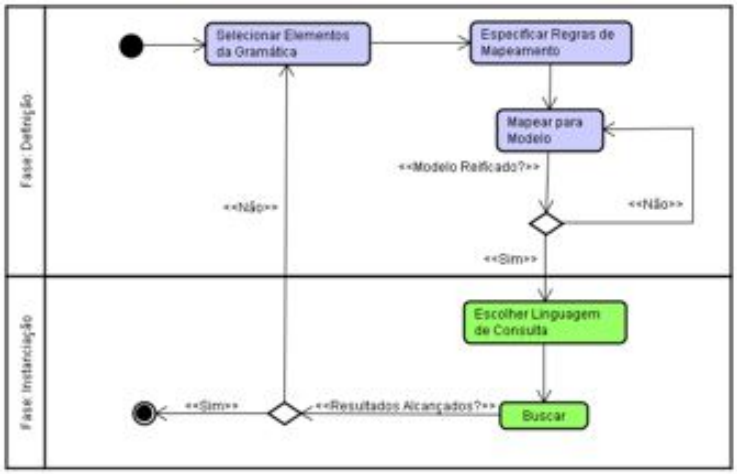

Figura 4. Diagrama de atividades do processo.

3.1 Selecionar elementos da gramática

A atividade inicial possui dois artefatos de entrada: Gramática e Linguagem Alvo.

Gramática: As linguagens de programação são criadas a partir da definição de gramáticas. As gramáticas são constituídas por um conjunto de sentenças, as quais são formadas através de regras e símbolos terminais e não terminais [Sebesta 2011] . A finalidade da utilização de uma gramática como artefato de entrada do processo proposto, é que por meio dela, pode-se verificar qual caminho percorrer (através de árvores sintáticas) para encontrar determinados elementos em código-fonte.

Linguagem Alvo: Este artefato especifica qual linguagem de programação será utilizada, tal como: Java, C, C++, C#, dentre outras. Pode ser considerada uma especificação menos formal (orientação) com a finalidade de "nortear" o processo.

[1] Disponível em: http://www.omg.org/spec/SPEM/2.0/PDF/

Como resultado da execução desta atividade, é obtido o artefato de saída: **Elementos Selecionados**. Neste artefato são apresentados os elementos gramaticais que proporcionarão a busca em trechos de código-fonte desta mesma linguagem. Cada elemento da gramática será selecionado de acordo com sua estrutura e necessidade de busca, servindo de referência para a extração de metadados. Por exemplo, caso o usuário queira buscar trechos de código em programas escritos em uma linguagem orientada a objetos, seria interessante que ele fosse capaz de se referir a classes, atributos ou métodos.

Esta atividade é dependente de um objetivo específico e de conhecimento da linguagem de programação a ser pesquisada.

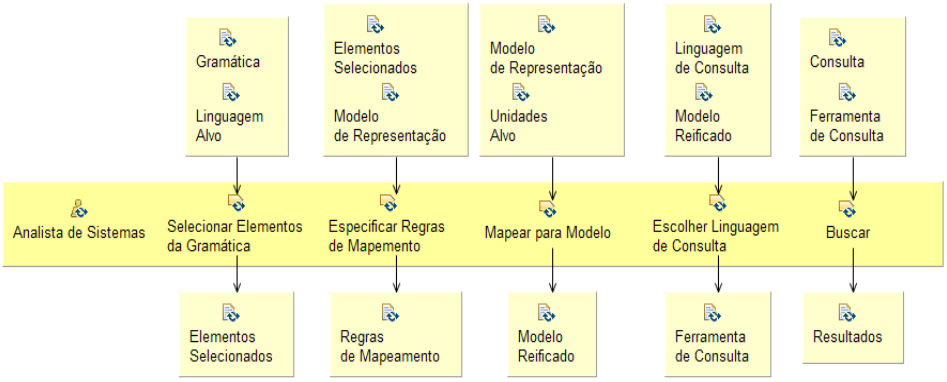

Figura 5. Visão detalhada do processo.

3.2 Especificar as regras de mapeamento

A criação de regras de mapeamento é necessária para que seja possível definir como os elementos do código-fonte serão obtidos e armazenados.

