MALLÚ EDUARDA BATISTA

UMA ABORDAGEM HÍBRIDA PARA DETECÇÃO DE CÓDIGO CLONADO EM LINHAS DE PRODUTOS DE SOFTWARE

LAVRAS-MG
2020

MALLÚ EDUARDA BATISTA

UMA ABORDAGEM HÍBRIDA PARA DETECÇÃO DE CÓDIGO CLONADO EM LINHAS DE PRODUTOS DE SOFTWARE

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Banco de Dados e Engenharia de Software, para a obtenção do título de Mestre.

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2020

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Batista, Mallú Eduarda.
Uma abordagem híbrida para detecção de código clonado em Linhas de Produtos de Software / Mallú Eduarda Batista. - 2020.
100 p. : il.

Orientador(a): Heitor Augustus Xavier Costa.

Dissertação (mestrado acadêmico) - Universidade Federal de Lavras, 2020.
Bibliografia.

1. Clonagem de Código. 2. Linha de Produtos de Software. 3. Abordagem Híbrida. I. Costa, Heitor Augustus Xavier.

MALLÚ EDUARDA BATISTA

UMA ABORDAGEM HÍBRIDA PARA DETECÇÃO DE CÓDIGO CLONADO EM LINHAS DE PRODUTOS DE SOFTWARE

A HYBRID APPROACH FOR CLONED CODE DETECTION IN SOFTWARE PRODUCT LINES

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Banco de Dados e Engenharia de Software, para a obtenção do título de Mestre.

APROVADA em 10 de dezembro de 2020

Dr. Paulo Afonso Parreira Júnior - UFLA
Dr. Eduardo Magno Lages Figueiredo - UFSJ

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2020

À minha família pelo amor e apoio.
Dedico

AGRADECIMENTOS

Agradeço a Deus por se fazer presença constante na minha vida me guiando na realização dos meus objetivos e sonhos.

Aos meus pais pelo incentivo e apoio durante todo esse período.

Às minhas tias e tios pelo carinho e companheirismo.

Aos meus avós pelo amor e cuidado.

Aos meus amigos pela compreensão.

Ao meu orientador pela persistência e ensinamentos.

À todos que fizeram parte da minha vida durante esse ciclo.

Ao programa de Pós-Graduação em Ciência da Computação.

À coordenação de Aperfeiçoamento de Pessoal no Nível Superior (CAPES), pelo apoio financeiro.

RESUMO

Linha de Produtos de Software (LPS) corresponde a sistemas de software que compartilham um conjunto comum de funções (comunalidades) que foram desenvolvidas adotando uma base de ativos comum, acrescidas de variações (variabilidades) que são importantes características para diferir os produtos de uma mesma família de sistemas. Códigos clonados adentram o escopo de desenvolvimento de sistemas de software por diversas razões, como copiar e colar, reutilizar código, adicionar funções e aumento e derivação de dados. Dada a existência de quatro Tipos de clones, sendo eles classificados quanto a sua semelhança sintática (Tipos 1, 2 e 3) ou semântica (Tipo 4), tem-se vários relatos da ocorrência dos mesmos em sistemas orientados a objetos cuja relações estão diretamente ligadas a qualidade e a manutenibilidade de software. No contexto de desenvolvimento de LPS, é possível detectar a existência de código clonado e efetuar alterações necessárias para que esse código não seja propagado para outros produtos originários dessa LPS. Neste trabalho de Mestrado, o objetivo é abordar como detectar clonagem de código em LPS, apresentando uma proposta de abordagem híbrida de detecção de clones de código dos Tipos 1 e 2 em LPS. Tal abordagem é baseada na construção de gráfico de dependências utilizando a AST e consiste na utilização de técnicas para análise estática de código, sustentada pela análise de sequência de chamadas de métodos e na análise estrutural de assinaturas de métodos. Para que se possa (semi) automatizar essa proposta de abordagem, foi implementado um apoio computacional (*plug-in* para a plataforma Eclipse IDE) e realizada a avaliação dessa proposta, por meio da avaliação interna e externa onde foram mensurados os valores da precisão e *recall* na utilização desse apoio e realizada a comparação com outra ferramenta de detecção de clones. Os resultados obtidos mostram uma alta precisão e confiabilidade na utilização da ferramenta desenvolvida para detectar clones em LPS orientada a característica além da comparação dos clones obtidos em relação a ferramenta CPD do PMD.

Palavras-chave: Clonagem de Código, Linha de Produtos de Software.

ABSTRACT

Software Product Line (SPL) corresponds to software systems that share a common set of functions (commonality) that were developed using a common asset base, plus variations (variability) that are important characteristics to differentiate products from the same systems family. Cloned codes enter the scope of software systems development for several reasons, such as copy and paste, reuse code, add functions and increase and derive data. Given the existence of four types of clones, being classified according to their syntactic (Types 1, 2 and 3) or semantic similarity (Type 4), there are several reports of their occurrence in object-oriented systems whose relations are directly linked to software quality and maintainability. In the context of LPS development, it is possible to detect the existence of cloned code and make necessary changes so that this code is not propagated to other products originating from that LPS. In this Master's work, the objective is to address how to detect cloning of code in LPS, presenting a proposal for a hybrid approach to detecting Type 1 and 2 code clones in LPS. Such an approach is based on the construction of a dependency graph using AST and consists of the use of techniques for static code analysis, supported by the analysis of the sequence of method calls and the structural analysis of method signatures. In order to be able to (semi) automate this proposal of approach, a computational support (plug-in for the Eclipse IDE platform) was implemented and the evaluation of this proposal was carried out, through the internal and external evaluation where the values of precision and recalling the use of this support and comparing it with another clone detection tool. The results obtained show a high precision and reliability in the use of the tool developed to detect clones in LPS oriented to the characteristic besides the comparison of the clones obtained in relation to the PMD CPD tool.

Keywords: Code Cloning, Software Product Line.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE SIGLAS

AHEAD - *Algebraic Hierarchical Equations for Application Design*

APT - *Annotation Processing Tool*

AST - *Abstract Syntax Tree*

ELPS - Engenharia de Linha de Produto de Software

FODA - *Feature Oriented Domain Analysis*

IWSC - *International Workshop on Software Clones*

JDT - *Java Development Tools*

LOC - Line of Code

LPS - Linha de Produtos de Software

MSL - Mapeamento Sistemático da Literatura

PDE - *Plug-in Development Environment*

PDG - *Program Dependence Graph*

POO - Paradigma Orientado a Objetos

POC - Paradigma Orientado a Características

QP - Questão de Pesquisa

RSL - Revisão Sistemática da Literatura

CPD - *Copy Past Detection*

SUMÁRIO

1 INTRODUÇÃO

Uma das inovações tecnológicas da era industrial é conhecida como linha de produção ou linha de montagem. Concebida no ano de 1913 por Henry Ford [Corrêa; Corrêa, 2000], o processo de produção em série possibilitou a redução de custos e a produção em massa. Aliados à evolução e à automação dos processos na produção industrial, destaca-se, na área de Engenharia de Software, a evolução de sistemas de software legados ou criação de novos sistemas de software utilizando o conceito de Linha de Produtos de Software (LPS) (*Software Product Line* - SPL) [Laguna; Crespo, 2013]. Tal conceito atende aos sistemas de software que compartilham um conjunto comum de funções (comunalidades) desenvolvidas a partir de uma base em comum e a seleção de características de forma composicional, que criam a variedade de combinação na geração de produtos (variabilidades). Esses sistemas são destinados a atender necessidades de um segmento de mercado ou missão.

Clones de código são encontrados em diversos contextos de desenvolvimento de sistemas de software. Com a identificação desses clones, pode-se prevenir a propagação de erros e a identificação e a correção de *bugs*, o que acarreta menos dificuldades de manutenção. Trechos de código fonte podem ser idênticos ou possuir similaridades e, por esse motivo, são classificados como clones de código, uma vez que surgem de diversas maneiras no código e são comumente oriundos de práticas de "*copy and paste*" por programadores de forma intencional ou não. Algumas dessas práticas são reutilização de código *ad-hoc* e adição de funções semelhantes.

