Revista Brasileira de Computação Aplicada, Novembro, 2025

DOI: 10.5335/rbca.v17i3.16482
Vol. 17, Nº 3, pp. 84–99
Homepage: seer.upf.br/index.php/rbca/index

ARTIGO ORIGINAL

Um processo para buscas não estruturadas em código fonte

A process for source code unstructured search

Rodrigo de Castro Gil [1], Eduardo Kessler Piveta[1], Cristiano De Faveri[2], Deise de Brum Saccol[1], Lisandra Manzoni Fontoura[1]

[1]Universidade Federal de Santa Maria, [2]AMF - Antonio Meneghetti Faculdade

rcgil@inf.ufsm.br; *piveta@inf.ufsm.br; cristiano.faveri@amf.edu.br; deise@inf.ufsm.br; lisandra@inf.ufsm.br

Recebido: 16/11/2024. **Revisado:** 14/11/2025. **Aceito:** 30/11/2025.

Resumo

A consulta em código fonte representa um importante recurso para auxiliar desenvolvedores na compreensão de programas, bem como em atividades de refatoração, especialmente em grandes repositórios de código. Este trabalho apresenta um processo cujo objetivo é facilitar a recuperação de informação, utilizando técnicas de consultas não estruturadas em código fonte. O processo apresenta arquivos classificados por ordem de importância com base nas características da linguagem de programação e nas preferências definidas pelos desenvolvedores. Foram realizadas duas instanciações do processo, nas linguagens de programação Java e AspectJ, demonstrando a aplicação do processo e como ele pode facilitar a identificação das informações recuperadas. O objetivo geral é permitir buscas mais precisas em programas escritos em diferentes linguagens de programação.

Palavras-Chave: Busca não estruturada; Processos de Software; AHP

Abstract

Source-code querying is an important resource for assisting developers in program comprehension and refactoring tasks, particularly in large code repositories. This work presents a process designed to facilitate information retrieval by employing techniques for unstructured queries over source code. The process ranks files according to their relevance, based on both the characteristics of the programming language and the preferences defined by developers. Two instantiations of the process were carried out—using the Java and AspectJ programming languages—demonstrating its application and how it can support the identification of retrieved information. The overarching goal is to enable more precise searches in programs written in different programming languages.

Keywords: Unstructured search; Software Processes; AHP.

1 Introdução

A evolução de código é uma prática contínua no contexto de desenvolvimento de software. Os motivos pelos quais um sistema de software é alterado refletem diretamente no tipo de manutenção realizada. A norma ISO/IEC 14764 define a manutenção de software como sendo reativa ou pró-ativa (ISO, 2022). De forma reativa, a manutenção do código é realizada para corrigir defeitos evidenciados em produção (manutenção corretiva) ou para adaptar um sistema a um novo ambiente (manutenção adaptativa). Pró-ativamente, um desenvolvedor pode modificar código já em produção para corrigir defeitos conhecidos e latentes antes que se tornem falhas efetivas (manutenção preventiva). Por fim, problemas de desempenho e manutenibilidade podem levar ao refinamento de código para torná-lo mais legível e mais rápido (manutenção perfectiva).

À medida que os sistemas de software se tornam maiores e mais complexos, manter a qualidade do código gerado é um desafio permanente. Antes de efetuar uma modifica-

ção em um sistema, os desenvolvedores devem explorar o código fonte e identificar quais partes são relevantes e candidatas à manutenção. Para ajudá-los a compreender o código e identificar possíveis oportunidades de refatoração, os desenvolvedores geralmente recorrem a recursos de busca de um ambiente integrado de desenvolvimento (IDE) e ferramentas de análise estática (Piveta, 2009; Robillard e Murphy, 2007; Hecht et al., 2006; Fokaefs et al., 2007).

A busca de código é uma característica básica dos IDEs usados profissionalmente, tais como Eclipse, IntelliJ e VS-Code. Em geral, os recursos de busca possuem pesquisas simples por palavras e também recursos mais avançados usando expressões regulares e construções da linguagem, tais como classes e métodos. Por outro lado, as ferramentas de análise estática são comumente usadas para buscar oportunidades de refatoração e também defeitos no código que podem resultar em falhas.

Para auxiliar nesse processo, linguagens de consulta em código fonte (de Faveri, 2013) e mecanismos de busca (Basu, 2024; *NerdyData*, 2024; *Krugle*, 2024) têm sido propostos, algumas restritas a sistemas orientados a objetos, outras para linguagens de consulta para programas orientados a aspectos e também motores de busca não estruturada em código fonte. Essas abordagens auxiliam os desenvolvedores não apenas a encontrar oportunidades de refatoração, mas também, em outras tarefas rotineiras como localizar um trecho de código para reutilizá-lo ou simplesmente navegar pelo código de um sistema a fim de entender o seu funcionamento.

Apesar de existirem maneiras para representar (Piveta et al., 2009) e para buscar por oportunidades de refatoração por meio de percurso em ASTs (*Abstract Syntax Trees*) de programas usando ferramentas de análise estática, tais abordagens fornecem uma flexibilidade limitada e são dependentes de conhecimentos de baixo nível das linguagens de programação associadas.

Dentro desse contexto, este artigo apresenta um processo que fornece suporte à busca de informações de forma não estruturada em código fonte. O processo busca fornecer elementos que possibilitem que a recuperação de informação seja possível independente da linguagem ou do paradigma adotados no repositório no qual se deseja aplicar o processo, considerando a importância dos elementos da linguagem.

De forma a avaliar o processo proposto são apresentadas duas instanciações distintas do processo, para as linguagens Java (orientada a objetos) e AspectJ (orientada a aspectos). Tais linguagens foram selecionadas por serem representativas em seus paradigmas correspondentes. Java é uma das linguagens orientadas a objetos mais amplamente usada atualmente. E AspectJ foi a primeira linguagem orientada a aspectos proposta, e continua sendo a mais popular. Busca-se mostrar como o processo proporciona resultados mais precisos para as consultas realizadas e fornece a classificação dos arquivos buscados de acordo com a linguagem e os parâmetros utilizados.

O restante deste artigo está organizado da seguinte forma. A Seção 2 descreve alguns conceitos que auxiliam na definição do processo proposto. A Seção 3 descreve um processo para buscas não estruturadas, incluindo as atividades, papéis, e artefatos associados. A Seção 4 mostra as duas instanciações feitas para avaliar o processo (para Java e para AspectJ). Por fim, a Seção 5 descreve alguns trabalhos relacionados e a Seção 6 elenca as principais conclusões deste trabalho.

2 Referencial Teórico

Esta seção descreve um conjunto de conceitos e ferramentas necessários para o entendimento deste trabalho. Ela está organizada da seguinte forma. A Seção 2.1 descreve conceitos relacionados à evolução de programas, refatoração e consulta em código fonte. A Seção 2.2 apresenta conceitos sobre o AHP, um método de decisão multi-critérios.

2.1 Evolução, Refatoração e Consulta em Código

Uma característica que se pode observar em sistemas de software é a necessidade de eles evoluírem. O processo de manutenção ou evolução envolve de forma geral três atividades: compreensão da sistema atual, modificação do sistema atual e reavaliação do sistema modificado (Pressman e Maxim, 2021). Ao adaptar, melhorar e modificar um sistema, seu projeto pode se afastar de sua concepção original, diminuindo sua qualidade (Mens e Tourwé, 2004).

As mudanças representam características cruciais no desenvolvimento de software (Godfrey e German, 2008). Várias métricas e leis da evolução de software podem ser seguidas para obter melhores resultados no processo evolutivo, como mudança contínua, aumento da complexidade e auto-regulação, entre outras (Lehman et al., 1997). Apesar de extensa pesquisa e progresso significativo nesta área, o desenvolvimento e a manutenção de sistemas de software continuam sendo processos demorados e dispendiosos. Portanto, a redução do custo de desenvolvimento e manutenção de software permanece um tema de pesquisa vital na engenharia de software (Pizka e Jurgens, 2007). Uma das técnicas que auxilia na manutenção de software é a refatoração.

Refatoração pode ser definida como o processo de modificar a estrutura interna de um sistema de software de maneira que não afete seu comportamento externo perceptível (Fowler, 1999). O procedimento de refatoração comumente abrange a simplificação da lógica condicional, aprimoramento da estrutura do código e eliminação de trechos duplicados (Kerievsky, 2004). Os resultados gerados pelo sistema refatorado devem ser idênticos aos produzidos antes da refatoração. Para identificar oportunidades de refatoração, é necessário examinar as estruturas das aplicações. Dessa maneira, ferramentas que realizam consultas em código fonte são instrumentais para esse propósito.

A ausência de refatoração pode acarretar a progressiva deterioração do projeto de um programa. Ao longo do tempo, o código é sujeito a modificações, e conforme evolui de forma *ad hoc*, sua integridade pode ser comprometida, afastando-se da estrutura original delineada. A leitura do código torna-se progressivamente mais desafiadora à medida que a estrutura se degrada. Refatorar, assim, assemelha-se a organizar o código, envolvendo a remoção de segmentos que não ocupam a posição adequada.

A perda de estrutura no código tem efeitos cumulativos, como destacado por Fowler (Fowler, 1999).

A identificação de trechos de código passíveis de refatoração consiste na busca por locais nos quais melhorias podem ser aplicadas, levando em consideração deficiências, inadequações ou incompletudes (Júnior et al., 2019). Dessa maneira, oportunidades de refatoração podem ser identificadas, estabelecendo uma relação de associação entre uma deficiência ou limitação e um padrão de refatoração específico (Piveta, 2009). Ferramentas automatizadas de detecção de oportunidades de refatoração desempenham um papel crucial nesse contexto.

