Aplicação da Técnica Análise de Acessibilidade para Detecção de Métodos Mortos em Software Orientado a Objetos

Camila Bastos, Paulo Afonso Júnior, Heitor Costa

Departamento de Ciência da Computação - Universidade Federal de Lavras - MG - Brazil

camilabastos@posgrad.ufla.br, {pauloa.junior, heitor}@dcc.ufla.br

Resumo. *As modificações realizadas ao longo do ciclo de vida do software contribuem com o aumento da complexidade e da poluição do código. A presença de código morto favorece essa poluição, dificultando a rastreabilidade de requisitos, a execução de testes e a legibilidade e a compreensão do código fonte. A detecção e a eliminação de código morto são procedimentos árduos e dependem da análise e da compreensão do código fonte. Desse modo, a técnica análise de acessibilidade tem sido utilizada para verificar chamadas de métodos desencadeadas a partir de métodos que iniciam o software, de forma a identificar métodos inacessíveis (código morto). Uma limitação dessa verificação é a dependência de conhecimento prévio do software para determinar seus métodos de inicialização. Assim, neste trabalho, é proposta uma aplicação da técnica análise de acessibilidade que elimina a necessidade de conhecimento prévio e detecta métodos inacessíveis em sistemas de software orientados a objetos. Resultados experimentais indicaram que a abordagem proposta detectou maior quantidade de métodos inacessíveis do que uma ferramenta consolidada no mercado.*

Abstract*. Modifications performed during the software life cycle contribute for increasing code complexity and pollution. The presence of dead code favors such pollution making it difficult to trace requirements, to run tests, and to read and to understand source code. To detect and to remove dead code are arduous processes and depend on analyzing and understanding of source code. Thus, the reachability analysis technique has been applied to verify the method calls triggered from software initialization methods to identify unreachable methods (dead code). A limitation of this verification is the reliance on prior knowledge of software to determine your initialization methods. This work proposes an application of reachability analysis technique, which eliminates the need for prior knowledge and detects unreachable methods in object-oriented software. Experimental results indicate that this proposal can detect larger amount of unreachable methods in relation to a consolidated tool.*

1. Introdução

Sistemas de software sofrem constantes modificações ao longo do seu ciclo de vida para corrigir defeitos, para adicionar funções ou para atualizar tecnologias. De acordo com a Lei da Mudança da Contínua, essas modificações são necessárias para o software evoluir e não se tornar insatisfatório [Lehman; Belady, 1985]. Apesar de necessária, a evolução tem sido apontada como aspecto crítico em assegurar a manutenibilidade do software, por causa do aumento da complexidade que ocorre ao longo das versões. Essa

complexidade contribui com a degradação da qualidade interna, causando envelhecimento do software e aumentando a poluição do código [Pérez *et al*., 2011; Bakota *et al*., 2014].

Um dos fatores que contribui com a poluição do código e com o aumento da complexidade é a existência de código morto. De modo geral, código morto pode ser definido como trechos de código cuja existência não afeta a execução do software. A presença de código morto afeta a rastreabilidade dos requisitos, dificulta a execução de testes, aumenta desnecessariamente o tamanho do software, afeta a legibilidade e prejudica a compreensão [Guerrouat; Richter, 2006; Martins *et al*., 2010]. Esses fatores colaboram para a manutenção ser considerada a etapa mais cara do ciclo de vida do software, visto que aproximadamente metade do tempo utilizado nessa etapa é destinada à compreensão [Scanniello, 2014]. A presença de código morto agrava-se em sistemas de software orientados a objetos, nos quais desenvolvedores enfatizam propriedades e comportamento dos objetos, em vez de detalhes de implementação [Srivastava, 1992].

A eliminação de código morto reduz a complexidade e a poluição do software. No entanto, a detecção desse código é considerada uma atividade árdua de ser realizada por depender da análise e da compreensão do código, visto que é o principal artefato disponível para essa análise [Boomsma; Gross, 2012]. Além disso, a eliminação incorreta desse código pode trazer efeitos colaterais por causa das dependências desconhecidas com os recursos implementados [Scanniello, 2014]. Desse modo, técnicas baseadas em análises estáticas e em análises dinâmicas em software foram propostas para auxiliar na identificação de código morto [Bastos *et al*., 2016]. Técnicas que utilizam análise dinâmica podem ser mais precisas em seus resultados, contudo dependem da execução de todos os cenários possíveis de utilização do software, fazendo da análise estática mais flexível [Romano *et al*., 2016].

Nesse contexto, a técnica análise de acessibilidade utiliza análise estática para detectar trechos de código inacessíveis e não executados durante o funcionamento do software. Trechos de código inacessíveis não podem ser acessados a partir de um ponto de iniciação do software. Diferentes abordagens aplicaram a técnica análise de acessibilidade para identificar métodos inacessíveis (métodos mortos) [Chen *et al*., 1998; Bacon *et al*., 2003; Romano *et al*. 2016]. De modo geral, essas abordagens executam três passos para identificar: i) métodos de iniciação do software; ii) métodos acessíveis pelos métodos de iniciação; e iii) métodos inacessíveis, obtidos pela diferença entre todos os métodos do software e os métodos acessíveis.

Uma limitação das abordagens existentes é a necessidade de conhecimento prévio do software, que depende da análise antecipada de todo o código (se for automatizado) ou da familiaridade do mantenedor (se for manual) para os métodos de iniciação do software serem identificados antes de iniciar a detecção de métodos inacessíveis. Essa familiaridade não é trivial, visto que parte das pessoas envolvidas no processo de manutenção não possuem conhecimento do software [Tjortjis; Layzell, 2001]. Com base nesses fatores, neste trabalho, foi proposta uma aplicação da técnica análise de acessibilidade, que identifica métodos inacessíveis de forma objetiva e elimina a necessidade de conhecimento prévio do software para ser executada.