Por conseguinte, pesquisas recentes mostram a existência de vasta gama de ferramentas que realizam a detecção de clones de código em sistemas de software. Elas identificam tipos variados de clones e são implementadas por técnicas baseadas, em sua maioria, em texto (*text-based*), *tokens* (*token-based*), árvores (*tree-based*), grafos (*graph-based*), medidas (*metric-based*) e híbridas (*hybrid-based*). No cenário de detecção de clones, a maioria dessas ferramentas é desenvolvida para detectar clones de código em sistemas de software implementados no Paradigma Orientado a Objetos (POO). Contudo, por se tratar de um paradigma emergente, o Paradigma Orientado a Características (POC) vem se destacando na utilização para implementar LPS, isto é, Linhas de Produtos de Software Orientada a Características [Schulze *et al.*, 2011].

1.1 Motivação

Como uma LPS gera vários produtos derivados de comunalidades e acrescidos de variabilidades, a detecção de clones diretamente na LPS visa à propagação de alterações por todos os produtos dessa família de software. A utilização do POC na implementação de LPS é beneficiada por não possuir limitações especificadas por conceitos de herança, por exemplo. Posterior ao processo de criação da LPS, os processos de evolução e de adição de *features* são comuns e frequentes quando se deseja criar ou modificar um produto. Tais processos podem levar a alterações no código fonte feitas de maneira não estratégica, podendo ocasionar o reúso frequente de código a fim de implementar os novos requisitos do cliente causando a duplicação de código. Constatada a quase inexistência de pesquisas acerca da detecção de clones de código em LPS [Batista *et al.*, 2019], pode ser observada uma lacuna de pesquisa a ser preenchida com os resultados esperados com a realização deste trabalho, pois não foram detectadas ferramentas utilizadas em LPS nem abordagens destinadas para tal contexto.

1.2 Objetivo

Neste trabalho, o objetivo é proporcionar ao Engenheiro de Software a detecção de código clonado em LPS. Para isso, foi elaborada uma abordagem híbrida para detecção de clones em código fonte, baseada na análise de chamadas de métodos e na estrutura de assinaturas de métodos em LPS. Para dar suporte ao desenvolvimento dessa abordagem, foram estabelecidos os seguintes objetivos específicos:

a) **Estado da arte**. Para melhor entendimento do contexto de Clonagem de Código e de LPS, foi realizado um estudo desses conceitos utilizando referências presentes na literatura;
b) **Identificação de abordagens existentes**. Para ter apoio para o desenvolvimento da abordagem híbrida, foi necessário identificar a existência de ferramentas e de abordagens para detecção de clones de código e em qual paradigma ambas são utilizadas;
c) **Desenvolvimento de um apoio computacional**. Para (semi) automatizar a abordagem híbrida de detecção de código clonado, foi desenvolvido um apoio computacional que a implementa. Posteriormente, foram realizadas duas avaliações: i) avaliação de detecção de código clonado para determinar a precisão e confiabilidade de utilização da ferramenta (*recall*); e ii) avaliação comparativa com ferramenta existente.

1.3 Método de Pesquisa

A metodologia da pesquisa determina formas a serem utilizadas para reunir dados necessários para o êxito do trabalho, utilizando técnicas de coleta e de análise de dados [Moresi *et al.*, 2003]. Uma pesquisa científica pode ser classificada sob vários aspectos [Moresi *et al.*, 2003], por exemplo:

a) **Natureza**. Do ponto de vista de sua natureza, a pesquisa pode ser classificada em: i) Básica. Gera conhecimentos úteis para obter avanços na ciência, sem previsão de aplicação prática, envolvendo verdades e interesses universais; e ii) Aplicada. Gera conhecimento aplicável na prática, buscando solução para problemas específicos e envolvendo verdades e interesses locais. Este trabalho pode ser classificado como **pesquisa aplicada**, pois visa ao desenvolvimento de uma abordagem híbrida de detecção de clones de código em LPS;

b) **Abordagem**. Do ponto de vista da abordagem do problema, a pesquisa pode ser classificada em: i) Quantitativa. Traduz informações em números para classificá-las e analisá-las, utilizando técnicas de estatística; e ii) Qualitativa. Utiliza interpretação de fenômenos e atribuição de significado, sendo considerada descritiva. Este trabalho pode ser classificado como **pesquisa qualitativa**, pois analisa e interpreta o problema por meio de uma abordagem proposta;

c) **Finalidade**. Do ponto de vista da finalidade, a pesquisa pode ser classificada em: i) Exploratória. Quando existe pouco conhecimento acumulado e sistematizado; ii) Descritiva. Quando a pesquisa é direcionada para mostrar características de fenômenos ou populações; iii) Explicativa. Tenta tornar algo compreensível por meio de justificativa dos motivos; iv) Metodológica. Refere-se à elaboração de instrumentos de captação ou de manipulação da realidade; e v) Intervencionista. Interfere na realidade estudada visando modificá-la. Este trabalho pode ser classificado como **pesquisa exploratória e metodológica**, visto que foram encontrados poucos estudos sobre detecção de clones de código em LPS em um mapeamento sistemático da literatura realizado e a abordagem de detecção proposta disporá de um apoio computacional que (semi) automatiza essa abordagem;

d) **Meios de investigação**. Do ponto de vista da maneira de buscar por informações, a pesquisa pode ser classificada em: i) de Campo. Consiste na investigação empírica no local onde ocorre/ocorreu um fenômeno ou que disponibiliza elementos que o explica; ii) de Laboratório. A experiência é realizada em local restrito; iii) Telematizada. Utiliza computador e telecomunicações; iv) Documental. Faz investigação em documentos; v)

Bibliográfica. O estudo sistematizado desenvolvido baseia-se em material disponível ao público em geral; vi) Experimental. A investigação empírica utiliza variáveis independentes de forma a manipulá-las e controlá-las para observar variações que elas surtem em variáveis dependentes; vii) *Ex post facto*. Refere-se a um fato ocorrido; viii) Participante. Introduz a fronteira pesquisador/pesquisado ao contexto do problema investigado; ix) Pesquisa-ação. Supõe intervencionar de maneira participativa na realidade social; e x) Estudo de caso. Delimitado a uma ou poucas unidades e tem caráter de detalhamento. Este trabalho pode ser classificado como **pesquisa bibliográfica e experimental**, pois realiza o estudo sistemático de materiais contidos em bibliotecas digitais, livros e outros e realiza experimentos com dados e discussão de resultados.

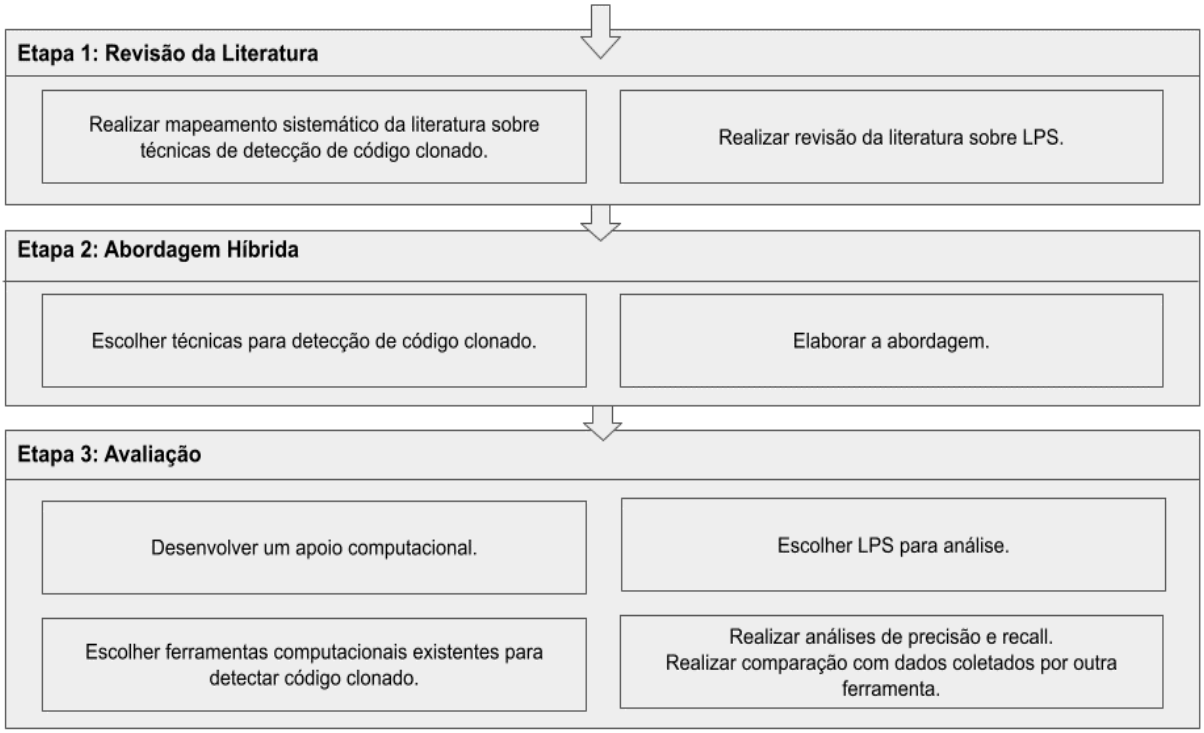
O presente trabalho iniciou-se em março de 2018 e terminou em dezembro de 2020. Neste trabalho, o objetivo é apoiar o Engenheiro de Software em encontrar código clonado em LPS. Para isso, foi elaborada uma abordagem híbrida de detecção de clones de código em LPS. Para o desenvolvimento e a análise dessa abordagem, foi utilizado o método de pesquisa composto pelas etapas (Figura 1.1):