2.2 AHP

AHP (*Analytic Hierarchy Process*) (Saaty, 1990) é um método multicritério amplamente utilizado e conhecido no apoio à tomada de decisão durante a resolução de conflitos negociados, em problemas usando múltiplos critérios (Vargas e IPMA-B, 2010). O AHP fornece um meio de decomposição de um problema em uma hierarquia de subproblemas que podem ser mais facilmente compreendidos e avaliados. Para esse processo, o AHP possui os seguintes passos:

Passo 1: O problema é decomposto em uma hierarquia de critérios, subcritérios e alternativas. A Fig. 1 mostra uma estrutura hierárquica genérica. A raiz da hierarquia é a meta ou objetivo do problema a ser estudado e sintetizado, os nós folha são as alternativas a serem comparadas. Entre estes dois níveis estão diferentes critérios e sub-critérios.

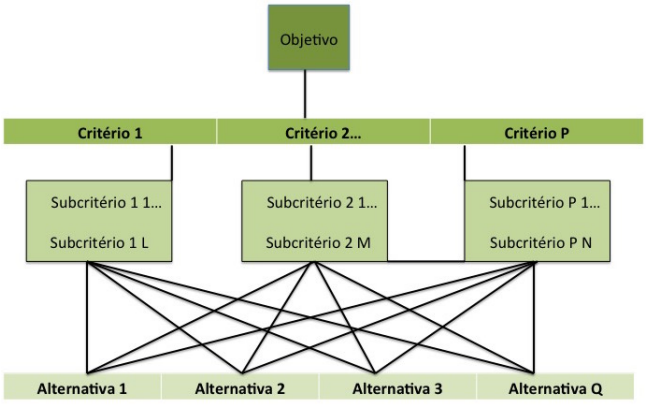

Figura 1: Hierarquia genérica do AHP (Bhushan e Rai, 2004)

Passo 2: Neste passo o especialista define as prioridades para cada critério e alternativa. A importância de cada critério em relação aos outros e de cada alternativa em relação às outras é apurada utilizando comparação em pares. A Tabela 1 (Saaty, 1990), é usada para expressar numericamente a importância relativa sobre critérios e alternativas.

Passo 3: As comparações par a par dos vários critérios gerados no passo anterior são organizadas em uma matriz quadrada. Os elementos da diagonal principal da matriz têm o valor 1. O critério da linha (**i**) é melhor que o da coluna (**j**) se o valor do elemento (**i, j**) for maior que 1. Caso contrário o critério da coluna (**j**) é melhor que o da linha (**i**). Os dois são equivalentes se o valor do elemento (**i,j**) for igual a 1.

Tabela 1: Escala gradativa de comparação quantitativa das alternativas

Valor	Importância Relativa
1	Igual importância
2	Ligeiramente mais importante
3	Fracamente mais importante
4	Fracamente a moderadamente mais importante
5	Moderadamente mais importante
6	Moderadamente a fortemente mais importante
7	Fortemente mais importante
8	Extremamente mais importante
9	Absolutamente mais importante

Passo 4: Neste passo é construído um vetor com os pesos relativos para todos os critérios da matriz do passo anterior. Esse vetor representa os pesos de cada critério. Esse vetor é obtido através da elevação ao quadrado da matriz de prioridade, obtida no passo anterior, e são somados os valores de suas linhas, para que cada um desses valores seja dividido pela soma do total das linhas. Esses valores são calculados e normalizados.

Passo 5: A consistência da matriz de ordem **n** é avaliada. As comparações feitas por este método são subjetivas e o AHP tolera inconsistência através da quantidade de redundância na abordagem. Se este índice de consistência não conseguir chegar a um nível desejado, respostas para as comparações devem ser re-examinadas. Essa relação de consistência (CR) é dada pela razão:

$$CR = \frac{CI}{RCI}$$

Onde o índice de consistência, CI, é calculado como:

$$CI = \frac{(\lambda_{max} - n)}{(n-1)}$$

onde λ_{max} é o autovalor máximo da matriz de julgamento. Este CI pode ser comparado com o de uma matriz aleatória, RCI. Os detalhes do cálculo RCI são discutidos por Saaty (Saaty, 1990), o qual fornece os valores de RCI para serem usados no cálculo da CR. E esses valores são apresentados na Tabela 2. Ele sugere ainda que o valor da CR deve ser inferior a 0,1 ou seja, o processo é considerado aceitável se a CR for inferior a 10%.

Tabela 2: Valores referentes ao RCI

n	**< 3**	**3**	**4**	**5**	**6**	**7**	**8**
RCI	0	0,58	0,9	1,12	1,24	1,32	1,41
n	9	10	11	12	13	14	15
RCI	1,45	1,49	1,51	1,48	1,56	1,57	1,59

Passo 6: A classificação de cada alternativa é multipli-

cada pelos pesos dos subcritérios e agregada para obter classificações locais em relação a cada critério. As classificações locais são então multiplicadas pelos pesos dos critérios e agregadas para obter as classificações globais. O AHP produz então valores dos pesos para cada alternativa baseados no julgamento de importância de cada alternativa em relação às outras com respeito a um critério comum.

3 Um Processo para Buscas não Estruturadas em Código Fonte

Buscando auxiliar os desenvolvedores nas buscas em código fonte, este trabalho consiste na especificação de um processo para a busca de informação de forma não estruturada, otimizando assim o tempo gasto com a análise dos resultados obtidos, e diminuindo a necessidade de refinamento das consultas para melhorar os resultados.

O processo possibilita que um determinado provedor de ferramentas possa viabilizar a classificação de elementos de uma ou mais linguagens de programação. E que essa classificação determine a ordem de retorno dos arquivos relevantes em consultas realizadas por desenvolvedores. Por exemplo, uma vez que um determinado provedor tenha definido a importância relativa dos elementos de programação escritos em uma dada linguagem de programação, um desenvolvedor qualquer pode efetuar buscas não estruturadas em repositórios desta linguagem e obter os resultados ordenados de acordo com essa importância.

O processo proposto consiste em três etapas: **Definição**, **Reificação** e **Análise**. O processo inicia na etapa de **Definição**, na qual são selecionados os elementos relevantes da linguagem de programação. Esses elementos são priorizados, e é definida e avaliada uma função de classificação. A segunda etapa é a **Reificação**. Nesta etapa, é criado um modelo reificado para que as consultas sejam executadas nesse modelo. A terceira etapa, na qual o processo termina, é a etapa de **Análise**. Nela são escritas as consultas, que são processadas e executadas e seus resultados são classificados e analisados.

Como cada etapa do processo pode ser desempenhada separadamente optou-se, para um melhor entendimento, por detalhá-las em separado. Para que qualquer processo seja realizado, é preciso definir quem será responsável por executar as tarefas e por analisar os artefatos de entrada e a integralidade dos artefatos produzidos. Isso faz com que seja necessário definir os papéis que serão desempenhados durante o processo. Para este processo, foram definidos dois papéis: o **Tool Provider**, que é o responsável por dar início ao processo e decidir quais as ferramentas serão utilizadas na sua efetivação, e o **Analista**, o qual realizará as consultas.

A seguir são descritas as etapas do processo. A Seção 3.1 descreve a etapa de **Definição**, suas atividades e artefatos, a Seção 3.2 mostra a etapa de **Reificação**, a qual gera um metamodelo que pode ser manipulado. A Seção 3.3 detalha as atividades da etapa de **Análise**, na qual as consultas são executadas e os resultados obtidos.

3.1 Etapa de Definição

O objetivo desta etapa é priorizar elementos (selecionados a partir de uma determinada linguagem de programação), e definir uma função de classificação que terá a responsabilidade de classificar os arquivos retornados como relevantes nas consultas realizadas. Esta etapa possui as seguintes atividades: *Selecionar conceitos*, *Priorizar elementos*, *Definir função de classificação* e *Avaliar função de classificação*. A Fig. 2 mostra essas atividades. A seguir são detalhadas as atividades da etapa de **Definição**.

3.1.1 Selecionar conceitos

O objetivo desta atividade é selecionar as estruturas da linguagem de programação que devem ser consideradas importantes para a busca. Esta atividade tem como entrada os artefatos *Requisitos de busca* e *Linguagem alvo* e gera o artefato *Conceitos*. A Fig. 3 mostra a atividade com os seus artefatos, os quais são detalhados a seguir:

- **Requisitos de busca:** lista quais os objetos que serão objetivos das buscas. Por exemplo, *Deve ser possível buscar por nomes de pacotes,* ou, *Deve ser possível buscar por tipo de adendos*.
- **Linguagem alvo:** define em qual linguagem de programação as consultas serão realizadas. Por exemplo, pode ser utilizada uma linguagem OO como Java, ou uma linguagem estruturada como C, dentre outras.
- **Conceitos:** define quais elementos de uma determinada linguagem de programação serão considerados relevantes. Um *Conceito* pode ser uma classe, um aspecto, dentre outros. Estes elementos serão a referência para a extração dos metadados. Por exemplo, caso o processo seja empregado na linguagem de programação Java e fosse necessário buscar por nomes de pacotes, o artefato *Conceitos* definiria os *Pacotes Java*, como um desses elementos.