O restante do artigo está organizado da seguinte forma. Referencial teórico sobre código morto e formas de detecção está resumido na Seção 2. A abordagem proposta

para aplicação da técnica análise de acessibilidade é apresentada na Seção 3. Resultados obtidos com a avaliação da abordagem em relação a uma ferramenta tradicional e amplamente utilizada no mercado para detecção de código morto são discutidos na Seção 4. Alguns trabalhos relacionados são descritos na Seção 5. Considerações finais e sugestões de trabalhos futuros são apresentados na Seção 6.

2. Código Morto

Nesta seção, são apresentados conceitos básicos relacionados a código morto e a técnicas para sua detecção.

2.1. Conceitos Básicos

O termo código morto está relacionado com trechos de código desnecessários/inoperantes, que podem ser eliminados sem alterar a funcionalidade do software [Eder *et al*., 2012]. Grande parte dos trabalhos relacionados a Engenharia de Software consideram código morto como trechos de código inacessíveis e que não podem ser executados, por causa da impossibilidade de serem invocados durante o funcionamento do software. Em outras áreas, como linguagem de programação, código morto é considerado código desnecessário, ou seja, são executados e produzem resultados que não interferem no funcionamento do software [Sunitha; Kumar, 2006]. Existe ainda a relação de código morto com trechos de código não alcançáveis por qualquer fluxo de execução, como trechos de código localizados após instruções de desvio incondicional ou após instruções de retorno [Martins *et al*., 2010]. Alguns tipos de código morto são:

- **Variáveis Mortas**. Variáveis declaradas e iniciadas, mas não manipuladas em outras partes do código. Outro procedimento que torna uma variável morta é a propagação de cópias, em que dada uma atribuição `x ← y`, o uso de `y` será posteriormente substituído pelo uso de `x`, sem que instruções intermediárias alterem seu valor, fazendo com que ambas as variáveis sejam iguais e tornando `y` inútil [Sunitha; Kumar, 2006];
- **Parâmetros Mortos**. Passagem de parâmetros não utilizados no corpo do método. Esse tipo de código dificulta a compreensão da funcionalidade desse método, principalmente se pertencer às bibliotecas/APIs públicas, as quais os desenvolvedores possuem pouca familiaridade com o código [Fehnker; Huuck, 2013];
- **Valor de Retorno Morto**. O valor retornado por um método não é utilizado pelo seu chamador. Esse fato pode ocorrer por causa de chamadas realizadas de forma incorreta, que ignoram a funcionalidade do método;
- **Métodos Mortos**. Métodos mortos não acessíveis (não possuem chamadas) a partir de outros métodos acessíveis. Por causa de sua inacessibilidade, esses métodos não são executados durante o funcionamento do software.

Dentre os diferentes tipos de código morto, os métodos mortos podem ser considerados os mais prejudiciais para manutenibilidade do software. Isso ocorre pelo fato de possuírem maior quantidade de linhas de código em relação aos outros tipos, contribuindo de forma mais significativa com o aumento desnecessário do tamanho do código, da poluição e da complexidade. Além disso, os métodos mortos não estão vinculados a requisitos do software, prejudicando sua rastreabilidade. Outro problema relacionado aos métodos mortos é a existência de código não testado, podendo

ocasionar falhas e anomalias durante o funcionamento do software, caso esses métodos tornem-se acessíveis e ativos novamente [Martin, 2008].

As constantes modificações realizadas no software são as principais causas do surgimento de métodos mortos. Essas modificações podem ocorrer para substituição de métodos obsoletos, atualização de tecnologias, atualização de funções, troca de requisitos e desativação de funções, fazendo com que métodos sejam adicionados/substituídos frequentemente e contribuindo para aumentar a quantidade de código morto sem a percepção dos desenvolvedores [Gold; Mohan, 2003]. Esses fatores estão relacionados com o aumento da complexidade ao longo da evolução do software, conforme abordado na Lei da Complexidade Crescente de Lehman [Lehman; Belady, 1985].

2.2. Detecção de Código Morto

A detecção de código morto pode ser realizada utilizando análise estática ou análise dinâmica do código. A precisão dos resultados obtidos com a análise dinâmica pode superar a precisão dos resultados da análise estática [Romano *et al*., 2016]. No entanto, a detecção de código morto utilizando análise dinâmica depende da execução de inúmeros cenários de uso, para que todas as chamadas possíveis aos métodos do software sejam realizadas. Dessa forma, a detecção de código morto com análise estática pode ser considerada mais flexível por não depender da execução do software.

Técnicas de detecção de código morto foram propostas considerando análise estática e análise dinâmica do software. A técnica Análise de Fluxo de Dados analisa dinamicamente o fluxo de execução para identificar código morto. Essa técnica é indicada para identificar instruções que produzem resultados não utilizados durante o funcionamento do software, além de instruções que não podem ser alcançadas a partir de qualquer fluxo de execução. Em contrapartida, na técnica Análise de Acessibilidade, o objetivo é identificar trechos de código não acessíveis a partir de métodos *root* (métodos de iniciação) do software. Essa técnica utiliza análise estática e é indicada para detectar métodos mortos não invocados durante a execução do software por não possuírem chamadas (diretas ou indiretas) de métodos acessíveis [Bastos *et al*., 2016].

3. Aplicação da Análise de Acessibilidade para Detecção de Métodos Mortos

Neste trabalho, foram considerados dois tipos de métodos mortos: i) **Métodos Inacessíveis**, métodos não invocados (chamados) por algum método; e ii) **Métodos Acessíveis Apenas por Métodos Inacessíveis**, métodos invocados (chamados) por métodos, mas todos os métodos invocadores (chamadores) são métodos inacessíveis. Considerando essas definições, a abordagem proposta para detecção de métodos mortos consiste em executar a técnica análise de acessibilidade em duas fases: i) identificar métodos inacessíveis; e ii) identificar métodos acessíveis apenas pelos métodos inacessíveis identificados na primeira fase.