a) **Etapa 1 - Revisão da Literatura**. Explorar o conteúdo bibliográfico de artigos e outras fontes de informação disponíveis a fim de obter informação de qualidade para fundamentar o estudo realizado nos temas “Clonagem de Código” e “Linha de Produtos de Software”. Essa etapa possui duas atividades:
   - **Realizar Mapeamento Sistemático da Literatura sobre Técnicas de Detecção de Código Clonado**. Foi realizado um estudo exploratório e utilizada a técnica Mapeamento Sistemático da Literatura para abranger o cenário de abordagens de detecção de clones, buscando identificar aquelas destinadas a LPS, bem como técnicas para sistemas de software orientados a objetos e orientados a características;
   - **Realizar Revisão da Literatura sobre LPS**. Foram realizadas pesquisas *ad-hoc* em livros e em artigos científicos para construir a fundamentação teórica sobre o tema “Linha de Produtos de Software”;
b) **Etapa 2 - Abordagem Híbrida**. Elaboração da abordagem híbrida com apoio em abordagens existentes e em técnicas utilizadas. Essa etapa possui duas atividades:
   - **Escolher Técnicas para Detecção de Código Clonado**. Foram escolhidas técnicas de detecção de código clonado a serem utilizadas tendo em vista as técnicas

catalogadas no MSL e em conhecimentos prévios da leitura de outros textos relacionados de interesse;

- **Elaborar a abordagem**. Foi elaborada a abordagem híbrida de detecção de clones, composta pelas técnicas "Sequência de Chamadas de Métodos" e "Análise Estrutural de Assinaturas de Métodos". Essas técnicas são combinadas por meio de duas situações chave que, ao passo em que uma situação é identificada, um passo da técnica é aplicado e o resultado final pode ser obtido;

Figura 1.1 - Método da Pesquisa.



Fonte: Do Autor (2020)

c) **Etapa 3 - Avaliação**. Realizar a coleta de dados necessários para avaliar a abordagem proposta por meio do desenvolvimento de um apoio computacional que (semi) automatiza a abordagem híbrida. Para isso, foram mensurados os valores precisão e *recall* (avaliação interna) e uma análise comparativa com uma ferramenta existente e com propósito similar (avaliação externa). Essa etapa possui quatro atividades

- **Desenvolver um Apoio Computacional.** Desenvolver um apoio computacional que (semi) automatiza a abordagem híbrida de detecção de clones. Foi desenvolvido um *plug-in* para a Eclipse IDE utilizando o *plug-in* FeatureIDE para analisar código fonte de LPS desenvolvidas na linguagem Jakarta de AHEAD (*Algebraic Hierarchical Equations for Application Design*);

- **Escolher LPS para Análise**. Foi escolhida a LPS `Tank War` para avaliar a abordagem proposta por ser amplamente conhecida e fazer parte dos exemplos disponíveis no *plug-in* FeatureIDE;
- **Escolher Ferramentas Computacionais Existentes para Detectar Código Clonado**. Foi escolhida uma ferramenta computacional existente para detectar código clonado. Essa escolha foi feita utilizando o catálogo de ferramentas obtido como parte dos resultados do MSL e artigos/documentos não considerados na MSL (*e.g.*, artigos que não atendiam o propósito da MSL);
- **Realizar Análise Interna e Externa**. Foi realizada a análise da abordagem proposta, utilizando o apoio computacional, em função da análise interna (precisão e *recall*) e externa (comparação com outra ferramenta) para obtenção de resultados que permitam análise comparativa e qualitativa das possíveis melhorias acarretadas pela aplicação real da abordagem proposta. A análise realizada foi (i) **interna**, que avaliou os resultados do apoio computacional por meio de calcular a precisão e o *recall*, e (ii) **externa**, consistiu na comparação com outra ferramenta para possibilitar a análise de cobertura por meio da comparação de resultados obtidos por ambas.

1.4 Organização do Trabalho

Este documento está organizado da seguinte forma.

No Capítulo 2, são mostrados os resultados de um mapeamento sistemático da literatura sobre clonagem de código, dando destaque às ferramentas e às técnicas de detecção de clones encontradas e utilizadas pelos artigos analisados.

No Capítulo 3, uma visão geral sobre os conceitos e a construção de Linhas de Produtos de Software é descrita, abrangendo conhecimentos gerais e específicos sobre LPS.

No Capítulo 4, é descrita a abordagem híbrida proposta por meio do processo fundamentado em quatro etapas que envolvem idealização, escolhas e construção para chegar ao objetivo de detectar clones em LPS por meio de chamadas de métodos e análise estrutural de assinaturas de métodos. Essa abordagem foi implementada a fim de (semi) automatizar a detecção de clones por meio de um *plug-in* para o eclipse e detalhes dessa implementação são descritos nesse capítulo.

No Capítulo 5, a avaliação da abordagem é descrita em termos de avaliação interna e externa cujos objetivos são realizar a análise da ferramenta para mensurar precisão e *recall* e realizar a avaliação do apoio computacional desenvolvido por meio da comparação com outra ferramenta, respectivamente, e apresentar os resultados encontrados em cada uma dessas avaliações.

No Capítulo 6, são listados e explicados resumidamente os trabalhos relacionados encontrados na literatura para dar exemplos de trabalhos que abordam temas similares com este trabalho e cujos objetivos foram detectar clones, explicitando os pontos de diferenciação com este trabalho.

No Capítulo 7, são apresentadas as ameaças a validade interna, externa, de construção, e de conclusão encontradas na realização deste trabalho que podem de alguma forma interferir e/ou influenciar nos resultados e avaliações realizadas.

No Capítulo 8, são descritas as conclusões obtidas na realização deste trabalho bem como suas contribuições. Também são listados possíveis trabalhos futuros observados, como possíveis contribuições após a finalização deste trabalho.

2 CLONAGEM DE CÓDIGO FONTE - ESTADO DA ARTE

2.1 Considerações Iniciais

Com a crescente demanda de sistemas de software para suprir as necessidades geradas pelo avanço da tecnologia, a atenção ao modo como esses sistemas são desenvolvidos é essencial. Clones de código fonte são fragmentos de código replicados e a inserção de clones nesses sistemas ocorre por diversos motivos, entre eles inserção de novas funções, cópia e colagem de código e reutilização de código para determinada finalidade [Solanki; Kumari, 2016]. Nesse sentido, diversas ferramentas e abordagens foram desenvolvidas para detectar clones de código fonte em sistemas de software visando, principalmente, impedir que mudanças inconsistentes possam propagar erros e *bugs*, bem como auxiliar em manutenções necessárias ao bom funcionamento desses sistemas. De modo a explorar a detecção de clones, neste capítulo, é descrito um estudo sistemático da literatura sobre clonagem de código em sistemas de software.