3.1.2 Priorizar elementos

Esta atividade tem como objetivo fornecer os elementos de uma linguagem de programação priorizados. Para tal, ela recebe o artefato *Conceitos*, gerado na atividade anterior e juntamente com os artefatos *Opiniões de especialistas* e *Método de priorização*, fornece esses elementos priorizados de forma quantitativa, ou seja, com valores para que possam ser utilizados na próxima atividade. Como artefato de saída, essa atividade produz o artefato *Priorização*. A Fig. 4 mostra a atividade *Priorizar elementos*, e a seguir são detalhados os artefatos desta atividade:

- **Opiniões de especialistas:** descrevem a prioridade relativa entre itens definidos no artefato *Conceitos*. Por exemplo, se o processo for aplicado em uma linguagem orientada a objetos, as opiniões de especialistas podem definir que uma ocorrência do objeto da busca em uma classe deve ter um valor maior que a mesma ocorrência em um método.
- **Método de priorização:** Um *Método de priorização* possibilita quantificar os itens dos *Conceitos*, a partir das definições das *Opiniões de especialistas*. Este método pode ser um método estatístico, um método de decisão multicritério, como o AHP apresentado na Seção 2, ou

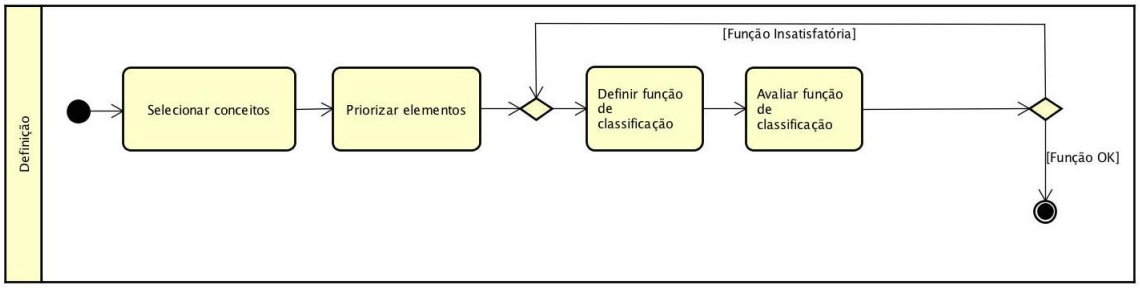

Figura 2: Etapa de Definição

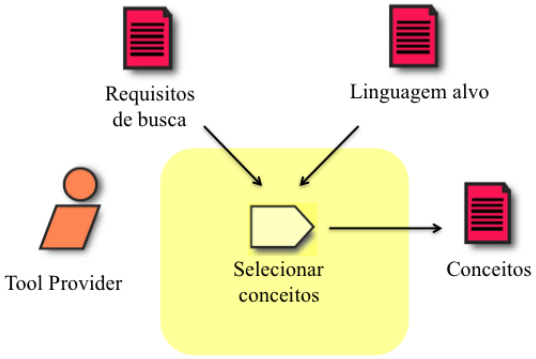

Figura 3: Atividade *Selecionar conceitos*

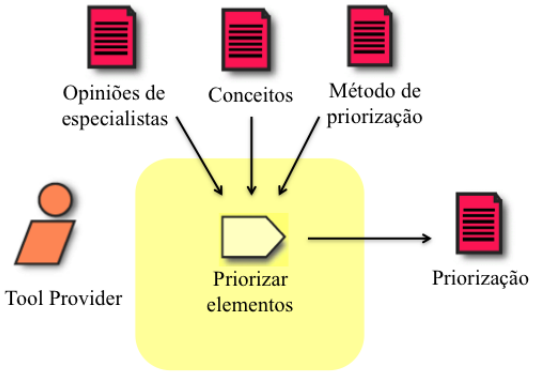

Figura 4: Atividade *Priorizar elementos*

outro método que o **Tool Provider** achar conveniente. Esta atividade tem como saída a *Priorização*, que servirá de entrada para a próxima atividade.

- **Priorização:** é uma lista a qual retrata as estruturas de uma determinada linguagem de programação, classificadas em ordem de importância. Também é possível que essa lista possua pesos para cada um de seus itens. Por exemplo, em uma linguagem OO essa lista poderia ser representada da seguinte forma (usando pesos hipotéticos):

Ordem	Estrutura	Peso
1º	Pacote	0,3
2º	Classe	0,2
3º	Método	0,1
...	...	...

3.1.3 *Definir função de classificação*

O objetivo desta atividade é definir uma função para ser aplicada em cada unidade de compilação que seja retornada como relevante durante a realização das consultas. A Fig. 5 mostra a atividade *Definir função de classificação* que recebe a *Priorização*, gerada na atividade anterior, e tem como saída uma *Função de classificação*, a qual é detalhada a seguir:

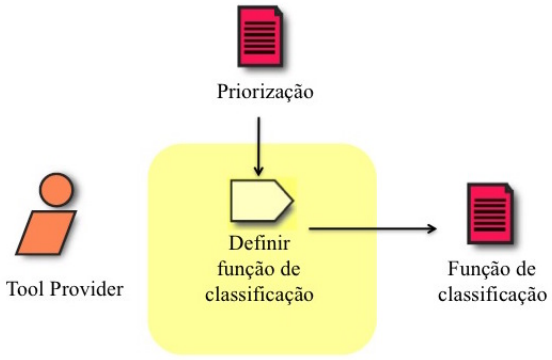

Figura 5: Atividade *Definir função de classificação*

- **Função de classificação:** define a função que será aplicada para classificar as unidades de compilação para as consultas realizadas. Por exemplo, essa função poderia ser representada pelo peso de uma determinada estrutura de código, multiplicado pelo número de vezes que o objeto da consulta aparece nessa estrutura. Por exemplo, se considerássemos apenas a ocorrência de um termo em métodos, a representação da função poderia ser: $f(t) = (nEmMétodos(t) * pesoMétodos)$, onde **n** seria o número de ocorrências de **(t)** em métodos e o **peso** seria o valor dado à importância dos métodos, com base nas *Opiniões de especialistas*.

3.1.4 *Avaliar função de classificação*

Esta atividade tem como objetivo avaliar se a *Função de classificação* contempla as necessidades definidas pelo **Tool Provider**. Para essa avaliação, esta atividade necessita da realização da **Reificação**, e também, das atividades da **Análise**.

A Fig. 6 mostra a atividade *Avaliar função de classificação*, a qual recebe como artefatos de entrada: *Função de classificação*, gerado na atividade anterior, os *Programas de teste* e as *Consultas de teste*, e tem como artefato de saída a *Avaliação*. Abaixo os artefatos *Programas de teste*, *Consultas de teste* e *Avaliação*, são detalhados:

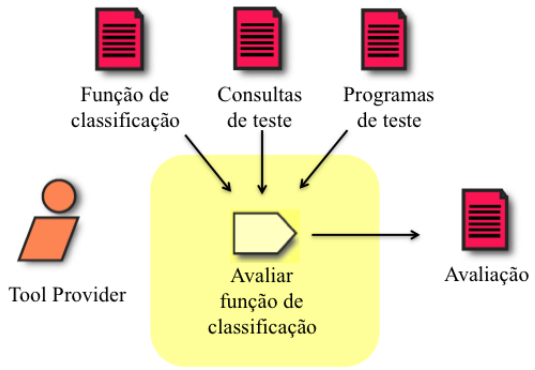

Figura 6: Atividade *Avaliar função de classificação*

- **Programas de teste:** São programas em um repositório de código fonte, que serão utilizados para que sejam executadas as consultas para que possa ser analisada a classificação dos seus resultados.
- **Consultas de teste:** São consultas realizadas para que sejam analisados os resultados da busca e a classificação desses resultados.
- **Avaliação**: Este artefato define se a função de classificação atende os critérios de importância definidos pelo usuário na atividade *Priorizar elementos*. Esta avaliação pode ser feita usando métricas de medição de desempenho em recuperação de informação como a Precisão e a Revocação, ou pode ser feita de maneira empírica, ou seja, baseada no conhecimento do decisor.

3.2 Etapa de Reificação

A etapa de **Reificação** é responsável por transformar os elementos de código fonte em informações estruturais, ou seja, metamodelos que serão disponibilizados para a utilização de várias formas. Esta etapa possui apenas uma atividade: *Reificar programas*. O motivo pelo qual optou-se por deixar essa etapa separada, apesar de ter apenas uma atividade, foi que ela pode ter que ser desempenhada periodicamente para atualizar a base de dados consultada. No caso da base de dados ser pequena, esta etapa pode ser desempenhada toda a vez que novos arquivos de código fonte forem adicionados, alterados ou excluídos. Por outro lado, se o processo for aplicado em um repositório de tamanho expressivo, esta etapa terá que ser realizada de maneira periódica, como é feito em alguns repositórios disponíveis na Web.

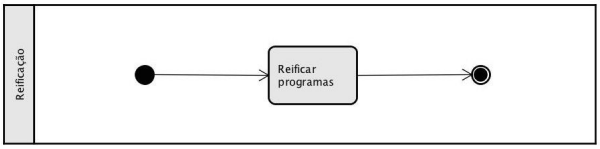

Figura 7: Etapa de Reificação

Esta atividade é a responsável por transformar os arquivos de código fonte em um metamodelo que será usado para realizar as consultas. A Fig. 8 mostra a atividade que tem como entrada os *Conceitos* e as *Unidades de compilação*, e como saída o *Modelo Reificado*, os quais são detalhados a seguir:

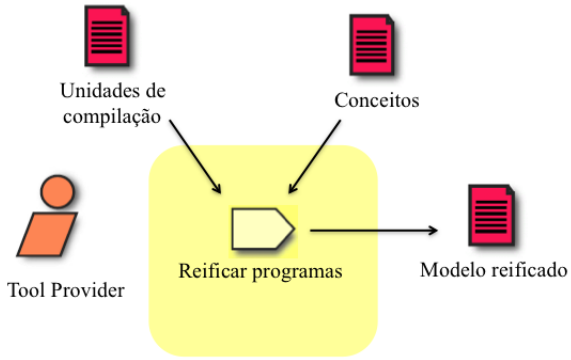

Figura 8: Atividade de *Reificar programas*

- **Unidades de compilação:** São os arquivos que compõem o repositório onde o processo será aplicado. Este artefato pode representar um repositório pessoal, de uma empresa ou um repositório disponível na nuvem como: *Github*, *Google Code*, *BitBucket*, entre outros.
- **Modelo Reificado:** Representa as informações sobre a estrutura dos programas, disponíveis e prontas para serem consultadas, ou seja, são os metadados dos programas em um metamodelo que será manipulado.