3.1. Análise de Acessibilidade (Fase I)

Na primeira fase da análise de acessibilidade, o objetivo é identificar métodos inacessíveis, que não podem ser executados durante o funcionamento do software. No ***Algoritmo 1***, é apresentada uma ideia geral do funcionamento dessa análise. Nesse

algoritmo, informações dos métodos do software e das chamadas existentes entre eles, que devem ser abstraídas em uma estrutura de dados, são fornecidas como entrada. Representações baseadas em grafos ou em árvores são estruturas de dados frequentemente utilizadas em abordagens que dependem da análise do código de sistemas de software orientados a objetos, podendo ser utilizadas para realizar essa análise de acessibilidade. Estruturas de dados contendo os métodos acessíveis, os métodos inacessíveis, os métodos chamadores (para os métodos acessíveis) e os métodos chamados (se existirem) são o resultado desse algoritmo.

Algoritmo 1: Análise de Acessibilidade (Fase I)

```
        Entrada:    representação (Estrutura de Dados que Representa o Software)
        Resultado: métodosVivos (Estrutura de Dados com Métodos Acessíveis)
                   métodosMortos (Estrutura de Dados com Métodos Inacessíveis)
 1  Início
 2    declare métodosVivos, métodosMortos                    // Figura 2A
 3    para cada (pacote ∈ representação) faça
 4      para cada (classe ∈ pacote) faça
 5        para cada (metodo ∈ classe) faça
 6          se (metodo ≠ main | construtor | listener | método de interface) então
 7            métodosChamadores ← obterChamadoresDoMétodo(método)
 8            métodosChamados ← obterMétodosChamados(método)
 9            se (métodosChamadores é vazio) então      // o método é inacessível
10              adiciona método, métodosChamados em métodosMortos
11            senão                                     // o método é acessível
12              adiciona método, métodosChamadores, métodosChamados em
                    métodosVivos
13            fim se
14          fim se
15        fim para
16      fim para
17    fim para
18  Fim
```

Esse algoritmo foi implementado em um *plug-in* para IDE Eclipse, no qual o software foi abstraído em uma estrutura de dados gerada pelo JDT (*Java Development Tools*) (www.eclipse.org/jdt/). O JDT representa o código em uma estrutura de árvore utilizando uma AST (*Abstract Syntax Tree*) ou um *Java Model*. A AST é uma estrutura mais robusta em relação ao *Java Model* por conter detalhes que permitem inserir, alterar ou apagar trechos de código. Por causa da sua robustez, a AST é computacionalmente mais custosa de ser gerada do que o *Java Model*. Na utilização da análise de acessibilidade proposta, são necessárias informações sobre quais métodos estão presentes no software, suas assinaturas e as chamadas existentes entre eles. Informações dessa granularidade são obtidas pelo *Java Model*, fazendo desnecessário a utilização da AST. Desse modo, foi considerada uma árvore do *Java Model* como entrada do algoritmo, cuja estrutura hierárquica é ilustrada de forma simplificada na Figura 1. O ***Algoritmo 1*** percorre essa árvore e utiliza a hierarquia de chamadas do JDT para obter a lista de métodos chamadores (linha 7) e de métodos chamados (linha 8) para cada método analisado, sendo definidos como métodos inacessíveis aqueles que não possuírem chamadores.

A condição na linha 6 desconsidera alguns métodos da análise de acessibilidade. Métodos `main` foram desconsiderados por serem responsáveis pela iniciação do software. Construtores e métodos abstratos de interface foram desconsiderados por não fazerem parte do escopo da proposta (analisar a acessibilidade de métodos concretos). Métodos relacionados aos eventos (`listeners`) foram desconsiderados por causa da impossibilidade de detectar suas chamadas utilizando análise estática [Romano *et al*., 2016]. O formato dos resultados gerados na primeira fase da análise de acessibilidade pode ser visualizado na Figura 2. A estrutura `métodosMortos` armazena uma coleção de métodos inacessíveis, juntamente com uma lista de métodos chamados por eles (Figura 2A). A estrutura `métodosVivos` armazena uma coleção de métodos acessíveis, juntamente com uma lista de métodos chamados por eles e uma lista de métodos que os chamam (Figura 2B). Essas informações são necessárias para executar a segunda fase da análise de acessibilidade, evitando que novos acessos sejam efetuados na estrutura do *Java Model*.

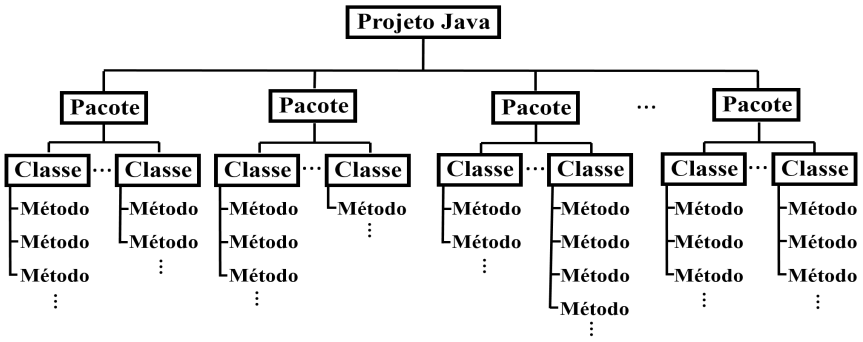

Figura 1. Estrutura de Entrada do *Algoritmo 1*

Figura 2. Estrutura de Saída do *Algoritmo 1*

3.2. Análise de Acessibilidade (Fase II)

Na segunda fase da análise de acessibilidade, são encontrados métodos acessíveis apenas por métodos inacessíveis, não identificados pelo ***Algoritmo 1*** por possuírem uma lista de chamadores. A proposta dessa fase é apresentada no ***Algoritmo 2***. As estruturas de dados `métodosMortos` e `métodosVivos` geradas na fase anterior são fornecidas como entrada a esse algoritmo. Basicamente, nesse algoritmo, uma fila é criada com os métodos inacessíveis (linha 2) e a busca em largura utilizada para percorrer as chamadas desencadeadas a partir dos métodos presentes na fila. Os métodos cujo conjunto de chamadores esteja contido no conjunto da estrutura `métodosMortos` (satisfazendo a condição na linha 9) também são considerados inacessíveis e inseridos no final da fila, permitindo que os métodos chamados por eles sejam analisados. Essa busca é finalizada quando todos os métodos inacessíveis forem identificados.