Este capítulo está estruturado da seguinte forma. Na Seção 2.2, são tratados os tipos de clones de código. Na Seção 2.3, estão descritas técnicas de detecção de clones de código. Na Seção 2.4, é apresentado o processo de realização do Mapeamento Sistemático da Literatura (MSL). Na Seção 2.5, são discutidos os resultados obtidos no MSL.

2.2 Tipos de Clones

A prática de "copiar e colar" trechos de código em outras partes do código é chamada de clonagem de código. Clonagem de código é uma prática considerada comum e é introduzida por desenvolvedores principalmente na etapa de manutenção [Fordos; Toth, 2016; Duala-Ekoko; Robillard, 2007]. A detecção de clones de código é essencial para prevenção da propagação de erros, correção de *bugs*, manutenção e gerência de sistemas de software [Petersen *et al.*, 2008; Solanki; Kumari, 2016].

Clones podem ser idênticos ou possuir similaridades que os classificam como tal. A similaridade desses clones pode ser classificada em dois grupos: i) **similaridade sintática ou representacional**; e ii) **similaridade semântica ou comportamental**. Isso possibilitou a adoção da classificação de clones em quatro tipos, sendo o Tipo 1, o Tipo 2 e o Tipo 3 pertencentes ao primeiro grupo e o Tipo 4 pertencente ao segundo grupo [Yuko *et al.*, 2017; Gautam; Saini, 2016; Solanki; Kumari, 2016]. Esses tipos são:

a) **Tipo 1 (*exact clones*)**. Composto por fragmentos de código com estrutura idêntica, possuindo variações de espaço, guias, *layout* e comentários, por exemplo;
b) **Tipo 2 (*renamed/parameterized clones*)**. Fragmentos de código com estrutura/sintaxe similar a outro(s) fragmento(s) de código, acrescidos de alterações em identificadores, literais, tipos, *layout* e comentários, por exemplo;
c) **Tipo 3 (*near-miss clones*)**. Fragmentos de código acrescidos de declarações, inserções/eliminações e alterações em identificadores, literais, tipos e *layout*, por exemplo;
d) **Tipo 4 (*semantic clones*)**. Fragmentos de código funcionalmente semelhantes, mas não possuem semelhança textual. Ou seja, são funções do código original implementadas com sintaxe diferente.

À medida que o código aumenta de tamanho, a frequência com que a clonagem pode ocorrer pode aumentar e, consequentemente, aumentar o risco de falhas e de erros visto que o controle de novos trechos precisa ser mais rigoroso. Por outro lado, estudos mostram que nem sempre o tamanho do código influencia na ocorrência de clones [Torres *et al.*, 2017] e, em sua maioria, tais ocorrências são relatadas em etapas de manutenções corretivas ou evolutivas de código. Como exemplo desses tipos de clonagem, pode-se analisar um trecho de código hipotético (Figura 2.1) e 4 trechos de código clonados, que apresentam modificações que caracterizam esses tipos.

Figura 2.1 - Trecho Original.

```
1     ResultSet res = sist.executeQuery(query);
2     boolean ok;
3     for(ok = rs.first(); ok; ok = res.next()){
4     results.add(new Project(res.getInt(2)));
5     }
6     return results;
```

Fonte: Do Autor (2020)

No trecho do código da Figura 2.2, nas linhas 1 a 4, estão realçadas (cor de fundo diferente) informações que caracterizam o Tipo 1 de clonagem, sendo espaçamento (linha 1), adição de comentário (linha 2), espaçamento (linha 3) e *layout* (linha 4). No código da Figura 2.3, as linhas 1, 2, 3, 5 e 7 possuem alterações em identificadores de variáveis e, na linha 4, há alteração no *layout*. Essas alterações caracterizam clonagem de código do Tipo 2. No código da Figura 2.4, o Tipo 3 de clonagem pode ser identificado na linha 4, onde ocorre a eliminação da chamada do método `res.getInt(2)`. No código da Figura 2.5, pode ser verificada a mudança semântica no contexto de execução das instruções, onde o comando de repetição `for` foi substituído pelo comando de repetição `do ... while`, sem alterações no resultado final.

Figura 2.2 - Tipo 1 de Clonagem.

```
ResultSet res=sist.executeQuery(query);
boolean ok; //Controlador
for(ok = rs.first(); ok; ok=res.next()){
results.add(new Project(res.getInt(2))); }
return results;
```

Fonte: Do Autor (2020)

Figura 2.3 - Tipo 2 de Clonagem.

```
ResultSet r = sist.executeQuery(query);
boolean ver;
for(ver = r.first(); ver; ver = r.next())
{
ok.add(new Project(res.getInt(2)));
}
return ok;
```

Fonte: Do Autor (2020)

Figura 2.4 - Tipo 3 de Clonagem.

```
ResultSet res = sist.executeQuery(query);
boolean ok;
for(ok = rs.first(); ok; ok = res.next()){
results.add(new Project( );
}
return results;
```

Fonte: Do Autor (2020)

Figura 2.5 - Tipo 4 de Clonagem.

```
ResultSet res = sist.executeQuery(query);
boolean ok;
if(!res.first());
return results;
do{
results.add(new Project(res.getInt(2)));
} while(res.next());
return results;
```

Fonte: Do Autor (2020)

2.3 Técnicas de Detecção

Existem ferramentas desenvolvidas cujo propósito é detectar clones em código. Essas ferramentas diferem-se, principalmente, em como fragmentos de código são representados para serem comparados a outros fragmentos de código. Em geral, essa representação é por meio de *String*, *Token*, AST (*Abstract Syntax Tree*), PDG (*Program Dependence Graph*), Medidas e formas híbridas para chegar a uma abordagem final [Schugerl, 2011]. Tal detecção pode ser feita analisando blocos de código, métodos, classes ou outra medida que determine, por exemplo, o tamanho do fragmento a ser analisado.

Para a classificação dos tipos de clonagem ser feita de maneira adequada, técnicas de detecção foram elaboradas, nas quais a variação da representação do trecho de código a ser analisado é a principal diferença entre as técnicas mais comumente utilizadas [Rattan; Kaur, 2016; Jang; Brumley, 2009]. Essas técnicas de representação são descritas como:

a) **Baseada em texto (*text-based*)**. Técnica que consiste na combinação de textos e de *strings* para encontrar candidatos a clones que se diferem por meio da existência ou não de comentários e no *layout* do(s) fragmento(s) de código a ser(em) analisado(s). É basicamente a busca por trechos idênticos em diferentes partes do código.
b) **Baseada em símbolos (*token-based*)**. O processo de análise léxica tem como resultado a produção de uma sequência de *tokens* (símbolos). Esses *tokens* são utilizados como parâmetros para métodos de busca para encontrar possíveis clones.
c) **Baseada em árvore (*tree-based*)**. *Abstract Syntax Tree* (AST) fornece uma abstração da análise sintática em forma de árvore do código, por meio de visitas efetuadas, e a constrói com base nos parâmetros visitados. A AST é percorrida por subárvores semelhantes até encontrar possíveis clones sintáticos contidos nessas subárvores, utilizando alguma técnica de correspondência de árvores.
d) **Baseada em grafo (*graph-based*)**. *Graphy Dependence Program* (PDG) é um grafo direcionado e funciona como abstração semântica. Com a utilização do isomorfismo de um subgráfico obtido de um sistema de software, é possível encontrar subgrafos semelhantes sendo classificados como clones, e preservar a semântica original do código.
e) **Baseada em híbrido (*hybrid-based*)**. Essa técnica consiste em combinar diversas técnicas, inclusive as anteriores, cujo objetivo é criar/melhorar a detecção dos tipos de clone existentes. Por meio da sua utilização, nota-se que melhorias podem ser obtidas no modo em que essas detecções são efetuadas, a fim de melhorar aspectos de precisão, de desempenho e de qualidade. Logo, visando a esses aspectos de melhoria, técnicas híbridas são construídas e testadas para melhorar tal escopo.