3.3 Etapa de Análise

O objetivo desta etapa é que a partir da necessidade de encontrar determinados termos de consulta, o **Analista** escreva uma consulta, e sejam encontrados os arquivos que contém esses termos, e esses arquivos sejam apresentados para o **Analista** em ordem de relevância com base nas definições da etapa de **Definição**. Possivelmente a etapa de **Análise** será a mais desempenhada do processo, pois as outras etapas só são realizadas para melhorar os resultados da **Análise**. A Fig. 9 apresenta o fluxo desta etapa, a qual possui as seguintes atividades: *Escrever consulta*,

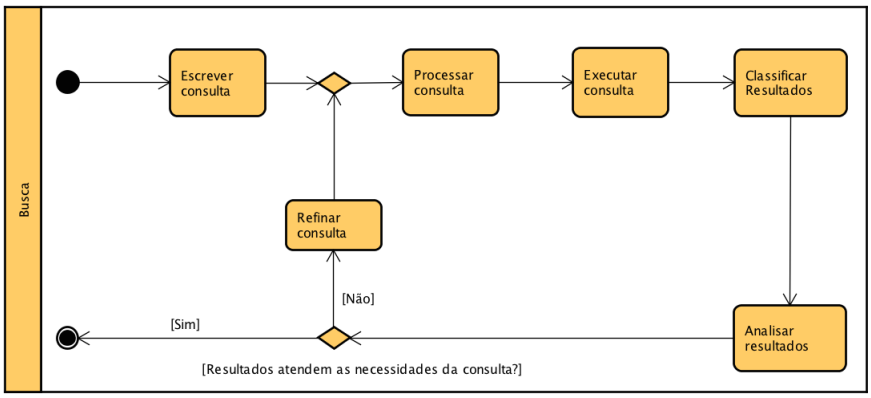

Figura 9: Etapa de Análise

Processar consulta, *Executar consulta*, *Classificar resultados*, *Analisar resultados* e *Refinar consulta*, que serão detalhadas nas próximas seções.

3.3.1 *Escrever consulta*

Nesta atividade, o *Analista* transcreve para uma consulta os termos que ele deseja encontrar. A Fig. 10 mostra a atividade *Escrever consulta* a qual produz uma *Consulta escrita*, que é detalhada a seguir:

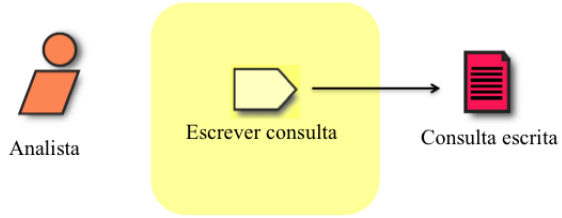

Figura 10: Atividade *Escrever consulta*

- **Consulta escrita:** Este artefato representa a consulta feita pelo **Analista**. Como qualquer consulta não estruturada, essa consulta pode ser apenas uma sequência de caracteres, ou pode ter algum refinamento, dependendo do motor de busca utilizado. No caso deste artefato possuir mais de um termo, a *Função de classificação* será aplicada nos **n** termos da consulta. A *Consulta escrita* servirá como um dos artefatos de entrada para a próxima atividade.

3.3.2 *Processar consulta*

Esta atividade é responsável por submeter a *Consulta escrita* ao mesmo processamento ao qual foram submetidas as *Unidades de compilação* do repositório utilizado. Esta atividade sendo executada por um sistema automatizado, dispensaria o papel do **Tool Provider**. Como artefatos de entrada, esta atividade possui a *Consulta escrita*, gerada na atividade anterior, e o *Modelo de processamento*, e, como saída, a *Consulta processada*. A Fig. 11 apresenta a atividade *Processar consulta*. A seguir são detalhados os artefatos desta atividade:

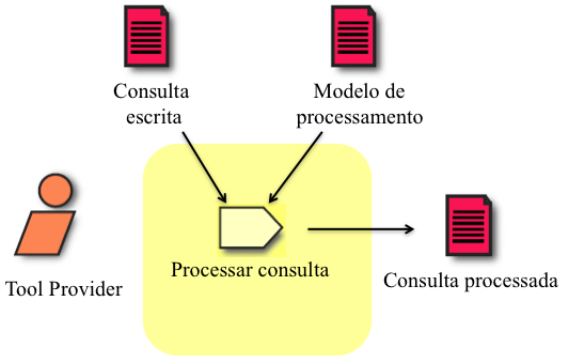

Figura 11: Atividade *Processar consulta*

- **Modelo de processamento:** O *Modelo de processamento* depende do motor de busca utilizado para a realização das consultas, e necessariamente precisa seguir os mesmos passos de processamento da atividade *Reificar programas* realizada na etapa de **Reificação**.
- **Consulta processada:** Este artefato representa os termos da consulta processados com base no artefato *Modelo de processamento*. Este processamento é feito de forma transparente ao **Analista**.

3.3.3 *Executar consulta*

A atividade *Executar consulta* tem o objetivo de buscar no repositório alvo os documentos que satisfazem os termos do artefato *Consulta processada*, para que esses arquivos possam ser classificados na próxima atividade. Esta atividade recebe o artefato *Consulta processada*, produzido na atividade anterior mais o artefato *Modelo Reificado* gerado na etapa de **Reificação** e juntamente com o artefato *Motor de busca*, produz o artefato *Documentos retornados*. A Fig. 12 mostra esta atividade, e seus artefatos são detalhados a seguir:

- **Motor de busca:** Este artefato representa o motor de

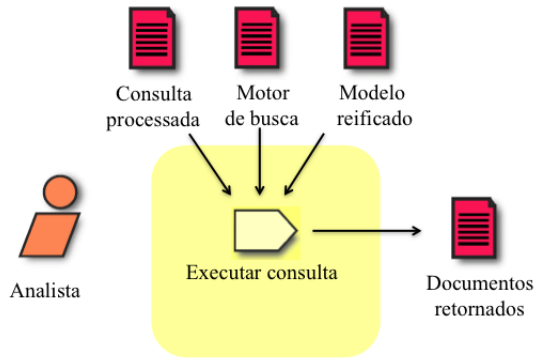

Figura 12: Atividade *Executar consulta*

busca que será utilizado para executar as consultas, tal como o *Code Finder* (de Faveri, 2013).

- **Documentos retornados:** Caracteriza-se pelos arquivos que serão retornados como relevantes na execução de uma determinada consulta. Este artefato servirá como entrada para a próxima atividade. Por exemplo:

```
Arquivo1.java
Arquivo2.java
Arquivo3.java
...
```

3.3.4 *Classificar resultados*

O objetivo desta atividade é apresentar para o **Analista** os documentos retornados como relevantes de uma maneira que seja possível afirmar que esses documentos estão em ordem do mais relevante para o menos relevante. A Fig. 13 exibe a atividade *Classificar resultados* juntamente com seus artefatos de entrada *Documentos retornados* e *Função de classificação* e também o seu artefato de saída *Resultados classificados*, o qual é detalhado abaixo:

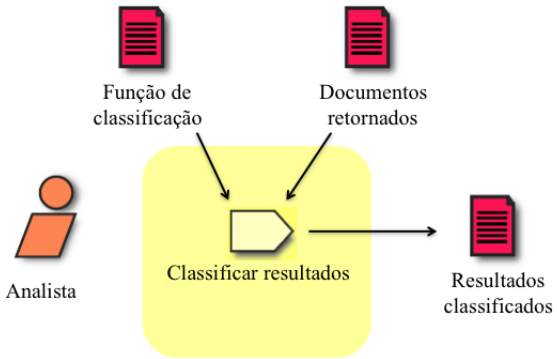

Figura 13: Atividade *Classificar resultados*

- **Resultados classificados:** Caracteriza-se por apresentar os *Documentos retornados* classificados conforme a *Função de classificação*, ou seja, classificados com as preferências do usuário. Por exemplo a lista apresentada no artefato *Documentos retornados*, poderia ser alterada desta forma:

```
Arquivo2.java
Arquivo3.java
Arquivo1.java
...
```

3.3.5 *Analisar resultados*

O Objetivo desta atividade é avaliar se os *Resultados classificados* atendem às necessidades do **Analista**. A Fig. 14 mostra esta atividade juntamente com seus artefatos, o quais são detalhados a seguir:

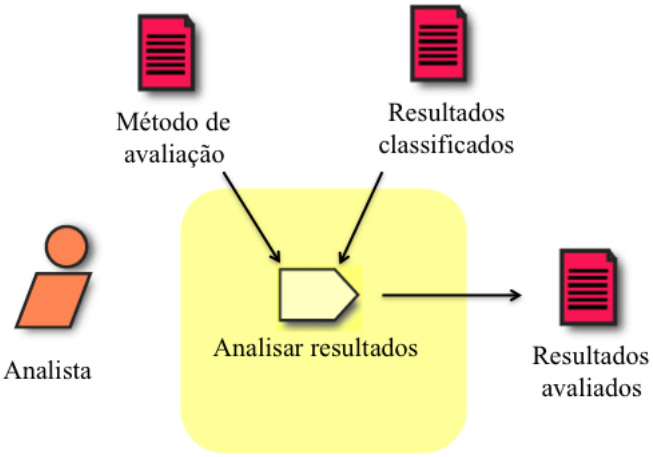

Figura 14: Atividade *Analisar resultados*

- **Método de avaliação:** Descreve a maneira pela qual será avaliado o resultado das consultas realizadas. Essa avaliação pode ser utilizando algum método pré-definido, ou simplesmente através da visualização dos resultados.
- **Resultados avaliados:** Define se as necessidades da consulta foram atendidas ou se será necessário realizar a atividade de *Refinar consulta*, ou até mesmo reiniciar o processo novamente.