Para esta atividade, é necessário o artefato gerado na atividade anterior, **Elementos Selecionados** e, também deter o conhecimento de como os metadados serão disponibilizados:

Modelo de Representação: é definido por um Analista de Sistemas e será utilizado para a organização e o armazenamento dos metadados durante a extração dos mesmos. Dependendo do propósito da instanciação do processo, o modelo de representação pode ser um modelo relacional de banco de dados, um arquivo XML, uma ontologia, dentre outros. A execução desta atividade resulta em regras que serão usadas para a extração de metadados e sua representação. Como artefatos de saída, são obtidas as **Regras de Mapeamento**.

Regras de Mapeamento: este artefato mostra como os elementos selecionados são mapeados para o modelo de representação quando encontrados em programas, tais como classes com seus modificadores, atributos, métodos dentre outros elementos. Como exemplo, pode ser citado: um Aspecto, na gramática, passa para o formato <aspect> em linguagem XML.

3.3 Mapear para modelo

Com a especificação de regras para mapear e extrair metadados é feita a seleção das unidades alvo que serão mapeadas (processo de reificação) para o modelo reificado. Além do artefato **Regras de Mapeamento**, esta tarefa também utiliza o artefato **Unidades Alvo**. Após a execução desta tarefa, o artefato Modelo Reificado é obtido.

Unidades Alvo: representa o arquivo que contém o código fonte a ser lido e/ou um repositório de vários arquivos (pasta ou espaços de trabalho) que seguem a mesma gramática, sendo, portanto, a base da qual o processo extrairá os metadados.

Modelo Reificado: caracteriza-se por tornar informações sobre a estrutura de um programa (metadados) disponível no modelo reificado e pronto para consultas.

A reificação pode ser feita com a implementação de uma ferramenta focada nos objetivos a serem alcançados, por meio da especificação de requisitos. Na Tabela 1 são mostrados alguns possíveis requisitos que poderiam ser utilizados na construção de uma ferramenta de auxílio ao processo.

Tabela 1. Especificação de requisitos de uma ferramenta para busca em código fonte.

Requisitos Funcionais		
Identificador	Nome	Descrição
RF01	Selecionar Unidades Alvo	Implementar uma classe (ou interface) para a escolha de Unidades Alvo.
RF02	Implementar Regras de Mapeamento	Implementar (ou estender) métodos que atendam às Regras de Mapeamento.
RF03	Definir Linguagem de Consulta	Escolher ou criar uma linguagem de consulta que atenda aos objetivos do processo.
RF04	Efetuar buscas	Implementar buscas na linguagem de consulta escolhida, de modo que forneçam resultados que possam ser analisados claramente.
Requisitos Não Funcionais		
Identificador	Nome	Descrição
RNF01	Escolher sistema operacional	Definir o sistema operacional cuja aplicação será executada.
RNF02	Escolher linguagem de programação	Definir a linguagem de programação para a implementação da aplicação.
RNF03	Preparar forma de armazenamento de informações	Preparar o Modelo de Representação a ser usado no armazenamento das informações.

3.4 Escolher linguagem de consulta

Com base no **Modelo Reificado**, pode-se ter opções de escolha de Linguagens de Consulta existentes dependendo da especificação das regras e dos modelos. Com a escolha de uma linguagem de consulta, podem ser utilizadas ferramentas prontas ou implementadas exclusivamente para o objetivo das consultas, obtendo o artefato **Ferramenta de Consulta**.

Linguagens de Consulta: refere-se à escolha de uma linguagem de consulta já existente, tais como SQL, XQuery, SPARQL, etc., ou a criação de uma nova linguagem específica para a consulta, como por exemplo, uma linguagem específica de domínio (DSL). Tal escolha poderá ser utilizada em uma ferramenta (implementada para o propósito) ou usada diretamente (uso nativo).

Ferramenta de Consulta: é o mecanismo de consultas nos metadados extraídos. Pode ser a implementação de uma ferramenta apropriada com uso de linguagens de consultas existentes (ou criadas para domínios específicos).

A escolha da linguagem de consulta varia de acordo com os objetivos a serem alcançados e o modelo de representação escolhido. Podem ser utilizadas linguagens de

consultas já existentes ou, caso nenhuma delas proporcione o resultado esperado, desenvolver uma DSL.