Tais técnicas são implementadas de diversas formas. Para cada contexto de implementação, a(s) medida(s) a ser(em) utilizada(s) para comparação depende(m) da finalidade de utilização por meio dos programadores. Tamanho de fragmento, escopo de detecção, granularidade e escalabilidade são exemplos de medidas comumente utilizadas na implementação de ferramentas/abordagens desenvolvidas. Essas ferramentas detectam clones em várias linguagens e paradigmas, por exemplo, a ferramenta `CCFinder` [Kamiya *et al.*, 2002] é amplamente utilizada e pesquisada no contexto de detecção de clones por causa de suas

características de detecção eficientes, quantidade de tipos de clones possíveis de detecção e o bom desempenho da ferramenta na detecção de clones em códigos com grande quantidade de linhas de código. Em sua maioria, as ferramentas são implementadas para detecção de clones em sistemas de *software* desenvolvidos nas linguagens orientadas a objetos (por exemplo, JAVA e C++) [Kamiya *et al.*, 2002; Lin *et al.*, 2014; Yuan; Guo, 2011) ou para o paradigma procedural (por exemplo, C). Entretanto, a detecção de clones não se limita a tais paradigmas/linguagens.

Diante da variedade de ferramentas, de técnicas, de métodos e de outras formas para detectar código clonado em sistemas de software, foi realizado um estudo exploratório, cujo resultado final obtido foi a identificação de 158 artigos científicos que apresentam pesquisas sobre clonagem de código. Neste estudo, foi utilizada a técnica Mapeamento Sistemático da Literatura (MSL), tendo em vista que os requisitos de pesquisa são menos rigorosos em relação à técnica Revisão Sistemática da Literatura (RSL), sendo o seu interesse em tendências de pesquisa [Kitchenham; Brereton, 2010]. Em MSL, não há realização da avaliação de qualidade, pois o seu resultado é um inventário de artigos sobre a área temática, mapeados para uma classificação [Wieringa *et al.*, 2005], sendo uma visão geral do escopo da área que permite descobrir lacunas e tendências de pesquisa [Petersen *et al.*, 2008].

2.4 Planejamento do Estudo Exploratório

Nesta seção, é descrito como o estudo exploratório, utilizando MSL, foi planejado, incluindo a descrição das questões de pesquisa, o escopo de pesquisa, a estratégia de busca e os critérios de classificação dos artigos selecionados.

2.4.1 Questões de Pesquisa

O objetivo foi realizar o estudo exploratório de pesquisas existentes, que abordam a detecção de códigos clonados em sistemas de software. É importante considerar a abrangência de informações que tal tema retorna e possibilitar que, com a definição e o estabelecimento do tipo de informações coletadas, haja o fornecimento às pesquisas futuras do estado da arte acerca da clonagem de código. Para tanto, foram coletadas informações sobre técnicas, processos, métodos, procedimentos, metodologias e ferramentas disponíveis na literatura que detectam código clonado em sistemas de software. Assim, foram elaboradas e respondidas as seguintes questões de pesquisa (QP):

QP1: Como é realizada a detecção de clones em código de sistemas de software?

Justificativa: Mostrar o cenário de como são detectados clones em código fonte de sistemas de software bem como coletar informações comuns às técnicas, aos processos, aos métodos, aos procedimentos e às metodologias identificadas. Além disso, identificar qual(is) ferramenta(s) é(são) utilizada(s) para identificar a existência de clones por parte dos programadores.

QP2: Quais tipos de clones são abordados nos artigos analisados?

Justificativa: Identificar quais tipos de clonagem são identificados pelos autores e se existe uma convenção da utilização desse termo, com base nos resultados obtidos.

QP3: Quais linguagens e paradigmas de programação são utilizados?

Justificativa: Identificar as linguagens de programação comumente utilizadas para identificar clones e, com base nisso, quais paradigmas são predominantemente adotados pelos autores para realizar a detecção de clones em código.

2.4.2 Estratégia de Pesquisa

Após estabelecer as questões de pesquisa (QP), foram definidas palavras/termos relevantes para obter resultados importantes e objetivos para o estudo exploratório. Para isso, foi utilizado o *framework* PICo[1] [Manual, 2014; Murdoch University, 2020] (Tabela 2.1):

a) ***Population*** **- População (P)**, trata-se do objeto de estudo em questão. Nesse MSL, a população é "código clonado";
b) ***Interest*** **- Interesse (I)**, refere-se a um evento, uma atividade, uma experiência ou um processo definido. Nesse MSL, o interesse é "técnicas", "processos", "métodos", "procedimentos", "metodologias" e "ferramentas";
c) ***Context*** **- Contexto (Co)**, consiste no cenário ou nas características distintas relacionados ao objeto de estudo. Nesse MSL, o contexto é "software".

Com os operadores lógicos `OR` e `AND`, juntamente com as palavras-chave e respectivos sinônimos, foram aplicados nos elementos do framework PICo. Assim, a seguinte *string* de busca foi elaborada:

[1] Foi desenvolvido para identificar palavras-chave e formular *strings* de busca [Manual, 2014; Murdoch University, 2020].

("code clones" OR "cloned code") AND (technique OR process OR method OR procedure OR methodology OR tool) AND (detect OR detection) AND software

Tabela 2.1 - Estrutura para Caracterizar a *String* de Busca Utilizando *Framework* PICo.

PICo	Termos
P	"code clones" OR "cloned code"
I	"technique" OR "process" OR "method" OR "procedure" OR "methodology" OR "tool"
Co	"software"

Fonte: Do autor (2020)

Para responder às questões de pesquisa utilizando a *string* de busca, foram selecionadas as bibliotecas digitais de artigos científicos ACM[2], Ei Compendex[3], IEEE Xplorer[4], Science Direct[5], Scopus[6] e Springer Link[7]. Essas bibliotecas foram escolhidas por suportar (i) pesquisa avançada com utilização de palavras-chave, (ii) filtragem dos resultados por ano e área de publicação, (iii) filtragem por tipo de publicação e (iv) exportação do resultado da consulta em formato `BibTex` ou `Endnote`. O retorno inicial da pesquisa em cada biblioteca digital foi armazenado em bases de dados.

2.4.3 Critério de Inclusão/Exclusão

Na Tabela 2.2, é apresentada a quantidade de trabalhos resultantes nas bibliotecas digitais (Figura 2.6[8]). O primeiro critério é o resultado obtido (**Resultado Inicial**) com a utilização da *string* de busca, que foi armazenado em bases de dados diferentes para cada biblioteca (1.485 trabalhos). Em cada base de dados, foram removidos os trabalhos caracterizados como não artigos (por exemplo, livros, normas e *table of contents*) (**Apenas Artigos**) (1.101 artigos), caracterizando o segundo critério. Após a remoção, os artigos foram agrupados em uma única base de dados para eliminar artigos duplicados e artigos que continham conteúdo que não agregavam conteúdo para a pesquisa (**Resultado Final**), sendo o terceiro critério. Nessa eliminação, foi considerada a quantidade de palavras-chave, ou seja, o artigo duplicado e armazenado na biblioteca digital de artigos científicos com menor quantidade de palavras-chave foi removido[9]. Os artigos restantes (158 artigos) foram lidos na íntegra de modo que o foco da leitura foi direcionado para responder as questões de pesquisa.

[2] www.acm.org
[3] www.engineeringvillage.com
[4] www.ieeexplore.ieee.org
[5] www.sciencedirect.com
[6] http://www.scopus.com
[7] www.link.springer.com
[8] Ressalta-se a alteração do termo "trabalho" para "artigo", pois "trabalho" = "artigo" + "não artigo", nesse contexto.
[9] Justificativa: Com mais quantidade de palavras-chave, há melhor caracterização do artigo.