Algoritmo 2: Análise de Acessibilidade (Fase II)

Entrada: *métodosMortos* (Figura 2A)
métodosVivos (Figura 2B)
Resultado: *estrutura*: *métodosMortos* (Figura 2A) com todos os métodos mortos identificados no software

```
1  Início
2     fila ← criarFila(métodosMortos)
3     enquanto (fila não vazia) faça
4        raiz ← removerDaFila(fila)            // remove o primeiro método da fila
5        métodosChamados ← métodos chamados pela raiz em métodosMortos
6        para cada (metodo ϵ métodosChamados) faça
7           se (metodo ∉ métodosMortos) então
8              chamadores ← métodos que chamam método em métodosVivos
9              se (chamadores ⊂ métodosMortos) então
10                adiciona método em métodosMortos
11                remove método de métodosVivos
12                adicionaNaFila(método)
13             fim se
14          fim se
15       fim para
16    fim enquanto
17 Fim
```

Na Figura 3, é apresentada uma ilustração de alguns métodos (vértices) de um software e as chamadas existentes entre eles (arestas direcionadas). Considerando esse exemplo, após a execução do ***Algoritmo 1***, a estrutura `métodosMortos` conterá os métodos inacessíveis `A()`, `B()` e `C()`, destacados em preto. No ***Algoritmo 2***, esses métodos são adicionados na fila e inicia-se o percurso em largura pelos métodos chamados por eles. O primeiro elemento a ser retirado da fila é o método `A()` (considerando que a fila esteja em ordem alfabética), fazendo com que os métodos `D()` e `E()` sejam visitados. A condição na linha 9 falha para o método `D()`, visto que existe uma chamada (seta pontilhada) de um método não pertence ao conjunto de `métodosMortos`. No entanto, essa condição é satisfeita para o método `E()`, considerado código morto e adicionado no final da fila. Em seguida, serão retirados da fila o método `B()` e, posteriormente, o método `C()`, fazendo com que todos os métodos do primeiro nível da hierarquia de chamadas sejam visitados e analisados.

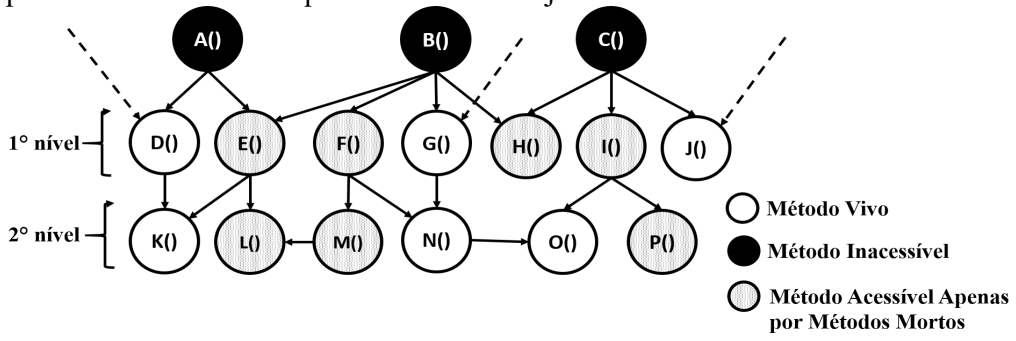

Figura 3. Exemplo de Chamadas entre Métodos

Após a análise dos métodos do primeiro nível, a fila conterá os métodos `E()`, `F()`, `H()` e `I()`, por satisfazerem a condição na linha 9. Dessa forma, com o percurso em largura, os métodos chamados por esses métodos são analisados, apresentados no

segundo nível da hierarquia de chamadas. A condição na linha 9 é satisfeita pelos métodos `L()`, `M()` e `P()`, também considerados inacessíveis e adicionados no final da fila. Dentre esses métodos, apenas o método `M()` efetua uma chamada para o método `L()`, identificado como morto (falha da condição na linha 7). A busca é finalizada e os métodos mortos identificados estão armazenados em `métodosMortos` e correspondem aos métodos destacados em preto e em cinza na Figura 3.

4. Avaliação da Utilização da Técnica Análise de Acessibilidade

Nessa avaliação, o objetivo é verificar a capacidade de detecção de código morto da análise de acessibilidade proposta, em relação a outras abordagens e ferramentas que realizam a detecção de código morto. Dessa forma, foi formulada a seguinte questão de investigação:

"A aplicação da análise de acessibilidade proposta é capaz de identificar métodos inacessíveis tanto quanto uma ferramenta consolidada no mercado?"

Cinco sistemas de software desenvolvidos com a linguagem de programação Java foram analisados por um *plug-in* do Eclipse que automatiza a proposta e pela ferramenta Understand (https://scitools.com).

4.1. Caracterização dos Sistemas de Software Analisados

A avaliação foi conduzida utilizando a versão mais recente de cinco sistemas de software selecionados (Tabela 1) por: i) serem habitualmente utilizados em pesquisas sobre qualidade de software; ii) serem de código aberto; iii) serem desenvolvidos em Java e; iv) possuírem uma quantidade variada de métodos.