3.3.6 *Refinar consulta*

A Fig. 15 mostra a atividade *Refinar consulta* a qual não é uma atividade obrigatória, ela apenas possibilita o refinamento de uma determinada consulta, se houver necessidade, para melhorar o seu resultado. Esta atividade, apesar de ser muito comum em consultas não estruturadas, diminui a necessidade de que o processo seja reiniciado desde o início. Porém, a ideia deste processo é que não seja preciso reescrever as consultas várias vezes. Esta atividade tem como artefato de entrada a *Consulta escrita*, e como saída produz o artefato *Consulta refinada*, que é detalhado abaixo:

- **Consulta refinada:** Representa a consulta produzida na atividade *Escrever consulta*, escrita de uma forma que possa produzir um resultado melhor, caso possível.

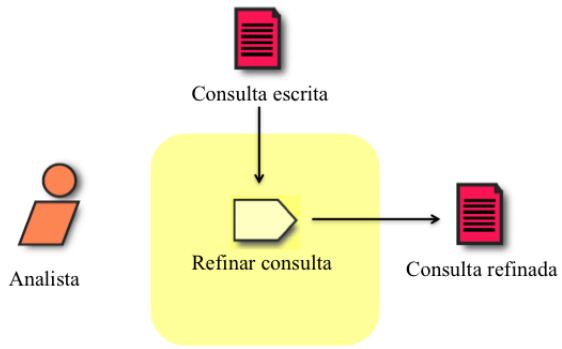

Figura 15: Atividade *Refinar consulta*

4 Avaliação

De forma a avaliar o processo proposto, foram feitas duas instanciações. A primeira para a linguagem Java e a segunda para a linguagem orientada a aspectos AspectJ. Para cada, são exemplificadas duas consultas e os resultados dessas consultas, analisados. Para as instanciações do processo, foi utilizado o método de priorização multi-critério AHP (Saaty, 1990), para reificar os arquivos foi usado o *framework* AOPJungle (de Faveri, 2013) e como motor de busca foi adotado o Code Finder (de Faveri, 2013) e implementado com base no Apache Lucene.

4.1 Instanciação para Java

Esta seção apresenta uma possível instanciação do processo para a linguagem de programação Java. A seguir são detalhados os artefatos de entrada e saída para cada atividade.

Os artefatos da atividade *Selecionar conceitos* foram assim definidos:

- **Linguagem alvo:** Java.
- **Requisitos de busca:** Deve ser possível buscar informações acerca de: pacotes, classes, métodos, e outros elementos.
- **Conceitos:** Pacote, Classe Principal[1], demais Classes, Método Principal, demais Métodos e Outros elementos[2].

A seguir são detalhados os artefatos da atividade *Priorizar elementos*:

- **Método de priorização:** AHP.
- **Opiniões de especialistas:** Foram consideradas as opiniões dos desenvolvedores do processo. Para a aplicação do método AHP, considerando o artefato *Conceitos*, essas opiniões ficaram assim representadas:
  - Um *Pacote Java* é:
    * *Fracamente mais importante* que uma Classe Principal;
    * *Moderadamente mais imp.* que as demais Classes;
    * *Fortemente mais imp.* que o Método Principal;
    * *Extremamente mais imp.* que os demais Métodos;
    * *Absolutamente mais imp.* que os Outros Elementos.
  - Uma *Classe Principal* é:
    * *Fracamente mais imp.* que as demais Classes;
    * *Moderadamente mais imp.* que o Método Principal;
    * *Fortemente mais imp.* que os demais Métodos;
    * *Extremamente mais imp.* que os Outros Elementos.
  - Uma *Classe* é:
    * *Fracamente mais imp.* que o Método Principal;
    * *Moderadamente mais imp.* que os demais Métodos;
    * *Moderadamente mais imp.* que os Outros Elementos.
  - Um *Método Principal* é:
    * *Fracamente mais imp.* que os demais Métodos;
    * *Moderadamente mais imp.* que os Outros Elementos.
  - Um *Método* é *Fracamente mais imp.* que os Outros Elementos.
- **Priorização:** Com as informações das *Opiniões de especialistas* e dos *Conceitos* foi possível utilizar o *Método de priorização* AHP para priorizar os elementos.
  A Tabela 3 mostra as comparações entre pares, com seus valores quantitativos respeitando o que foi apresentado na Tabela 1, e mostra, em valores, as relações entre as estruturas do código definidas nas *Opiniões de especialistas*.
  Aplicando os passos do AHP para esta matriz de comparações par a par entre os elementos selecionados, obtemos os pesos para cada elemento do programa.
  Portanto, os pesos para a 6-tupla $Pesos = (Pacote, ClasseP, Classes, MétodoP, Métodos, OutrosE)$, são respectivamente os valores de:
  $V' = [0,4634 \;\; 0,2593 \;\; 0,1409 \;\; 0,0708 \;\; 0,0405 \;\; 0,0248]$
  Ou seja a relação de consistência, calculada conforme descrito na Seção 2.2 é de aproximadamente 2,85%, adequado segundo o AHP. Desta forma, o artefato *Priorização* ficou assim definido:

Ordem	Estrutura	Peso
1º	Pacote	0,4634
2º	Classe Principal	0,2593
3º	Demais Classes	0,1409
4º	Método Principal	0,0708
5º	Demais Métodos	0,0405
6º	Outros Elementos	0,0248

O artefato de saída da atividade *Definir função de classificação*, é detalhado a seguir:

- **Função de classificação:** A *Função de classificação* foi definida levando em conta o número de vezes que o objeto da consulta foi encontrado em um determinado arquivo, multiplicado pelo peso relativo à estrutura de código na qual o objeto da consulta foi encontrado. Desta forma, a função ficou assim definida:

[1] Algumas bibliotecas desconsideram o conceito de classe principal, para essa instanciação foi decidido manter esse conceito.
[2] Aqui são considerados todos os elementos que não se enquadram nas classificações anteriores.

Tabela 3: Matriz de Prioridades

Localização	Pacote	Classe P.	Classes	Método P.	Métodos	Outros E.
Pacote	1	3	5	7	8	9
Classe P.	1/3	1	3	5	7	8
Classes	1/5	1/3	1	3	5	7
Método P.	1/7	1/5	1/3	1	3	5
Métodos	1/8	1/7	1/5	1/3	1	3
Corpo P.	1/9	1/8	1/7	1/5	1/3	1

$f(uc,t) = toP(uc,t) * wP + toCP(uc,t) * wCP + toC(uc,t) * wC + toMP(uc,t) * wMP + to(uc,t) * wM + toOE(uc,t) * wOE$
Onde *toP, toCP, toC, toMP, toM e toOE*, são respectivamente: funções que calculam total de ocorrências do termo **(t)** em uma unidade de compilação **(uc)** em um pacote, em uma classe principal, em outras classes, em um método principal, em outros métodos e em outros elementos. Da mesma forma, *woP, woCP, woC, woMP, woM e woOE*, são respectivamente: peso de pacote, peso de classe principal, peso de outras classes, peso de método principal, peso de outros métodos e peso de outros elementos.
De posse dos valores definidos no artefato *Priorização*, a *Função de classificação* para esta instanciação, pode ser definida como:
$f(uc,t) = toP(uc,t) * 0,4634 + toCP(uc,t) * 0,2593 + toC(uc,t) * 0,1409 + toMP(uc,t) * 0,0708 + toM(uc,t) * 0,0405 + toOE(uc,t) * 0,0248$.

A seguir são detalhados os artefatos da atividade *Avaliar função de classificação*:

- **Programas de teste:** Foi utilizado um repositório com alguns projetos conhecidos para facilitar a avaliação dos resultados das consultas.
- **Consultas de teste:** Foram realizadas duas consultas, a primeira buscando a string **cfc** e a segunda buscando a string **Metric**.

4.1.1 Primeira Consulta

Esta consulta tem como a *Consulta escrita* a string **cfg**. Após ser processada e executada, esta consulta retornou quatro arquivos como relevantes. Esses arquivos compõem o artefato *Documentos retornados*, e são analisados a seguir:

- O arquivo *HelloWord.java* contém a string **cfg** no nome do Pacote, na linha 1;

```
1  package cfg;
2
3  import org.hibernate.Session;
4  import org.hibernate.Transaction;
5
6  public class HelloWorld {
7     public static void main(String[] args) {
8        Session s = HibernateUtil.getSessionFactory().
9        openSession();
10       Transaction tx = s.beginTransaction();
11       s.save(new Metric("wom", "Weigthed Operations on Methods"));
12       s.save(new Metric("dit", "Deep of Inheritance Tree"));
13       s.save(new Metric("noc", "Number of Children"));
14       s.save(new Metric("loc", "Lines of Code"));
15       tx.commit();
16       s.close();
17    }
18 }
```

- O arquivo *DataBaseCreator.java* contém duas ocorrências da string **cfg** sendo as duas no método principal, nas linha 5 e 6;

```
1  package teste.consultas;
2
3  public class DatabaseCreator {
4     public static void main(String[] args) {
5        Configuration cfg = new AnnotationConfiguration().configure();
6        SchemaExport schemaExport = new SchemaExport(cfg);
7        schemaExport.create(false, true);
8     }
9  }
```

- O Arquivo *Activator.java* contém quatro ocorrências da string **cfg**, a primeira em uma declaração de variável na linha 5, a qual é considerada como outros elementos, e as outras três nos demais métodos nas linhas 11, 14 e 18;

```
1  package br.ufsm.aopjungle;
2
3  public class Activator extends AbstractUIcfg {
4     public static final String id = "AOPJungle";
5     private static Activator cfg;
6     public Activator() {
7        System.out.println ("Initializing Activator ...");
8        }
9     public void start(BundleContext context) throws Exception {
10       super.start(context);
11       cfg = this;
12    }
13    public void stop(BundleContext context) throws Exception {
14       cfg = null;
15       super.stop(context);
16    }
17    public static Activator getDefault() {
18       return cfg;
19    }
20 }
```

- O arquivo *WebService.java* contém três ocorrências da string **cfg**, todas sendo comentários e localizadas em outros elementos, na linha 8.

```
1  package comTrabWebservice;
2
3  public class WebService {
4     public String teste(String algo) {
5        return "o retorno" + algo;
6     }
7  }
8  //cfg cfg cfg
```

Ao aplicar a função de classificação os valores obtidos são:

- $f(WebService.java, \text{“cfg”}) = 3 * 0,0248 = 0,0744$
- $f(Activator.java, \text{“cfg”}) = 1 * 0,0248 + 3 * 0,0405 = 0,1463$
- $f(DatabaseCreator.java, \text{“cfg”}) = 2 * 0,0708 = 0,1416$
- $f(HelloWord.java, \text{“cfg”}) = 1 * 0,4634 = 0,4634$

A Tabela 4 mostra a classificação dos arquivos após a atividade *Classificar resultados*, representando o artefato *Resultados classificados*.