Cabe ressaltar que três pesquisadores (Pesquisador A, Pesquisador B e Pesquisador C) foram envolvidos na obtenção dos artigos e foi realizado o seguinte procedimento:

a) O Pesquisador A executou a *string* de busca nas bibliotecas digitais selecionadas e documentou os resultados no sistema de software JabRef[10];
b) O Pesquisador A verificou e excluiu os trabalhos que não eram artigos e os artigos repetidos (com título, autores e resumo iguais). Na identificação de artigos repetidos, foram mantidos os artigos com palavras-chave que melhor descreviam o artigo;
c) Os artigos encontrados foram avaliados pelo Pesquisador A e pelo Pesquisador B, de maneira individual e separada, quanto ao atendimento aos critérios de inclusão e de exclusão. Essa avaliação foi realizada por meio da leitura do título, do resumo e das palavras-chave. Os artigos, cujas avaliações causaram dúvidas quanto a sua inclusão/exclusão por parte dos pesquisadores, foram incluídos. Os artigos foram documentados em uma lista de artigos incluídos e excluídos com justificativa para sua inclusão ou exclusão;
d) Realizou-se a interseção entre os artigos selecionados pelo Pesquisador A e pelo Pesquisador B, sendo esses artigos documentados (Interseção). Na ocorrência de algum desacordo sobre a inclusão ou a exclusão de artigos, os dois pesquisadores discutiram e resolveram. Em casos que não houve consenso, o artigo foi incluído. Os artigos excluídos foram documentados em uma lista de artigos excluídos com justificativa para sua exclusão;
e) O Pesquisador C avaliou os artigos excluídos e as justificativas de exclusão e os artigos presentes na Interseção. O resultado foi o conjunto de artigos resultantes do estudo exploratório.

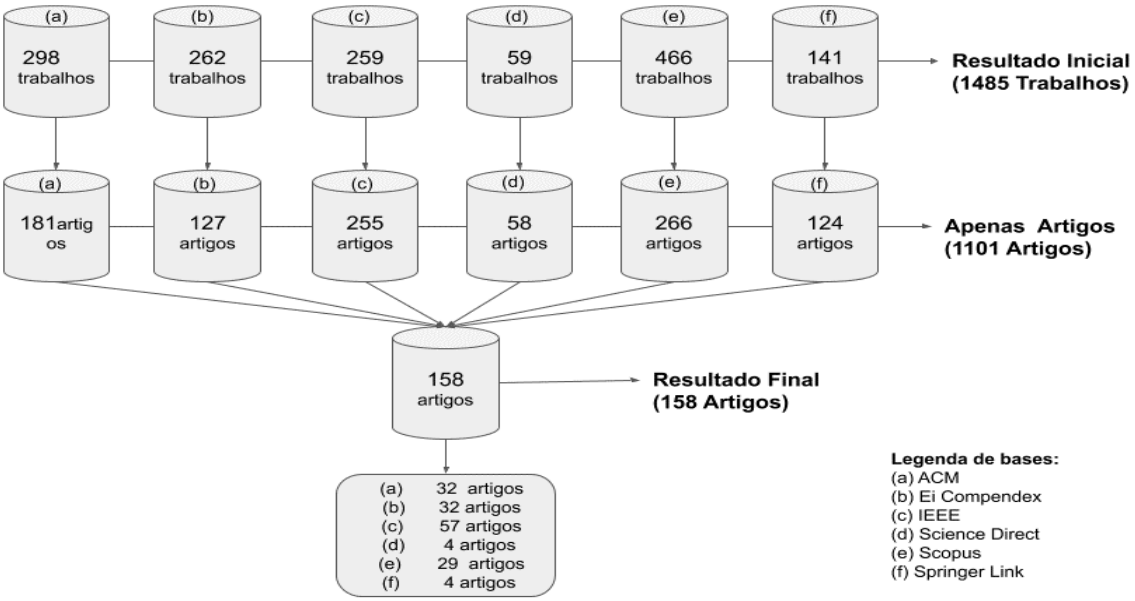
Tabela 2.2 - Quantidade de Artigos.

Bases	Bibliotecas Digitais de Artigos Científicos	Resultado Inicial (Trabalhos)	Apenas Artigos	Resultado Final (Artigos)
a	ACM	298	181	32
b	Ei Compendex	262	127	32
c	IEEE	259	255	57
d	Science Direct	59	58	4
e	Scopus	466	266	29
f	Springer Link	141	124	4
	Total	**1.485**	**1.101**	**158**

Fonte: Do Autor (2020)

[10] http://www.jabref.org/

Figura 2.6 - Processo de Obtenção dos Resultados por Bibliotecas Digitais.



Fonte: Do Autor (2020)

2.5 Resultados

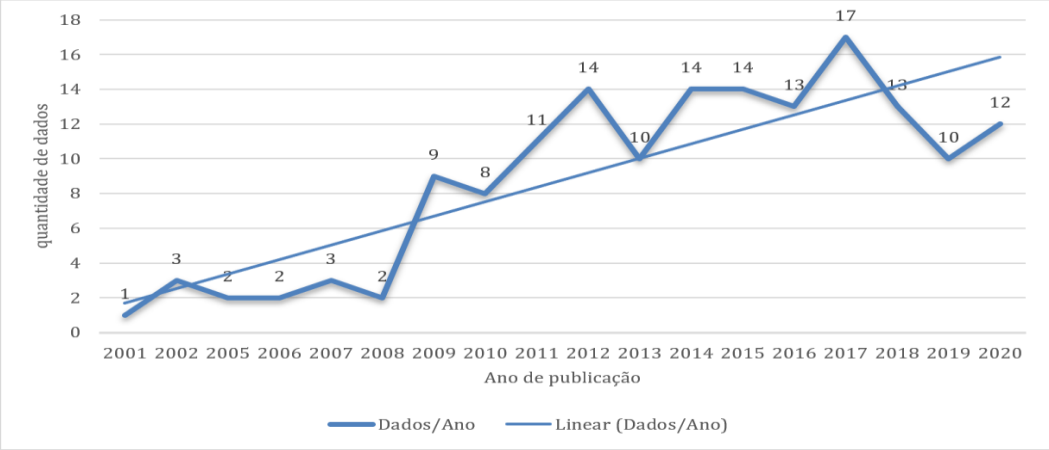
A finalização dos procedimentos e da seleção das informações necessárias permite ter noção da diversidade de ferramentas e de maneiras em comum de detecção de clones nas abordagens que propõe técnicas, processos, métodos, procedimentos e metodologias acerca do estado da arte da detecção de clones em sistemas de software. A detecção de clones é abordada de diversas maneiras e por várias ferramentas que implementam essas abordagens, ambas identificadas nos 158 artigos (Tabela A.1 no APÊNDICE A) após aplicar os critérios de inclusão/exclusão.

2.5.1 Características Gerais dos Artigos Selecionados

Como mencionado anteriormente, foram utilizadas 6 bibliotecas digitais de artigos científicos para a realização do estudo exploratório. Na Tabela 2.2, é apresentada a quantidade final de 158 artigos como resultado para análise, sendo 32 artigos (20,25%) na ACM, 32 artigos (20,25%) na Ei Compendex, 57 artigos (36,07%) na IEEE, 4 artigos (2,53%) na Science Direct, 29 artigos (18,35%) na Scopus e 4 artigos (2,53%) na Springer Link. Foi considerado o período que engloba, aproximadamente, 20 anos de publicações (2001 a 2020, inclusive e incompleto). A quantidade de artigos publicados acerca do tema “Detecção de Código Clonado”, aparentemente, tende a crescer (Figura 2.7) com o decorrer dos anos. A maior concentração dessas publicações foi encontrada no ano de 2017, com o total de 17 artigos.

Sobre a quantidade de pesquisadores, foram identificados 384 pesquisadores no tema "Detecção de Clonagem de Código", sendo os mais atuantes os pesquisadores S. Kusumoto com 7 artigos, C. K. Roy com 6 artigos, Y. Higo com 5 artigos e R. K. Tekchandani com 5 artigos. Cabe ressaltar a presença de dois grupos de pesquisadores que têm publicado juntos. Um desses grupos é composto por cinco pesquisadores Nguyen, H. A., Nguyen, T. N., Nguyen, T. T., Pham, N. H. e Al-Kofahi, J. M. com 4 artigos (A17, A33 e A125). Outro grupo é composto por dois pesquisadores Kanmani, S. e Kodhai, E. com 4 artigos (A92, A105, A106 e A108).

Figura 2.7 - Quantidade nos Anos.