Tabela 1. Características dos Sistemas de Software Utilizados

Software	Descrição	Endereço Eletrônico	Pacotes	Classes	Métodos
ArgoUML 0.34	Ferramenta de modelagem para projeto, desenvolvimento e documentação de aplicações orientadas a objetos, utilizando a notação UML (*Unified Modeling Language*).	www.argouml.org	238	2895	19633
FreeMind 1.0.1	Ferramenta para criação de mapas mentais, que permitem representar uma ideia ou um conjunto de ideias de forma visual.	www.freemind.sourceforge.net	88	1015	7676
JabRef 3.2	Ferramenta para gerenciamento de referências bibliográficas utilizando o padrão BibTeX	www.jabref.org	160	1284	6904
JEdit 5.3.0	Ferramenta de edição de textos voltada para programadores, com uma arquitetura de *plug-ins* extensível.	www.jedit.org	66	1082	8119
JHotDraw 7.6	*Framework* para criação de editores gráficos, como editores de diagramas UML e redes de Petri.	www.jhotdraw.org	74	758	7798

4.2 Execução da Avaliação

Existe carência de ferramentas funcionais fundamentadas em abordagens para detecção de métodos inacessíveis em sistemas de software Java. Algumas das ferramentas

existentes possuem limitações que as impediram de serem utilizadas nessa avaliação. Na Tabela 2, são apresentados o nome e o endereço eletrônico dessas ferramentas, além da razão pela qual não foram utilizadas. Dentre essas ferramentas, a única que possui uma abordagem embasada na literatura é DUM-Tool [Romano *et al*., 2016]. No entanto, essa ferramenta não foi utilizada por apresentar problemas de execução durante a análise de alguns sistemas de software.

Tabela 2. Ferramentas não Utilizadas na Avaliação

Ferramenta	Endereço Eletrônico	Motivo para não ser utilizada
CodePro Analytics	https://developers.google.com/java-dev-tools/codepro	Apresentou incoerência significativa dos resultados obtidos em um projeto piloto
DUM-Tool	http://www2.unibas.it/sromano/DUM-Tool.html	Não funcional; Necessita de conhecimento prévio do software para indicar os métodos de inicialização (*main*).
JTombstone	https://sourceforge.net/projects/jtombstone/	Necessita de conhecimento prévio do software.
PMD	https://sourceforge.net/projects/pmd/	Analisa apenas métodos privados.
UCDetector	http://www.ucdetector.org/	Analisa apenas métodos públicos.

Apesar de não estar relacionada a uma abordagem, a ferramenta Understand foi utilizada nessa avaliação por ser funcional e consolidada no mercado. Essa ferramenta foi desenvolvida por SciTools (*Scientific Toolworks* Inc.), entidade que desenvolve sistemas para manutenção de software desde 1996. Essa ferramenta realiza análise estática do código de sistemas de software desenvolvidos em diversas linguagens de programação, permitindo a análise, a medição e a compreensão desses sistemas. Uma de suas funções é a detecção de código morto em sistemas de software Java. Assim como na abordagem proposta, essa ferramenta analisa os métodos do software independente de sua visibilidade. A avaliação foi executada analisando os sistemas de software apresentados na Tabela 1, utilizando Understand e o *plug-in* desenvolvido.

4.3 Resultados e Discussões

Na Tabela 3, é apresentada a quantidade de métodos mortos identificada pelo *plug-in* desenvolvido e por Understand, assim como o percentual dessa quantidade em relação à quantidade total de métodos do software. Como pode ser observado, com a técnica análise de acessibilidade utilizada no *plug-in*, foi identificada maior quantidade de métodos inacessíveis do que Understand em todos os sistemas de software analisados.

Tabela 3. Quantidade de Métodos Mortos Identificada pelas Ferramentas

Software	Quantidade de Métodos Detectados com o *Plug-in*	%	Quantidade de Métodos Detectados com Understand	%
ArgoUML	3.201	16%	1.860	9%
FreeMind	703	9%	495	6%
JabRef	1.432	21%	1.151	17%
JEdit	1.583	19%	987	12%
JHotDraw	1.457	19%	878	11%

Examinando essa quantidade em relação aos métodos do software, a análise de acessibilidade identificou que, em média, aproximadamente 17% dos métodos dos sistemas de software analisados são inacessíveis. Em contrapartida, Understand identificou, em média, aproximadamente 11% de métodos inacessíveis em relação a todos os métodos do software. Os resultados obtidos foram analisados de forma mais precisa considerando três casos:

- **Caso 1**. Métodos inacessíveis identificados por Understand e pelo *plug-in*;

- **Caso 2**. Métodos inacessíveis identificados por Understand, mas não identificados pelo *plug-in*;
- **Caso 3**. Métodos inacessíveis não identificados por Understand, mas identificados pelo *plug-in*.

Após a identificação dos métodos mortos, ambas as ferramentas geraram um arquivo de *log* com os resultados obtidos. Foi desenvolvido um *script* para comparar esses arquivos de *log* e identificar quais métodos mortos foram identificados pelas duas ferramentas (Caso 1), métodos mortos identificados apenas por Understand (Caso 2) e métodos mortos identificados apenas pelo *plug-in* (Caso 3). Os métodos listados com a execução desse *script* foram analisados manualmente para identificar suas características. Seja A_S o conjunto de métodos identificados pelo *plug-in* e U_S o conjunto de métodos identificados por Understand, o total T_S de métodos identificados para um software S pode ser dado por $T_S = A_S \cup U_S$. Na Figura 4, é apresentada uma visão geral da proporção de A_S e de U_S em relação ao total de métodos inacessíveis T_S para os sistemas de software analisados.