Tabela 4: Ordem do Resultado da Consulta por "cfg"

Arquivo	Val. Classificação	Classificação
HelloWord.java	0,4634	1º
Activator.java	0,1463	2º
DatabaseCreator.java	0,1416	3º
WebService.java	0,0744	4º

4.1.2 *Segunda Consulta*

Até a etapa de **Reificação**, a segunda consulta realizada possui os mesmos artefatos da primeira consulta. Apenas na etapa de **Análise**, na atividade *Escrever consulta*, o artefato *Consulta escrita* é representado pela string **Metric**. Como resultado, foram retornados os arquivos: `TokenProcessor.java`, `HelloWorld.java`, `Metric.java`, `Teste.java` e `HibernateUtil.java`.

Assim como realizado na primeira consulta, foi aplicada a função de classificação e feita a comparação entre os resultados.

- O arquivo *TokenProcessor.java* apresenta cinco ocorrências da string **Metric**, todas elas em outros elementos, na linha 9;

```
1 package br.ufsm.ajsearch.inf;
2
3 public interface TokenProcessor {
4    public void setContext(boolean isContext);
5    public boolean isContext();
6    public String translate(Token token, TokenProcessor context);
7    public String getModifierTag();
8 }
9 //Metric Metric Metric Metric Metric
```

- O arquivo *HelloWord.java* possui quatro ocorrências da string **Metric**, todas no método principal, nas linha 11, 12, 13 e 14;

```
1  package cfg;
2
3  import org.hibernate.Session;
4  import org.hibernate.Transaction;
5
6  public class HelloWorld {
7     public static void main(String[] args) {
8        Session s = HibernateUtil.getSessionFactory().
9        openSession();
10       Transaction tx = s.beginTransaction();
11       s.save(new Metric("wom", "Weigthed Operations on Methods"));
12       s.save(new Metric("dit", "Deep of Inheritance Tree"));
13       s.save(new Metric("noc", "Number of Children"));
14       s.save(new Metric("loc", "Lines of Code"));
15       tx.commit();
16       s.close();
17    }
18 }
```

- No arquivo Metric.java foram encontradas quatro ocorrências da string **Metric**, uma no nome da classe principal na linha 4, e três em métodos nas linhas 20, 22 e 26;

```
1  package teste.consultas;
2
3  @Entity
4  public class Metric{
5     @Id
6     private String id;
7     private String description;
8     public String getId() {
9        return id;
10    }
11    public void setId(String id) {
12       this.id = id;
13    }
14    public String getDescription() {
15       return description;
16    }
17    public void setDescription(String description) {
18       this.description = description;
19    }
20    public Metric() {
21    }
22    public Metric(String id) {
23       this();
24       setId(id);
25    }
26    public Metric(String id, String description) {
27       this(id);
28       setDescription(description);
29    }
30 }
```

- O arquivo *Teste.java* apresenta uma ocorrência da string **Metric** no nome do pacote, na linha 1;

```
1 package Metric;
2
3 public class Teste {
4    public int soma(int a, int b){
5       return a+b;
6    }
7 }
```

- O arquivo *HibernateUtil.java* apresenta uma ocorrência da string **Metric** em outros elementos, na linha 11.

```
1  package teste.consultas;
2
3  public class HibernateUtil {
4     public static SessionFactory getSessionFactory() {
5        return sessionFactory;
6     }
7     public static void shutdown() {
8        getSessionFactory().close();
9     }
10 }
11 //Metric
```

Aplicando a função de classificação, os valores obtidos são:

- $f(\textit{TokenProcessor.java}, \textit{"Metric"}) = 5 * 0,0248 = 0,1240$
- $f(\textit{HelloWord.java}, \textit{"Metric"}) = 4 * 0,0708 = 0,2832$
- $f(\textit{Metric.java}, \textit{"Metric"}) = 1 * 0,2593 + 3 * 0,0405 = 0,3808$
- $f(\textit{Teste.java}, \textit{"Metric"}) = 1 * 0,4634 = 0,4634$
- $f(\textit{HibernateUtil.java}, \textit{"Metric"}) = 1 * 0,0248 = 0,0248$

A Tabela 5 mostra a saída classificada, após ser realizada a atividade *Classificar resultados*.

Tabela 5: Ordem do Resultado da Consulta por "Metric" Classificada

Arquivo	Valor de Classificação	Classificação
Teste.java	0,4634	1º
Metric.java	0,3808	2º
HelloWord.java	0,2832	3º
TokenProcessor.java	0,1240	4º
HibernateUtil.java	0,0248	5º

A comparação dos resultados das duas consultas utili-

zadas como exemplo mostra que o processo pode mudar consideravelmente a ordem dos resultados, facilitando as buscas

4.2 Instanciação para AspectJ

Apesar de o processo poder ser aplicado em repositórios com arquivos de mais de uma linguagem de programação, para uma avaliação mais clara, optou-se por realizar instanciações diferentes para cada linguagem. Esta seção apresenta uma instanciação do processo para a linguagem AspectJ, a seguir são detalhados os artefatos para cada atividade.

Para a atividade *Selecionar conceitos*, os artefatos foram assim definidos:

- **Linguagem alvo:** AspectJ.
- **Requisitos de busca:** Deve ser possível encontrar informações acerca de: nome de pacotes, nome de aspectos, adendos, declarações intertipos, pontos de corte e declarações de herança/alerta/erro (*declare parents/warning/error*).
- **Conceitos:** Pacote, Aspecto, Adendo, Declaração Intertipo, Ponto de Corte, Expressão de Junção (*Pointcut Expression*) e Declaração de Herança/Alerta/Erro.

Os artefatos da atividade *Priorizar elementos*, são detalhados a seguir:

- **Método de priorização:** AHP.
- **Opiniões de especialistas:** Foram consideradas as opiniões dos próprios desenvolvedores do processo. Para a aplicação do método AHP, essas opiniões ficaram assim representadas:
  - Um *Pacote AspectJ* é:
    * *Ligeiramente mais importante* que um aspecto;
    * *Fracamente mais imp.* que um adendo;
    * *Moderadamente mais imp.* que uma declaração intertipos (ITD)[3];
    * *Fortemente mais imp.* que um ponto de corte/expressão de junção.
    * *Extremamente mais imp.* que uma declaração de alerta/erro.
  - Um *Aspecto* é:
    * *Ligeiramente mais imp.* que um adendo;
    * *Fracamente mais imp.* que uma ITD;
    * *Moderadamente mais imp.* que um ponto de corte[4].
    * *Fortemente mais imp.* que uma declaração de alerta/erro.
  - Um *Adendo* é:
    * *Ligeiramente mais imp.* que uma ITD;
    * *Fracamente mais imp.* que um ponto de corte/expressão de junção .
    * *Moderadamente mais imp.* que uma declaração de alerta/erro.
  - Uma *Declaração Intertipo* é:
    * *Ligeiramente mais imp.* que um ponto de corte/expressão de junção.
    * *Fracamente mais imp.* que uma declaração de alerta/erro.
  - Um *Ponto de Corte/Expressão de Junção* é *Ligeiramente mais imp.* que uma declaração de alerta/erro.
- **Priorização:** Após serem feitos os passos realizados na Seção 4.1 para a obtenção dos pesos para cada conceito, o artefato *Priorização*, ficou assim definido:

Ordem	Estrutura	Peso
1º	Pacote	0,4107
2º	Aspecto	0,2560
3º	Adendo	0,1541
4º	Declaração Intertipos	0,0902
5º	Ponto de Corte	0,0544
6º	Declaração de Erro/Alerta	0,0347

Com uma relação de consistência (CR) de aproximadamente 1,2%, adequada segundo o AHP.

Como artefato de saída esta atividade produz o artefato *Função de classificação*, detalhado a seguir:

- **Função de classificação:** a função foi definida usando os pesos obtidos com o método AHP:
  $f(p) = toP * wP + toAs * wAs + toAd * wAd + toDIT * wDIT + toPC * wPC + toDEW * wDEW$).
  onde *toP, toAs, toAd, toDIT, toPC e toDEW*, são respectivamente: total de ocorrências em pacote, em aspectos, em adendos, em declarações intertipos, em pontos de corte e em declarações de erro/alerta. Da mesma forma, *woP, woAs, woAd, woDIT, woPC e woDEW*, são respectivamente: peso de pacotes, de aspectos, de adendos, de declarações intertipos, de pontos de corte e de declarações de erro/alerta.
  De posse dos valores definidos no artefato *Priorização*, a *Função de classificação* para esta instanciação, pode ser definida como:
  $f(p) = toP * 0,4107 + toAs * 0,2560 + toAd * 0,1541 + toDIT * 0,0902 + toPC * 0,0544 + toDEW * 0,0347$.

A seguir são detalhados os artefatos da atividade *Avaliar função de classificação*:

- **Programas de teste:** Foi utilizado um repositório com um conjunto de arquivos limitado, para que fosse possível avaliar o processo.
- **Consultas de teste:** Foram realizadas duas consultas a primeira buscando pela string **AspectTetris** e a segunda buscando a string **Menu**.

4.2.1 Primeira Consulta

A seguir, mostramos os arquivos de resultado para a primeira consulta realizada na linguagem AspectJ: `AOJBindingDataset.aj`, `AspectTest.aj`, `TestAspect.aj` e `DesignCheck.aj`. A string **AspectTetris** representa o artefato *Consulta escrita*.