3.5 Buscar

Ao final do processo têm-se a atividade de busca e, como artefato de saída, os Resultados. Por meio da análise dos resultados poderão ser encontrados erros no código fonte ou oportunidades de refatoração, por exemplo.

4. Instanciação do Processo

Para avaliar a aplicabilidade do processo proposto, foram realizadas duas instanciações distintas sob a ótica de dois paradigmas em diferentes linguagens de programação, em diferentes modelos de representação e em diferentes linguagens de consulta. A primeira delas para o paradigma OO utilizou-se um modelo de ontologia (linguagem OWL) e linguagem de consulta SPARQL. Para o paradigma OA, a linguagem XML foi utilizada juntamente com a linguagem de consulta XQuery.

4.1 Instanciação OO

Para a instanciação OO, usando o modelo de ontologias, foram utilizados: a linguagem Java (na plataforma Eclipse), o analisador sintático contido na JDT e a API Jena. Visando a facilitar a extração de metadados, foi desenvolvida uma ferramenta, denominada OOPExtract, para efetuar a leitura de código Java e, com a análise das ASTs geradas, inserir os metadados em instâncias no modelos de representação da ontologia.

A ferramenta OOPExtract faz a análise de um conjunto de arquivos (contidos em um diretório especificado estaticamente) por execução. A cada execução, a ferramenta efetua a busca pelos elementos (Tabela 2) e insere as informações como um novo indivíduo na ontologia criada.

Em relação ao uso de ASTs, foram usadas três classes para a identificação de elementos presentes no código fonte: (i) a classe **TypeDeclaration,** responsável pela identificação de classes e interfaces do código, (ii) a classe **FieldDeclaration** usada para identificar os atributos e anotações de cada classe presente no código, e (iii) a classe **MethodDeclaration** usada para identificar os métodos e construtores de cada classe e exceções (de cada método) contidos no código.

Tabela 2. Elementos selecionados para a instanciação OO

Elemento	Descrição	Sintaxe Analizada
Classe ou Interface	A AST considera uma classe ou interface como um tipo *TypeDeclaration*. Com isso podem ser extraídos diversos elementos que, sintaticamente, a compõe.	**modificador** "**class**" ["**interface**"] nome_classe ["**extends**" nome_superclasse] ["**implements**" nome_interface] "{" Sendo: • **modificador**: conjunto de um ou mais modificadores (ex:. public abstract). • **extends**: Usado caso a classe seja a especialização de outra. • **implements**: Usado caso a classe implemente uma interface.
Atributo	São denominados *FieldDeclaration*s, contidos dentro de um	**modificador** "**tipo**" nome_atributo"**;**" Sendo:

	TypeDeclaration. Abrangem variáveis e anotações.	• **modificador**: conjunto de um ou mais modificadores (ex:. public abstract). • **tipo**: tipo de dado da variável (ex.: int, String, tipos abstratos, entre outros).
Anotação	As anotações também são detectadas pela AST como um *FieldDeclaration*. Porém caracterizam-se pelo uso do identificador "@". Para este trabalho, foram considerados apenas os nomes das anotações.	@nome_anotacao Exemplo: @Produces("application/xml")).
Método	São detectados no interior de um *TypeDeclaration*, por meio de um *MethodDeclaration*.	**modificador** "**tipo_retorno**" nome_metodo "(" [**parametros**] ")" Sendo: • **modificador**: conjunto de um ou mais modificadores (ex:. public abstract). • **tipo_retorno**: tipo de dado da variável (ex.: int, String, tipos abstratos, entre outros). • **parâmetros**: lista de parâmetros necessários ao método. A ausência de parâmetros é caracterizada pelo uso do tipo de dado **void**.
Exceção	São identificadas dentro de um *MethodDeclaration*. Para este trabalho foram apenas consideradas as exceções declaradas juntamente com o método através da cláusula **throws**.	Declaração de método seguida do comando **throws** e lista de um ou mais tipos de exceções. Exemplo: **throws** ServletException, IOException.

Para a obtenção da instanciação OO, foi utilizada a API Jena na ferramenta OopExtract. Enquanto a ferramenta OOPExtract extrai as informações a partir da API do JDT, a inclusão de novos elementos em uma instância da ontologia é realizada. Também foram utilizadas a ferramenta Protégé, a linguagem de representação OWL e a linguagem de consultas SPARQL.