Fonte: Do Autor (2020)

Os 158 artigos selecionados no estudo exploratório foram publicados em 84 veículos distintos de divulgação científica, sejam *journals*, *workshops*, conferências e simpósios. Os quatro veículos com mais quantidade de artigos são ACM Symposium Application Computing (ACMSAC) com 5 artigos publicados (5,85%), Asia-Pacific Software Engineering Conference (APSEC) com 5 artigos publicados (5,85%), International Conference on Software Engineering (ICSE) com 7 artigos publicados (8,33%) e International Workshop on Software Clones (IWSC) com 10 artigos publicados (11,90%) (Tabela 2.3). Essa divergência de valores pode ser justificada pelo fato de IWSC ser um workshop específico para abordar assuntos relacionados a clonagem de código, a ICSE ser o principal evento da área de Engenharia de Software (segundo a ICSE[11]) e a APSEC e ACMSAC serem eventos importantes na área de Engenharia de Software.

[11] http://www.icse-conferences.org/

Tabela 2.3 - Principais Veículos de Publicação.

Veículos de Divulgação	Quant.	Artigos
International Workshop on Software Clones (IWSC)	10	A20, A27, A44, A73, A82, A87, A96, A98, A147, A152
International Conference on Software Engineering (ICSE)	7	A11, A13, A33, A43, A65, A80, A131
Asia-Pacific Software Engineering Conference (APSEC)	5	A7, A18, A28, A97, A100
ACM Symposium Application Computing (ACMSAC)	5	A19, A26, A45, A49, A78

Fonte: Do autor (2020)

2.5.2 Resposta às Questões de Pesquisa

Em resposta a

QP1: Como é realizada a detecção de clones em código de sistemas de software?

a detecção de clones é feita por meio de abordagens e de ferramentas que as implementam. Foram identificadas 52 ferramentas (Tabela 2.4). Cada ferramenta utiliza uma representação de fragmentos a serem analisados/considerados clone, sendo *Token* (16 ferramentas - 30,8%), Árvore (ou AST) (12 ferramentas - 23,1%) e Grafo (8 ferramentas - 15,4%) as representações que possuem mais quantidade de ferramentas. Essa informação foi anteriormente ressaltada (Seção 2.3) e constatada com os valores identificados. Cabe ressaltar que algumas ferramentas possuem mais de uma referência (`ClemanX` - A33 e A125; `XIAO` - A9 e A27 e `CCFinder` - A7, A13 e A14), justificada pelo fato que alguns autores publicam, inicialmente, a abordagem e, depois, algumas melhorias/implementações na ferramenta proposta.

Tabela 2.4 - Ferramentas Identificadas.

Ferramenta	Baseada em	Referência
CCCD	Análise Concólica	A72
CLORIFI	Análise Concólica	A57
Asta	Árvore	A23
CCLearner	Árvore	A79
CCR	Árvore	A99
ClemanX	Árvore	A17
Clever	Árvore	A35
CReN	Árvore	A42
Deckard	Árvore	A11
KLON	Árvore	A121
LICCA	Árvore	A4
RefactoreRL	Árvore	A58
MultiDup	Árvore e Medidas	A1
CloneDR	AST	A75
SHAPE	Extração de Procedimento Amorfo	A22
CCSharp	Grafo	A38
ClemanX	Grafo	A33, A125
CloneWorks	Grafo	A25
CMCD	Grafo	A18
DyCLINK	Grafo	A36

(Continua)

Tabela 2.4 - Ferramentas Identificadas. (cont.)

Ferramenta	Baseada em	Referência
GemScan	Grafo	A125
SCVD	Grafo	A50
Scopio	Grafo e Híbrida	A81
DebCheck	Híbrida	A21
PC Detector	Híbrida	A95
SynTex	Híbrida	A44
Hanni	Macros	A30
HeapAbsCC	Manipulação de *Heap*	A60
gCad	Mapeamento de Função	A103
Clone Miner	Medidas	A94
SDD	Medidas	A6
XIAO	Medidas	A9, A27
JCC	*Pipeline*	A111
CCDemon	Recomendação Interativa	A55
BinClone	*Token*	A89
BOREAS	*Token*	A70
CCFinder	*Token*	A7, A13, A14
CCFinderSW	*Token*	A97
Ctcompare	*Token*	A24
FRISC	*Token*	A63
IDCCD	*Token*	A100
LSC Miner	*Token*	A124
ReDeBug	*Token*	A78
RTF	*Token*	A41
SaCD	*Token*	A62
ScalClone	*Token*	A53
SHINOBI	*Token*	A66
SourcererCC	*Token*	A90
SourcererCC-1	*Token*	A90
VUDDY	*Token*	A49
Simone	Não Informado	A123
VFDTECT	Não Informado	A88
CLCDSA	chamada de API	A129
TBCCD	hibrida	A130
CCAligner	token	A131
ICDT	token	A132
DSCCD	hibrida	A135
JGibbLD	Modelagem de tópico	A136
CCDLC	híbrida	A138
Siamese	híbrida	A139
IBFET	índice	A141
TECCD	árvore	A142
DroidSD	híbrida	A143
SAGA	matriz de sufixo	A144
Twin-Finder	machine learn	A145
Vincent	Imagem	A147
ICCV	Imagem	A150
CPPCD	token	A152
CroLSim	híbrida	A156
LVMapper	híbrida	A157
Total: 70 Ferramentas		

Fonte: Do autor (2020)

Foram identificadas várias formas de detectar clones. Tais formas foram agrupadas em 30 técnicas (Tabela 2.5) baseadas na representação do código para detectar clones, sendo a técnica baseada em Árvore (AST e Distância) (29 artigos - 18,35%) a que possui mais quantidade de artigos. Esses valores reforçam que as técnicas mais utilizadas são as descritas

na Seção 2.3. Ao todo, 154 artigos (97,46%) apresentaram a detecção de clones baseadas nos tipos listados na tabela e 4 artigos (A91, A116, A119 e A123 - 2,54%) não apresentaram. Mesmo que possuam algum tipo de interface implementada, algoritmos e outros, 83 artigos (52,53%) não possuem nome para as abordagens descritas pelos autores para referenciá-las. Por esse motivo, os resultados são em relação ao tipo de representação (baseada em) utilizada para detecção de clones. Não foi identificada a técnica abordada em três artigos (A12, A88 e A123).

Tabela 2.5 - Técnicas Detecção de Código Clonado.

Técnica Baseada em	Métodos Utilizados	Quant.	Referência
Análise Concólica	Teste Simbólico	3	A26, A57, A72
Análise de Crescimento	Análise de Crescimento	1	A59
Aprendizado de Máquina	Aprendizado de Máquina	2	A104, A145
Árvore	AST e Distância	29	A1, A4, A11, A17, A23, A28, A31, A34, A35, A37, A39, A42, A45, A47, A48, A58, A61, A67, A73, A75, A79, A83, A93, A99, A112, A121, A125, A126, A142
Chave de Redução	Elemento Principal de Redução	1	A117
Controle de *Statements*	Controle de Fluxo de *Statements*	1	A113
Fechamento Transitivo	Fecho Transitivo	1	A52
Grafo	PDG, Vetores e Distância	14	A18, A25, A33, A36, A38, A43, A50, A54, A64, A76, A80, A81, A102, A125
Híbrida	AST, PDG, Medidas e combinação de métodos	26	A2, A8, A9, A10, A16, A19, A21, A44, A46, A74, A95, A98, A101, A106, A107, A108, A110, A115, A127, A130, A135, A138, A139, A143, A156, A157
Índice	Índice de Referência	2	A118, A141
Limiar	Limite	1	A68
Macro	Macros	1	A30
Manipulação de *Heap*	*Heap*	1	A60
Mapeamento de Função	Mapear Funções em Estruturas de Dados	1	A103
Métodos Formais	Formalismo Matemático	2	A65, A96
Métrica	Tamanho, Complexidade, Escalabilidade, etc	19	A1, A3, A6, A15, A27, A29, A40, A56, A77, A82, A87, A92, A94, A105, A109, A114, A115, A122, A128
Mineração de Conjunto	Itens Frequentemente Ponderados	1	A120
PALEX	PALEX	1	A69
Pipeline	Ações Paralelas	1	A111
Procedimento Amorfo	Procedimentos Amorfos	1	A22
Processo Hierárquico Analítico	Tomada de Decisões Complexas	1	A51
Recomendação Interativa	Recomendação Interativa	1	A55
Semântica Baseada na Web	Web Semântica	5	A5, A20, A84, A85, A86
Texto	Verificação de *String*	1	A92
Token	Comparação de *tokens* e distância	22	A7, A13, A14, A24, A32, A41, A49, A53, A62, A63, A66, A70, A78, A88, A89, A90, A97, A100, A124, A131, A132, A152

(Continua)

Tabela 2.5 - Técnicas Detecção de Código Clonado. (cont.)