[3]Toda a incidência de uma declaração de herança será considerada, para questões de classificação, com a mesma importância de uma ocorrência de uma declaração intertipos.

[4]Toda a ocorrência de uma expressão de junção, terá o mesmo valor de um ponto de corte. Tomou-se essa decisão porque uma expressão de junção sempre é utilizada para capturar um ponto de corte

Os quatro arquivos que foram retornados como relevantes, compondo o artefato *Documentos retornados*, são detalhados a seguir:

- O arquivo *AOJBindingDataset.aj* possui três incidências da string **AspectTetris**, na linha 9, todas elas em comentários. Comentários não foram definidos como estruturas relevantes, desta forma, o peso de classificação deste arquivo é zero.

```
1  package br.ufsm.aopjungle.bindings;
2
3  public privileged aspect AOJBindingDataset {
4     pointcut callType(AOJTypeble type) : call (* List+.add(..)) &&
            args(type) && within(AOJCompilationUnit);
5     /**
6      * First phase, only saves type on the typeTable. Second pass will
            looking for AOJungle
7      * Correspondence and fill the value
8      * AspectTetris AspectTetris AspectTetris
9      * @param type The visited type
10     */
11    after(AOJTypeble type) : callType(type) {
12       getProject(type).getBindingMapping().put(type.getNode(), type);
13    }
14    private AOJProject getProject(AOJTypeble type) {
15       AOJCompilationUnit cUnit = (AOJCompilationUnit)type.getOwner();
16       AOJPackageDeclaration pack = (AOJPackageDeclaration)cUnit.
               getOwner();
17       AOJProject project = (AOJProject)pack.getOwner();
18       return project;
19    }
20 }
```

- O arquivo *AspectTest.aj*, contém uma ocorrência da string **AspectTetris** em uma declaração de herança, na linha 4. Ficando assim o peso de classificação deste arquivo:
$f(AspectTest.aj) = 1 * 0,0902 = 0,0902$.

```
1  package com.ajtetris.core;
2
3  public aspect AspectTest {
4     declare parents : ParentTest extends AspectTetris;
5  }
```

- O arquivo *TestAspect.aj*, contém uma ocorrência da *string* **AspectTetris** em um adendo, na linha 10. Ficando assim o peso de classificação deste arquivo:
$f(TestAspect.aj) = 1 * 0,1541) = 0,1541$.

```
1  /*
2   * Copyright 2003 Gustav Evertsson All Rights Reserved.
3   */
4  package com.ajtetris.aspects;
5
6  public aspect TestAspect {
7     pointcut logPoint(String fileName) : call(* com.ajtetris.logic.
            TetrisImages.loadImage(String)) && args(fileName);
8     before(String fileName) : logPoint(fileName) {
9        System.out.println(thisJoinPoint.getSignature() + ", " +
               fileName + ", AspectTetris");
10    }
11 }
```

- O arquivo *DesignCheck.aj*,contém uma ocorrência da string **AspectTetris** em uma declaração de alerta, na linha 7. Sendo esta a classificação deste arquivo:
$f(DesignCheck.aj) = 1 * 0,347 = 0,0347$.

```
1  package com.ajtetris.aspects.developemnt;
2
3  public aspect DesignCheck {
4     declare warning: call(com.ajtetris.gui.BlockPanel.new(..)) && !
            within(Gui.*) && !within(Aspects..*): "Do not create
            BlockPanel outside the Gui package!";
5     declare warning: call(* com.ajtetris.core.AspectTetris.*(..)) && !
            within(com.ajtetris.core.AspectTetris) && !within(Aspects
            ..*): "Do not call AspectTetris outside the class, use the
            IEventListner interface!";
6  }
```

A Tabela 6 mostra a ordem de visualização dos resultados da consulta com a realização do processo.

Tabela 6: Ordem do Resultado da Consulta por "AspectTetris"Classificada

Arquivo	Peso	Classificação
TestAspect.aj	0,1541	1º
AspectTest.aj	0,0902	2º
DesignCheck.aj	0,0347	3º
AOJBindingDataset.aj	0	4º

4.2.2 *Segunda Consulta*

A seguir, é mostrado o resultado da segunda consulta realizada na linguagem AspectJ. O artefato *Consulta escrita* é a string **Menu**, e compõem o artefato *Documentos retornados*, os seguintes arquivos: `PolicyEnforcements.aj`, `ExceptionHandler.aj`, `NextBlock.aj` e `Menu.aj`.

- O arquivo *PoliceEnforcements*.aj possui duas ocorrências da *string* **Menu**, todas em comentários, na linha 4. Comentários não foram definidos como estruturas relevantes, desta forma, o peso de classificação deste arquivo é zero.

```
1  package br.ufsm.aopjungle.util;
2
3  /* This review is here only to test
4  * Menu Menu
5  */
6  import java.util.List;
7  import br.ufsm.aopjungle.metamodel.commons.AOJCompilationUnit;
8  import br.ufsm.aopjungle.metamodel.commons.AOJTypeble;
9
10 public aspect PoliceEnforcements {
11    pointcut setTypeNotWithinCompilationUnits() :
12       call (* List+.add(AOJTypeble)) && !within(AOJCompilationUnit);
13    declare error : setTypeNotWithinCompilationUnits() :
14       "Use addType method of AOJCompilationUnit class instead";
15          pointcut callType(AOJTypeble type) :
16       call (* List+.add(..)) && args(type) && within(
               AOJCompilationUnit);
17 }
```

- O arquivo *ExceptionHandler.aj*, contém uma ocorrência da string **Menu** dentro de um adendo, na linha 7. Ficando assim o peso de classificação deste arquivo:
$f(ExceptionHandler.aj) = 1 * 0,1541 = 0,1541$.

```
1  package br.ufsm.aopjungle.exception;
2
3  public aspect ExceptionHandler {
4     declare soft : InvalidStackObjectException : execution (* *.*(
            ASTNode));
5     pointcut visitStackException() : execution (* br.ufsm.aopjungle.
            AOJungleVisitor.*(..));
6     before() : visitStackException() {
7        System.out.println("Menu");
8     }
9
10    void around (ASTNode node) : execution (* *.visit(ASTNode)) &&
            args (node) {
11       try {
12          proceed(node);
```

```
13      } catch (InvalidStackObjectException e) {
14         AOJLogger.getLogger().info("Stack Object type is not the
                expected", e);
15      }
16   }
17 }
```

- O arquivo *NextBlock.aj*, duas incidências da string **Menu** em um adendo, nas linhas 28 e 29. Ficando assim o peso de classificação deste arquivo:
$f(NextBlock.aj) = 2 * 0,1541 = 0,3082$.

```
1  package com.ajtetris.aspects.logic;
2
3  import java.util.Random;
4  import com.ajtetris.aspects.gui.*;
5  import com.ajtetris.gui.*;
6  import com.ajtetris.logic.*;
7
8  public aspect NextBlock {
9     pointcut guiInit() : execution(com.ajtetris.gui.TetrisGUI.new(..))
            ;
10    pointcut getNextBlock() : call(* com.ajtetris.core.AspectTetris.
            getRandomBlock());
11    protected BlockPanel nextBlockPanel;
12    protected int nextBlock;
13    after() : guiInit() {
14       Random rn = new Random();
15       nextBlock = rn.nextInt(Blocks.NUMBEROFTYPES);
16       nextBlockPanel = new BlockPanel(4, 4, "");
17       if(GameInfo.infoPanel != null)
18          GameInfo.infoPanel.add(nextBlockPanel);
19          nextBlockPanel.setBlocks(Blocks.getBlock(nextBlock));
20    }
21    int[][] around() : getNextBlock() {
22       int currentBlock = nextBlock;
23       Random Menu = new Random();
24       nextBlock = Menu.nextInt(Blocks.NUMBEROFTYPES);
25       nextBlockPanel.setBlocks(Blocks.getBlock(nextBlock));
26       return Blocks.getBlock(currentBlock);
27    }
28 }
```

- O arquivo *Menu.aj*, contém uma ocorrência da string **Menu** no nome de um aspecto, na linha 9. Sendo esta a classificação deste arquivo:
$f(Menu.aj) = 1 * 0,2560 = 0,2560$.

```
1  package com.ajtetris.aspects.gui;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import com.ajtetris.gui.*;
7  import com.ajtetris.eventinterface.*;
8
9  public aspect Menu implements ActionListener {
10    protected JMenuItem newGameMI;
11    protected JMenuItem pauseMI;
12    protected JMenuItem exitMI;
13    protected IEventListner tetris;
14    pointcut guiInit() : execution(com.ajtetris.gui.TetrisGUI.new(..))
            ;
15    pointcut tetrisInit(IEventListner tetris) : execution(com.ajtetris
            .core.AspectTetris.new(..)) && target(tetris);
16    after() : guiInit() {
17      // code omitted...
18    }
19 }
```

A Tabela 7 mostra o resultado após a aplicação do processo.

Tabela 7: Resultado da Consulta por "Menu"

Arquivo	Peso	Classificação
NextBlock.aj	0,3082	1º
Menu.aj	0,2560	2º
ExceptionHandler.aj	0,1541	3º
PoliceEnforcements.aj	0	4º

5 Trabalhos Relacionados

Consultas em código de programas constituem uma das técnicas para as atividades de reengenharia de software (Kullbach e Winter, 1999). Seja na atividade de compreensão de código ou durante as atividades de refatoração, as linguagens de consulta em código, como JTL (*Java Tool Language*) (Cohen et al., 2006) e CodeQuest (Hajiyev et al., 2006), fornecem recursos para que desenvolvedores recuperem informações, coletem métricas e naveguem pelas estruturas de um programa.