Foram criadas classes, propriedades (*Object Properties*), e tipos de dados (*Data Properties*), conforme pode ser observado na Figura 6. Para os tipos de dados criados, foi utilizado apenas o tipo primitivo String.

Após popular a ontologia é possível visualizar graficamente as informações. A Figura 7 mostra o relacionamento de duas classes com seus respectivos atributos, anotações, métodos e exceções.

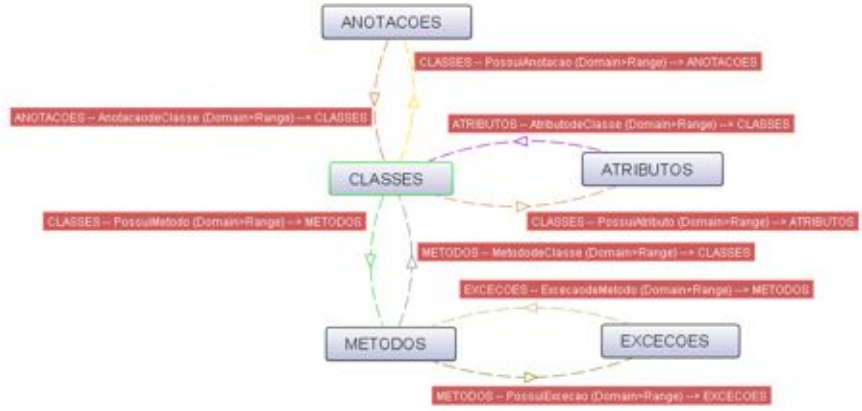

Figura 6. Ontologia OO.

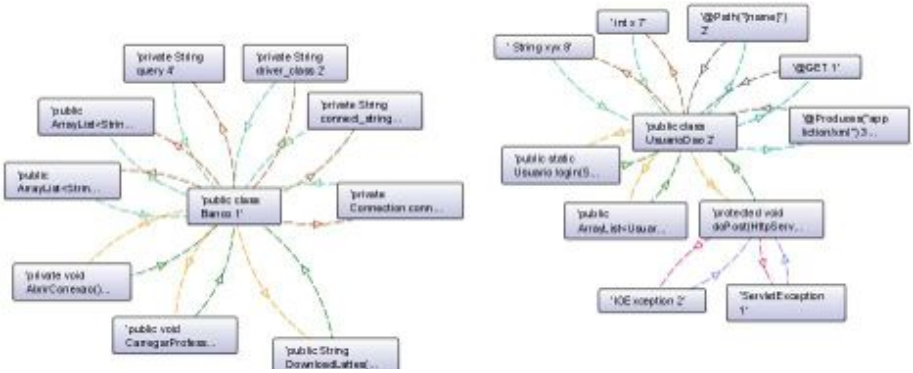

Figura 7. Relacionamento entre indivíduos da ontologia usada.

A Figura 8 mostra o código fonte de consulta, utilizando SPARQL, que lista todos os atributos das classes através da propriedade PossuiAtributo.

```
1 PREFIX tb: <http://www.owl-ontologies.com/Ontology1338318212.owl#>
2 SELECT ?classe ?atributo
3 WHERE {
4     ?classe tb:PossuiAtributo ?atributo.
5 }
6 ORDER BY ?classe ?atributo
```

Figura 8. Código da consulta por atributos.

Após a execução é obtido o resultado conforme a Figura 9.

classe	atributo
'public class Banco 1'	'private Connection conn 1'
'public class Banco 1'	'private String driver_class 2'
'public class Banco 1'	'private String connect_string 3'
'public class Banco 1'	'private String query 4'
'public class Banco 1'	'public ArrayList<String> lstlinks 5'
'public class Banco 1'	'public ArrayList<String> lstprofessores 6'
'public class UsuarioDao 2'	' int x 7'
'public class UsuarioDao 2'	' String xyx 8'

Figura 9. Resultado da consulta por atributos.