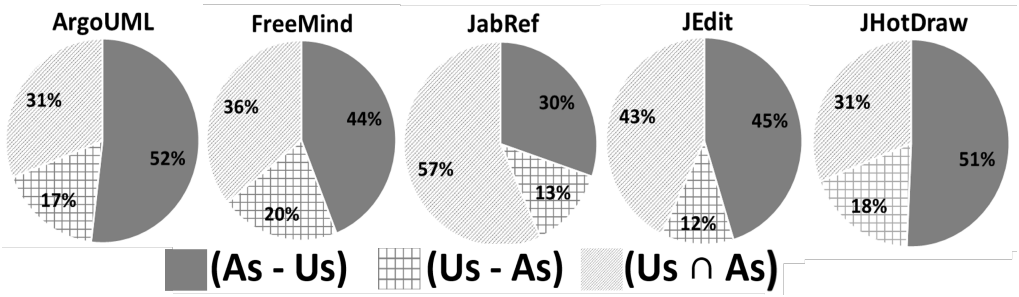

Figura 4. Percentual de Métodos Inacessíveis Identificados pelas Ferramentas

Os métodos pertencentes ao **Caso 1** podem ser obtidos por $U_S \cap A_S$, para um dado software S. Considerando, respectivamente, os sistemas de software ArgoUML, FreeMind, JabRef, JEdit e JHotDraw, essa operação mostrou que, aproximadamente, 64% (1.194 de 1.860 métodos), 64% (315 de 495 métodos), 81% (935 de 1.151 métodos), 78% (770 de 987 métodos) e 63% (553 de 878 métodos) dos métodos inacessíveis identificados por Understand foram identificados pelo *plug-in*. Na Figura 4, esse percentual pode ser visualizado em relação à quantidade total T_S. De modo geral, esses percentuais indicam que, em média, 70% dos métodos inacessíveis identificados por Understand foram identificados pelo *plug-in*. Em relação a T_S, aproximadamente, 40% dos métodos inacessíveis foram identificados por Understand e pelo *plug-in*.

Os métodos identificados por Understand, mas não identificados pelo *plug-in* (**Caso 2**), podem ser obtidos por $U_S - A_S$. Na Tabela 4, é apresentada a quantidade desses métodos em relação ao seu tipo. Como pode ser observado, construtores, métodos abstratos e métodos relacionados à interface gráfica (`listeners`) correspondem a maioria dos métodos não identificados pelo *plug-in*. Conforme discutido na Seção 3.1, métodos com essas características foram intencionalmente desconsiderados do processo de detecção. O propósito da abordagem proposta é analisar apenas a acessibilidade de métodos; por esse motivo, tipos enumerados foram identificados apenas por Understand. Além disso, pode ser observado que, em média, aproximadamente 5% dos métodos identificados apenas por Understand possuíam chamadores acessíveis e aproximadamente 1% desses métodos eram realmente mortos.

Esses percentuais podem indicar pequena falha no processo de detecção de Understand e do *plug-in*. Métodos declarados em blocos internos e em classes internas (*inner classes*) não foram detectados pelo *plug-in*. Esse fato não corresponde a uma restrição ou a uma limitação da abordagem proposta e está relacionado a algum problema técnico que ocorreu durante a execução do *plug-in* Eclipse.

Tabela 4. Tipo dos Métodos Pertencentes ao Caso 2

Tipo	ArgoUML	FreeMind	JabRef	JEdit	JHotDraw
Construtores	347 - 51%	68 - 38%	45- 21%	110 - 52%	164 - 51%
Métodos em Blocos Internos	65 - 10%	22 - 12%	60 - 28%	42 - 19%	73 - 22%
Tipo Enumerado	10 - 2%	-	20 - 9%	7 - 3%	7 - 2%
Métodos Com Chamadores	4 - 1%	9 - 5%	44 - 20%	-	3 - 1%
Métodos Abstratos	180 - 27%	31 - 17%	5 - 2%	26 - 12%	75 - 23%
Métodos de Interface Gráfica	51 - 8%	16 - 9%	37 - 17%	2 - 1%	-
Métodos em *Inner Classes*	9 - 1%	33 - 19%	2 - 1%	25 - 11%	-
Métodos Inacessíveis	-	1 - 0%	3 - 2%	5 - 2%	3 - 1%
Total de Métodos do Caso 2	**666**	**180**	**216**	**217**	**325**

Na Tabela 5, são apresentados os tipos e a quantidade de métodos pertencentes ao **Caso 3** ($A_s - U_s$). Em média, aproximadamente, 33% dos métodos do Caso 3 são chamados apenas por métodos inacessíveis. Cerca de 1% dos métodos são chamados por métodos inacessíveis e sobrescrevem métodos de superclasses. Além disso, 6% dos métodos foram considerados inacessíveis por não possuírem chamadores e, aproximadamente, 59% dos métodos são inacessíveis e sobrescrevem métodos de superclasses. Alguns métodos (~1%) possuem algum método acessível em sua hierarquia de chamadas, podendo ser considerados vivos.

Tabela 5. Tipo dos Métodos Pertencentes ao Caso 3

Tipo	ArgoUML	FreeMind	JabRef	JEdit	JHotDraw
Chamado por Métodos Inacessíveis	422 - 21%	116 - 30%	194 - 39%	366 - 45%	267 - 30%
Chamado por Métodos Inacessíveis (*Override*)	21 - 1%	23 - 6%	-	-	-
Métodos Vivos	8 - 0%	23 - 6%	5 - 1%	-	-
Métodos sem Chamador	280 - 14%	16 - 4%	40 - 8%	16 - 2%	-
Métodos sem Chamador (*Override*)	1.276 - 64%	210 - 54%	258 - 52%	431 - 53%	637 - 70%
Total de Métodos do Caso 3	**2.007**	**388**	**497**	**813**	**904**

Apesar da maioria dos métodos identificados por Understand (~70%) terem sido identificados pelo *plug-in*, algumas características adotadas na implementação ocasionaram divergências nos resultados obtidos. Ao contrário de Understand, o *plug-in* analisou os métodos do software sem considerar questões de herança, como a sobrescrita de métodos. Essa característica faz com que métodos chamados via polimorfismo possam ser detectados como inacessíveis. Em contrapartida, Understand analisou construtores, métodos abstratos e métodos de interface gráfica. A análise de métodos relacionados à interface gráfica é uma limitação de abordagens que utilizam análise estática, visto que a captura de chamadas desses métodos depende da execução do software.

No que se refere a execução da técnica análise de acessibilidade em duas fases, a execução da segunda fase aumentou a capacidade de detecção de métodos inacessíveis da abordagem. Na Figura 5, é apresentado o percentual de métodos inacessíveis detectados pela segunda fase em relação à quantidade total de métodos inacessíveis identificados. Como pode ser observado, essa análise adicional identificou, em média, aproximadamente 20% do total de métodos inacessíveis para os cinco sistemas de software analisados. Considerando, respectivamente, os sistemas de software

ArgoUML, FreeMind, JabRef, JEdit e JHotDraw, aproximadamente 34% (151 métodos de 443), 8% (11 métodos de 139), 0% (0 métodos de 194), 13% (48 métodos de 366) e 19% (50 métodos de 267) dos métodos identificados na segunda fase da análise de acessibilidade também foram identificados por Understand.