Mecanismos de consulta em código fonte têm sido extensivamente estudados, principalmente no contexto de orientação a objetos, para qual ferramentas como as apresentadas por Urma e Mycroft (2012, 2015), ou a apresentada por Bajracharya et al. (2014), assim como linguagens têm sido propostas (Janzen e De Volder, 2003; McCormick e De Volder, 2004; Cohen et al., 2006; Hajiyev et al., 2006). Essas ferramentas e linguagens também são relevantes no estudo de busca em outros paradigmas, pois normalmente as linguagens de novos paradigmas são extensões de linguagens existentes, como AspectJ, que é uma extensão de Java. Independentemente disso, existem linguagens propostas especificamente para um paradigma específico, tal como o paradigma orientado a aspectos (de Faveri, 2013).

Consultas não estruturadas, ou seja, sem um formato, regra ou sequência padronizados, dependem dos critérios de classificação dos algoritmos usados pelos motores de busca aos quais elas são submetidas. O usuário final não pode mudar a maneira como um algoritmo classifica as palavras ou os arquivos, mas ele pode interagir com o sistema formulando ou reformulando uma consulta (Croft et al., 2010). Essa interação é uma parte crucial para o processo de recuperação de informação, e pode determinar se o motor de busca está realizando um serviço eficaz.

As abordagens de consulta baseadas em palavras-chave exigem que os usuários descrevam suas necessidades, e em seguida, combinem essa consulta textual ao texto contido no código fonte. Assim, o sucesso da pesquisa geralmente depende da capacidade dos usuários em selecionar palavras que podem ter sido utilizadas, ou que sejam semelhantes às utilizadas no código (Stolee et al., 2016). Esse processo pode exigir que as consultas sejam remodeladas várias vezes.

Consultas não estruturadas tem sido exploradas em ferramentas que indexam código em diferentes níveis e diferentes linguagens a partir de repositórios contendo uma grande quantidade de arquivos. Diferentemente da abordagem proposta neste trabalho, a grande maioria das ferramentas de consulta não estruturada para código fonte não apresenta critérios de classificação que priorizem documentos de maior relevância ao usuário, tais como (*Nerdy-Data*, 2024), (*Krugle*, 2024) e (Basu, 2024).

Em síntese, embora linguagens de consulta estruturada, como JTL e CodeQuest, ofereçam mecanismos expressivos para recuperação de informações e identificação de oportunidades de refatoração, elas exigem conhecimento prévio

de sintaxe específica e dependem da estrutura da linguagem alvo. Por outro lado, ferramentas de busca não estruturada, como NerdyData, Krugle e OpenHub, permitem consultas livres, mas não fornecem critérios de priorização sensíveis ao contexto do usuário ou às características da linguagem.

Diferentemente dessas abordagens, o processo proposto combina a flexibilidade das consultas não estruturadas com um mecanismo de classificação baseado na importância relativa dos elementos da linguagem, permitindo ordenar resultados conforme preferências definidas pelos desenvolvedores e aumentando a precisão da recuperação de informação.

6 Conclusões

Este artigo apresentou um processo para consultas em código fonte. O processo possibilita aos usuários a definição de quais fragmentos de código fonte serão considerados importantes no momento da realização das consultas. Esta definição permite uma melhora na classificação dos resultados, ocasionando uma diminuição no tempo gasto com a análise desses resultados. Por utilizar técnicas de consultas não estruturadas, este processo não impõe aos usuários a necessidade de aprender uma nova sintaxe para obter resultados com uma maior exatidão nas consultas executadas. Para a validação do processo foram realizadas duas instanciações. A primeira na linguagem orientada a objetos Java e a segunda na linguagem orientada a aspectos AspectJ, demonstrando a aplicabilidade do processo. Como trabalho futuro, estuda-se a aplicação de outros métodos de decisão multi-critério para a definição dos pesos, incluindo abordagens automatizadas de acordo com repositórios de larga escala da linguagem alvo.

Referências

Bajracharya, S., Ossher, J. e Lopes, C. (2014). Sourcerer: An infrastructure for large-scale collection and analysis of open-source code, *Science of Computer Programming* **79**: 241–259. https://doi.org/10.1016/j.scico.2012.04.008.

Basu, S. (2024). Ohloh. Acessado em Novembro/2024, http://code.openhub.net/.

Bhushan, N. e Rai, K. (2004). *Strategic decision making: applying the analytic hierarchy process*, Springer Science & Business Media. https://doi.org/10.1007/b97668.

Cohen, T., Gil, J. e Maman, I. (2006). JTL: the Java tools language, *ACM SIGPLAN Notices* **41**(10): 89–108. https://dl.acm.org/doi/10.1145/1167515.1167481.

Croft, W. B., Metzler, D. e Strohman, T. (2010). *Search engines: Information retrieval in practice*, Vol. 520, Addison-Wesley Reading. https://dl.acm.org/doi/book/10.5555/1516224.

de Faveri, C. (2013). Uma linguagem específica de domínio para busca em código orientado a aspectos, *Mestrado / UFSM* . https://repositorio.ufsm.br/handle/1/5435.

Fokaefs, M., Tsantalis, N. e Chatzigeorgiou, A. (2007). Jdeodorant: Identification and removal of feature envy bad smells, *2007 IEEE ICSM*, IEEE, pp. 519–520. https://doi.org/10.1109/ICSM.2007.4362679.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*, Addison-Wesley, USA. https://dl.acm.org/doi/book/10.5555/311424.

Godfrey, M. W. e German, D. M. (2008). The past, present, and future of software evolution, *2008 Frontiers of Software Maintenance*, IEEE, pp. 129–138. https://doi.org/10.1109/FOSM.2008.4659256.

Hajiyev, E., Verbaere, M. e De Moor, O. (2006). Codequest: Scalable source code queries with datalog, *ECOOP 2006, Nantes, France, July 3-7, 2006.*, Springer, pp. 2–27. https://dl.acm.org/doi/10.1007/11785477_2.

Hecht, M., Piveta, E., Pimenta, M. e Price, R. T. (2006). Aspect-oriented code generation, *Anais do SBES'06*, SBC, pp. 209–223.

ISO (2022). ISO/IEC/IEEE 14764:2022 - Software engineering — Software life cycle processes — Maintenance , *Standard*, International Organization for Standardization, Geneva, CH.

Janzen, D. e De Volder, K. (2003). Navigating and querying code without getting lost, *Proc of AOSD'03*, pp. 178–187. https://dl.acm.org/doi/10.1145/643603.643622.

Júnior, J. E. T., Neto, H. E. V. T., Faveri, C. D., de Brum Saccol, D., Vizzotto, J. K. e Piveta, E. K. (2019). A refactoring catalog for lambda expressions in Java, *Int. J. Softw. Eng. Knowl. Eng.* **29**(6): 791–818. https://doi.org/10.1142/S021819401950027X.

Kerievsky, J. (2004). *Refactoring to patterns*, Pearson. https://dl.acm.org/doi/10.5555/993772.

Krugle (2024). Acessado em Novembro/2024, http://www.krugle.com/.

Kullbach, B. e Winter, A. (1999). Querying as an enabling technology in software reengineering, *Proc. of CSMR'99*, IEEE, pp. 42–50. https://dl.acm.org/doi/10.5555/794202.795241.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E. e Turski, W. M. (1997). Metrics and laws of software evolution-the nineties view, *Proc of Metrics'97*, IEEE, pp. 20–32. https://dl.acm.org/doi/10.5555/823454.823901.

McCormick, E. e De Volder, K. (2004). JQuery: finding your way through tangled code, *Companion to OOPSLA'04*, pp. 9–10. https://dl.acm.org/doi/abs/10.1145/1028664.1028670.

Mens, T. e Tourwé, T. (2004). A survey of software refactoring, *IEEE Transactions on software engineering* **30**(2): 126–139. https://dl.acm.org/doi/10.1109/TSE.2004.1265817.

NerdyData (2024). Acessado em Novembro/2024, http://nerdydata.com/.

Piveta, E. K. (2009). Improving the search for refactoring opportunities on object-oriented and aspect-oriented software (PhD Thesis/UFRGS). https://lume.ufrgs.br/handle/10183/15651.

Piveta, E., Pimenta, M., Araújo, J., Moreira, A., Guerreiro, P. e Price, R. T. (2009). Representing refactoring opportunities, *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 1867–1872. https://dlnext.acm.org/doi/abs/10.1145/1529282.1529701.

Pizka, M. e Jurgens, E. (2007). Automating language evolution, *IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, IEEE, pp. 305–315. https://dl.acm.org/doi/abs/10.1109/TASE.2007.13.

Pressman, R. S. e Maxim, B. R. (2021). *Engenharia de software. Uma abordagem profissional.*, McGraw Hill Brasil.

Robillard, M. P. e Murphy, G. C. (2007). Representing concerns in source code, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **16**(1): 3–es. https://dl.acm.org/doi/abs/10.1145/1189748.1189751.

Saaty, T. L. (1990). How to make a decision: the analytic hierarchy process, *European Journal of Operational Research* **48**(1): 9–26. https://doi.org/10.1016/0377-2217(90)90057-I.

Stolee, K. T., Elbaum, S. e Dwyer, M. B. (2016). Code search with input/output queries: Generalizing, ranking, and assessment, *Journal of Systems and Software* **116**: 35–48. https://doi.org/10.1016/j.jss.2015.04.081.

Urma, R.-G. e Mycroft, A. (2012). Programming language evolution via source code query languages, *Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '12, ACM, New York, NY, USA, p. 35–38. https://doi.org/10.1145/2414721.2414728.

Urma, R.-G. e Mycroft, A. (2015). Source-code queries with graph databases—with application to programming language usage and evolution, *Science of Computer Programming* **97**: 127–134. https://doi.org/10.1016/j.scico.2013.11.010.

Vargas, R. V. e IPMA-B, P. (2010). Using the Analytic Hierarchy Process (AHP) to select and prioritize projects in a portfolio, *PMI Global Congress*, Vol. 32, PA: PMI Washington, DC, pp. 1–22.