4.2 Instanciação OA

Para a instanciação OA e XML, a ferramenta AOPJungle foi utilizada para a extração de metadados do programa. Além da utilização do processo proposto neste trabalho, a ferramenta AOPJungle tem como objetivo atingir níveis de granularidade mais refinados do código fonte OA, fornecendo um metamodelo OO reificado do programa, no qual os elementos, os relacionamentos e as dependências entre os módulos de um sistema são representados.

Para a exemplificação de instanciação deste trabalho, foi usado um resultado prévio fornecido pela ferramenta AOPJungle, o qual consiste de metadados extraídos de código fonte escritos em linguagem Java com AspectJ e representados em um arquivo XML. Para esta instanciação, foram selecionados alguns elementos presentes em código AO (Tabela 3).

Tabela 3. Elementos selecionados para a instanciação POA.

Elemento	Descrição	Sintaxe Analizada
Aspecto	Os aspectos possibilitam a separação de interesses. Dentro deles são declarados os pontos de corte e adendos. Também podem, além de estender outros aspectos, estender uma classe ou implementar interfaces.	**modificadores** "**aspect**" nome_aspecto ["**extends**" nome_classe_aspecto] ["**implements**" nome_interface] "{" Sendo: • **modificador**: conjunto de um ou mais modificadores (ex:. public abstract).

		• **extends**: Usado caso a classe seja a especialização de outro aspecto ou classe. • **implements**: Usado caso o aspecto implemente uma interface.
Intertipo	São declarações que alteram estaticamente atributos, métodos ou construtores ao programa.	**modificadores** "**tipo**" nome_intertipo"**;**" Sendo: • **modificadores**: conjunto de um ou mais modificadores (ex:. public abstract). • **tipo**: tipo de dado que a variável irá suportar (ex.: int, String, tipos abstratos, entre outros).

Adendo	São responsaveis pela execução das ações associadas aos pontos de corte. Podem ser dos tipos *before*, *after* ou *around*.	tipo_advice **():** nome_pointcut "**{**"
		Sendo:
		• **tipo_advice**: *after, before* ou *around*.
		
		Exemplo:
		after() returning : testepointcut() {
		System.out.println("Teste");
		}

Ponto de corte	É um conjunto de pontos de junção agrupados sintaticamente pelo AspectJ. Dentre eles podem ser citados: chamadas de métodos, *handlers* e adendos.	**Pointcut** nome_pointcut() **:** expressao_chamada**;** Sendo: • **expressão_chamada**: onde o PointCut irá ser chamado. Ex:. call (void metodoteste()).
Warning	São mensagens de aviso (*warning*) ou erros (*errors*) contidas no interior do aspecto.	**declare error :** nome_pointcut **:** "mensagem"**;** **declare warning :** nome_pointcut **:** "mensagem"**;**

A partir do arquivo XML gerado pela ferramenta AOPJungle, foram elaboradas algumas consultas utilizando XQuery. A Figura 10 mostra o código-fonte elaborado para uma listagem dos aspectos e seus respectivos pontos de corte.

```
<Projetos>{
let $doc := .
for $project in $doc//*[ends-with(name(), "AOJProject")]
return
    <Projeto Nome = "{$project/name}">
        <Pacotes>{
        for $package in $project//*[ends-with(name(),"AOJPackageDeclaration")]
        return
           <Pacote Nome = "{$package/name}">{
                for $aspect in $package//*[ends-with(name(),"AOJAspectDeclaration")]
                return
                    <Aspecto Nome = "{$aspect/name}">{
                        for $pointCut in $aspect//*[ends-with(name(),"AOJPointcutDeclaration")]
                        return
                            <PontoCorte Expressão = "{$pointCut/pointcutExpression/code}"></PontoCorte>
                    }</Aspecto>
           }</Pacote>
        }</Pacotes>
    </Projeto>
}</Projetos>
```

Figura 10. Código fonte da consulta por aspectos e pontos de corte.

A Figura 11 mostra como resultado obtido apenas a estrutura dos programas lidos, seus pacotes, aspectos e pontos de corte.

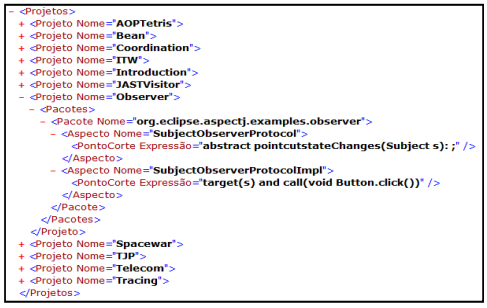

Figura 11. Resultado da consulta por aspectos e pontos de corte.