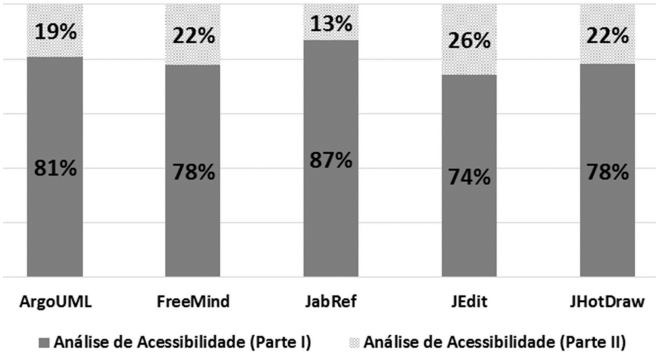

Figura 5. Percentual de Métodos Inacessíveis Detectados pela Técnica Análise de Acessibilidade (Fase I) e (Fase II)

Com base nos resultados descritos, obteve-se a seguinte conclusão para a questão de investigação:

- **Questão**.

"A aplicação da análise de acessibilidade proposta é capaz de identificar métodos inacessíveis tanto quanto uma ferramenta consolidada no mercado?"

- **Conclusão**. Parcialmente, sim. Com a análise dos resultados obtidos, pode-se observar que a resposta exata dessa pergunta depende de especificidades adotadas na implementação das ferramentas utilizadas na avaliação. Foi possível perceber que parte significativa dos métodos inacessíveis identificados por Understand foram identificados pelo *plug-in*. Algumas características adotadas na implementação ocasionaram divergências nos resultados. Na Figura 6, é apresentada uma síntese dessas características. Por exemplo, a maior parte dos métodos não identificados pelo *plug-in* correspondem às restrições intencionalmente definidas na linha 6 do ***Algoritmo 1***, que impede a análise de construtores, métodos abstratos e métodos de interface gráfica. Além disso, ao contrário de Understand, a abordagem proposta analisou a acessibilidade desconsiderando sobrescritas de métodos de superclasses, resultando em maior quantidade de métodos inacessíveis identificados.

Em suma, considerando que, aproximadamente, 70% dos métodos identificados por Understand foram identificados pelo *plug-in* e parte dos métodos não identificados correspondem às restrições adotadas na sua implementação, conclui-se que a análise de acessibilidade proposta é parcialmente capaz de detectar código morto tanto quanto Understand, desde que sejam alinhadas as questões de implementação identificadas.

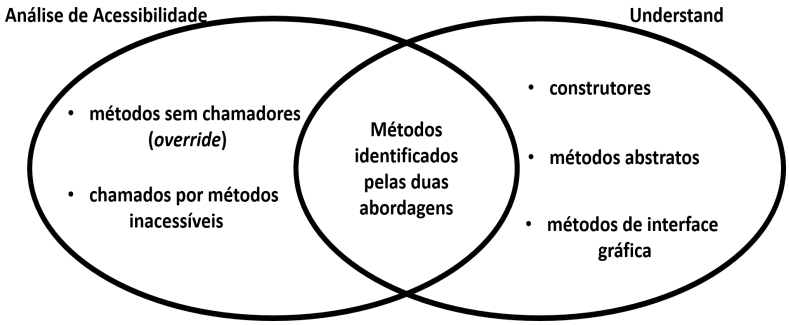

Figura 6. Principais Características Adotadas na Implementação das Ferramentas

5. Trabalhos Relacionados

A técnica análise de acessibilidade foi aplicada de diferentes formas para detectar código inacessível. Por exemplo, um estudo pioneiro definiu um modelo de dados que permite analisar a acessibilidade de entidades de software desenvolvido em C++ [Chen *et al*., 1998]. Esse modelo de dados deve satisfazer um critério de completude e abstrair as informações contidas em repositórios de código, mapeando as entidades do software, suas dependências e os relacionamentos existentes entre elas. O critério de completude é satisfeito se todas as dependências existentes no código forem mapeadas para o modelo de dados. A análise de acessibilidade é executada nesse modelo de dados para detecção de código inacessível a nível de declarações, funções ou arquivos. O primeiro passo dessa análise é identificar um conjunto de entidades alcançáveis a partir de entidades que iniciam a execução do software. O segundo passo é identificar as entidades do software. No terceiro passo, o conjunto de entidades inacessíveis é obtido pela diferença entre as entidades do software com as entidades alcançáveis.

Um estudo aplicou a técnica análise de acessibilidade com objetivo de identificar especificamente métodos inacessíveis em sistemas de software desenvolvidos em Java ou em C++ [Bacon *et al*., 2003]. De forma similar ao estudo anterior, o código inacessível é detectado pela diferença entre o conjunto dos métodos do software e o conjunto de métodos acessíveis. Nesse estudo, foram considerados como métodos acessíveis os responsáveis por iniciar a execução e os acessíveis a partir de bibliotecas utilizadas pelo software. As informações para executar a análise de acessibilidade e para verificar possíveis chamadas de bibliotecas aos métodos são obtidas de forma estática diretamente no código do software.

Com a abordagem DUM (*Detecting Unreachable Methods*), *bytecodes* Java são analisados e uma representação é criada baseada em grafo do software, no qual os nós correspondem aos métodos e as arestas correspondem às chamadas existentes entre eles [Romano *et al*., 2016]. A análise de acessibilidade é executada utilizando a representação em grafo em três passos, conforme descrito nos estudos anteriores. Os métodos considerados como acessíveis são métodos `main()`, métodos que iniciam campos ou blocos e métodos relacionados com a serialização/desserialização de objetos, invocados via reflexão.