5. Considerações Finais

As instanciações realizadas neste trabalho mostram que o processo proposto pode ser usado em diferentes contextos (paradigmas, linguagens, modelos de representação, linguagens de consulta). Tanto a instanciação OO quanto a OA mostram que independente dos paradigmas e/ou linguagens alvo a serem consultadas, o processo para a obtenção dos resultados nos metadados permanece o mesmo.

Na instanciação OO, foi implementada uma ferramenta (OopExtract) cujas funcionalidades básicas são ler código-fonte e popular uma instância da ontologia. As consultas podem ser feitas com linguagens criadas para o modelo de representação, como, por exemplo, SPARQL.

Na instanciação OA, foram usados documentos XML gerados pela ferramenta AOPJungle e as consultas foram realizadas com o uso da linguagem XQuery.

6. Referências

Asf. (2012c). What is Jena?. http://jena.apache.org/index.html. Maio de 2012.

Cohen, T., e Yossi, J. (2006). JTL the Java Tools Language. SIGPLAN Not. (pp. 89-108). ACM.

Elmasri, R., e Navthe, S. B. (2011). Sistemas de Banco de Dados. São Paulo: Pearson Addison Wesley.

Feijs, L., Krikhaar, R., e Van Ommering, R. (1998). A relational approach to support software architecture analysis. Softw. Pract. Exper. 28. John Wiley & Sons, Inc.

Filman, R. E., e Friedman, D. P. (2000). Aspect-Oriented Programming is Quantification and Obliviousness. RIACS. RIACS.

Fowler, M. (1999). Refactoring - Improving the Design of Existing Code. Boston, MA, US: Addison-Wesley.

Fowler, M., e Parsons, R. (2010). Domain-Specific Languages. Boston: Addison-Wesley.

Holt, R. C. (1998). Structural manipulations of software architecture using Tarski relational algebra. WCRE'98. IEEE Computer Society.

Hovemeyer, D., e Pugh, W. (2004). Finding bugs is easy. OOPSLA '04 , pp. 132--136.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Chris Lopes, C. V., Loingtier, J.-M., et al. (1997). Aspect-oriented programming. ECOOP. SpringerVerlag.

Korth, H. F., Silberschatz, A., e Sudarshan, S. (2006). Sistema de Banco de Dados. São Paulo: Makron Books.

Laddad, R. (2003). AspectJ in Action - Practical Aspect-Oriented Programming. Greenwich, CT, USA: Manning Publications Co.

Marin, M., Moonen, L., e Deursen, A. v. (2007). SoQueT - Query- based documentation of crosscutting concerns. ICSE'07. IEEE.

McCormick, E., e De Volder, K. (2004). JQuery - finding your way through tangled code. Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.

Moonen, E., e Emden, L. V. (2002). Java Quality Assurance by Detecting Code Smells. WCRE'02. 0, p. 97. Los Alamitos: IEEE Computer Society.

Pfeifer, H. J., Sardos, A., e Gurd, J. R. (2005). Complex Code Querying and Navigation for AspectJ. Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, (pp. 60-64).

Pmd (2012). http://pmd.sourceforge.net/. Dezembro de 2012.

Pugh, B. (2012). Findbugs. Acesso em Janeiro de 2013, disponível em http://findbugs.sourceforge.net/

Salom, E. (2007). A Query-Based Approach for the Analysis of Aspect-Oriented Systems. Dissertação de mestrado, University of Waterloo.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., e Shaw, M. (2010). Building empirical support for automated code smell detection. ESEM' 10. New York.

Sebesta, R. W. (2011). Conceitos de linguagens de programação (9ª ed.). Porto Alegre: Bookman.

Verbaere, M., Hajiyev, E., e Moor, O. d. (2007). Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. ACM.

Volder, K. D. (1998). Type-Oriented Logic Meta Programming. Vrije Universiteit Brussel. Brussels, Belgium: Vrije Universiteit Brussel.

Walmsley, P. (2007). XQuery. O'Reilly Media, Inc.