De modo geral, os estudos apresentados aplicaram a técnica análise de acessibilidade de forma semelhante, com diferença apenas na maneira na qual as informações do software são obtidas ou no tipo de código inacessível detectado

(métodos, declarações ou arquivos). Essas aplicações necessitam de conhecimento prévio do software, visto que é necessário identificar um conjunto de métodos acessíveis antes de realizar a detecção de código morto propriamente dita. O principal diferencial da aplicação da técnica análise de acessibilidade proposta é a identificação de métodos inacessíveis sem a necessidade de conhecer os métodos acessíveis do software. Dessa forma, a identificação de código morto pode ser realizada de forma direta e objetiva.

6. Considerações Finais

Neste trabalho, foi proposta uma aplicação da técnica análise de acessibilidade para identificar métodos inacessíveis em sistemas de software orientado a objetos. Ao contrário de abordagens existentes, a acessibilidade de um método é verificada sem a necessidade de conhecer os métodos de iniciação do software. Essa abordagem foi implementada em um *plug-in* para o Eclipse e avaliada em relação à ferramenta Understand. Os resultados dessa avaliação indicaram que, com a abordagem proposta, foi detectada maior quantidade de métodos inacessíveis do que Understand, sendo que 70% dos métodos identificados por Understand foram identificados pelo *plug-in*.

Uma limitação da abordagem proposta é a utilização de análise estática para detecção do código morto. Apesar de mais flexível que análise dinâmica, esse tipo de análise pode gerar alguns resultados inconsistentes. Por exemplo, métodos invocados durante a execução do sistema de software via reflexão ou polimorfismo podem ser detectados indevidamente como inacessíveis, por não possuírem chamadas explícitas de outros métodos. Como ameaças a validade, foi realizada a detecção de código morto nas classes e nos pacotes do software. Dessa forma, módulos em desenvolvimento e que contenham métodos ainda não utilizados podem ter sido analisados e identificados precipitadamente como inacessíveis. Além disso, o correto funcionamento do *script* utilizado na comparação dos métodos e possível fadiga causada nos pesquisadores durante a análise manual desses métodos podem ser considerados atributos que interferem na validade dos resultados apresentados.

Como trabalhos futuros, espera-se realizar refinamentos na abordagem proposta de forma a considerar algumas questões como polimorfismo ou reflexão, sem deixar de utilizar análise estática. Espera-se aprimorar a implementação do *plug-in* para que sejam analisados métodos presentes em *inner classes*. Além disso, espera-se utilizar a abordagem proposta para detectar métodos inacessíveis em diferentes versões do software e apresentar sua evolução utilizando técnicas de visualização de software.

Referências

Bacon, D. F.; Laffra, J. C.; Sweeney, P. F.; Tip, F. Removal of Unreachable Methods in Object-Oriented Applications Based on Program Interface Analysis. Patente Número US6654951 B1. 2003.

Bakota, T.; Hegedus, P.; Siket, I.; Ladányi, G.; Ferenc, R. Qualitygate SourceAudit: A Tool for Assessing the Technical Quality of Software. In: Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering. pp. 440-445. 2014.

Bastos, C; Júnior, P. A. P; Costa, H. Técnicas para Detecção de Código Morto: Uma Revisão Sistemática de Literatura. In: Simpósio Brasileiro de Sistemas de Informação. pp. 255-262. 2016.

Boomsma, H.; Gross, H.G. Dead Code Elimination for Web Systems Written in PHP: Lessons Learned from an Industry Case. In: International Conference of Software Maintenance. pp. 511-515. 2012.

Chen, Y.; Emden, R. G.; Eleftherios, K. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In: IEEE Transactions on Software Engineering. v. 24, n. 9, p. 682-694. 1998.

Eder, S.; Junker, M.; Jürgens, E.; Hauptmann, B.; Vaas, R.; Prommer, K. How Much Does Unused Code Matter for Maintenance? In: International Conference on Software Engineering. pp. 1102-1111. 2012.

Fehnker, A.; Huuck, R. Model Checking Driven Static Analysis for the Real World: Designing and Tuning Large Scale Bug Detection. In: Innovations in Systems and Software Engineering. v. 9. pp. 45-56. 2013.

Gold, N.; Mohan, A. A Framework for Understanding Conceptual Changes in Evolving Source Code. In: International Conference on Software Maintenance. pp. 431-439. 2003.

Guerrouat, A.; Richter, H. A Combined Approach for Reachability Analysis. In: International Conference on Software Engineering Advances. pp. 23-23. 2006.

Lehman, M. M.; Belady, L. Program Evolution: Processes of Software Change. Academic Press. 538p. 1985.

Martin, R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall. Ed 1. 431p. 2008.

Martins, R. C.; Pellegrino, S. R. M.; Santellano, J. Tratamento de “Dead Codes” em Software de uso Aeronáutico. In: Brazilian Conference on Dynamics, Control and their Applications. 2010.

Pérez, R. C.; Guzman, I. G. R.; Piattini, M. Diagonosis of Software Erosion Through Fuzzy Logic. In: Symposium on Computacional Intelligence in Dynamic Dynamic and Uncertain Environments. pp. 49-56. 2011.

Romano, S.; Scanniello, G.; Sartiani, C.; Risi, M. A Graph-Based Approach to Detect Unreachable Methods in Java Software. In: ACM Symposium on Applied Computing. pp. 1538-1541. 2016.

Scanniello, G. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In: Conference on Software Engineering and Advanced Applications. pp. 392-397. 2014.

Srivastava, A. Unreachable Procedures in Object-Oriented Programming. In: ACM Letters on Programming Languages and Systems. pp. 355-364. 1992.

Sunitha, K. V. N.; Kumar, V. V. A New Technique for Copy Propagation and Dead Code Elimination Using Hash Based Value Numbering. In: International Conference on Advanced Computing and Communications. pp. 601-604. 2006.

Tjortjis, C.; Layzell, P. Expert Maintainers' Strategies and Needs When Understanding Software: A Case Study Approach. In: Asia-Pacific Software Engineering Conference. pp. 281-287. 2001.