Técnica Baseada em	Métodos Utilizados	Quant.	Referência
Chave de Redução	Elemento Principal de Redução	1	A117
Controle de *Statements*	Controle de Fluxo de *Statements*	1	A113
Fechamento Transitivo	Fecho Transitivo	1	A52
Grafo	PDG, Vetores e Distância	14	A18, A25, A33, A36, A38, A43, A50, A54, A64, A76, A80, A81, A102, A125
Híbrida	AST, PDG, Medidas e combinação de métodos	26	A2, A8, A9, A10, A16, A19, A21, A44, A46, A74, A95, A98, A101, A106, A107, A108, A110, A115, A127, A130, A135, A138, A139, A143, A156, A157
Índice	Índice de Referência	2	A118, A141
Limiar	Limite	1	A68
Macro	Macros	1	A30
Manipulação de *Heap*	*Heap*	1	A60
Mapeamento de Função	Mapear Funções em Estruturas de Dados	1	A103
Métodos Formais	Formalismo Matemático	2	A65, A96
Métrica	Tamanho, Complexidade, Escalabilidade, etc	19	A1, A3, A6, A15, A27, A29, A40, A56, A77, A82, A87, A92, A94, A105, A109, A114, A115, A122, A128
Mineração de Conjunto	Itens Frequentemente Ponderados	1	A120
PALEX	PALEX	1	A69
Pipeline	Ações Paralelas	1	A111
Procedimento Amorfo	Procedimentos Amorfos	1	A22
Processo Hierárquico Analítico	Tomada de Decisões Complexas	1	A51
Recomendação Interativa	Recomendação Interativa	1	A55
Semântica Baseada na Web	Web Semântica	5	A5, A20, A84, A85, A86
Texto	Verificação de *String*	1	A92
Token	Comparação de *tokens* e distância	22	A7, A13, A14, A24, A32, A41, A49, A53, A62, A63, A66, A70, A78, A88, A89, A90, A97, A100, A124, A131, A132, A152
Wavelets	Decompor e Descrever ou Representar outra Função/Dados	1	A71
Chamada de API	Chamada a API	1	A129
Imagem	Semelhança de imagens	2	A147, A150
Modelos	Modelagem de tópicos	1	A136
Matriz	Matrix de sufixo	1	A144
Total: 30 Técnicas			

Fonte: Do autor (2020)

Em resposta a

QP2: Quais tipos de clones são abordados nos artigos analisados?

quanto aos tipos de clones, todos foram listados e relacionados na Tabela 2.6. Quando o tipo não estava explicitamente identificado, havia referência somente à detecção de clones semânticos ou sintáticos. Foram identificadas 193 ocorrências de abordagens de detecção de clones, sendo 48 artigos (24,87%) detectam clones do Tipo 1, 47 artigos (24,35%) detectam

clones do Tipo 2, 57 artigos (29,53%) detectam clones do Tipo 3, 12 artigos (6,21%) detectam clones do Tipo 4, 42 artigos (21,76%) detectam clones semânticos e 16 artigos (8,29%) detectam clones sintáticos. A diferença constatada em relação a quantidade inicial de 158 artigos do MSL, deve-se ao fato de que algumas abordagens detectam mais de um tipo de clone e/ou são classificados em clones semânticos/sintáticos. Cabe ressaltar que há repetições na numeração do artigo em diferentes tipos, justificados pelo fato das informações coletadas serem informadas explicitamente pelo autor, não ficando a critério de interpretação durante a análise.

Tabela 2.6 - Tipos de Clones Identificados.

Tipo	Quant.	Referência
1	48	A6, A8, A14, A16, A18, A24, A26, A30, A32, A34, A38, A39, A45, A46, A49, A59, A62, A71, A77, A79, A81, A82, A92, A96, A97, A98, A99, A103, A105, A106, A107, A108, A109, A113, A118, A120, A131, A132, A137, A139, A140, A141, A143, A144, A150, A152, A157
2	47	A6, A8, A14, A18, A24, A26, A31, A34, A38, A39, A43, A45, A46, A49, A53, A59, A62, A71, A77, A79, A81, A82, A83, A91, A92, A96, A97, A98, A99, A103, A105, A106, A107, A108, A113, A118, A120, A131, A132, A139, A141, A143, A144, A147, A150, A152, A157
3	57	A1, A6, A8, A2, A8, A14, A16, A18, A19, A22, A25, A26, A29, A34, A37, A38, A39, A41, A42, A43, A45, A46, A47, A63, A68, A70, A71, A73, A74, A75, A79, A81, A82, A85, A90, A94, A97, A99, A03, A105, A106, A107, A113, A117, A120, A131, A132, A136, A137, A139, A141, A142, A143, A144, A147, A150, A152, A157
4	12	A2, A14, A20, A26, A38, A39, A46, A72, A74, A79, A97, A113, A140,
Semântico	42	A4, A11, A28, A80, A84, A86, A87, A93, A95, A102, A112, A119, A124, A126, A129, A130, A131, A132, A133, A134, A135, A136, A137, A138, A139, A140, A141, A142, A143, A144, A145, A146, A147, A148, A149, A150, A151, A152, A153, A154, A155, A156
Sintático	16	A10, A60, A69, A78, A83, A100, A101, A110, A115, A116, A121, A127, A138, A146, A153, A155

Fonte: Do autor (2020)

Em resposta a

QP3: Quais linguagens e paradigmas de programação são utilizados?

foram identificadas 13 linguagens de programação (Tabela 2.7). A quantidade de vezes em que elas apareceram nos artigos está distribuída em C (44 artigos - 24,71%), C# (12 artigos - 6,74%), C++ (23 artigos - 12,92%), Cobol (3 artigos - 1,68%), Erlang (3 artigos - 1,68%), Fortran (1 artigo - 0,56%), Haskell (1 artigo - 0, 56%), Java (82 artigos - 46,06%), JavaScript (2 artigo - 1,12%), Modula2 (1 artigo - 0,56%), Python (1 artigo - 0,56%), Ruby (2 artigos - 1,12%) e Scheme (1 artigo - 0,56%).

Tabela 2.7 - Linguagens de Programação.

Linguagem de Programação	Quant.	Referência
C	44	A3, A4, A7, A9, A11, A15, A19, A21, A22, A25, A26, A27, A29, A31, A34, A37, A38, A40, A45, A49, A50, A52, A60, A62, A63, A66, A69, A75, A76, A77, A78, A80, A81, A83, A85, A92, A104, A106, A108, A113, A116, A118, A121, A141
C#	12	A1, A9, A19, A23, A24, A27, A29, A31, 45, A59, A91, A105

(Continua)

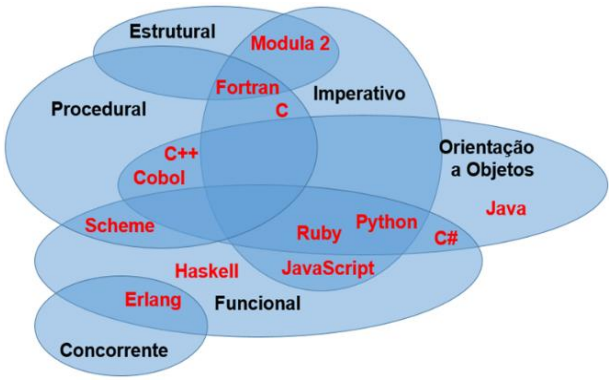
Tabela 2.7 - Linguagens de Programação. (cont.)

Linguagem de Programação	Quant.	Referência
C++	23	A3, A7, A8, A9, A15, A16, A27, A31, A40, A49, A50, A52, A62, A66, A57, A78, A80, A81, A94, A110, A113, A141, A146
Cobol	3	A7, A15, A30
Erlang	3	A46, A58, A61
Fortran	1	A121
Haskell	1	A48
Scheme	1	A4
Modula 2	1	A4
JavaScript	2	A4, A141
Ruby	2	A69, A91
Python	3	A73, A149, A150
Java	82	A2, A4, A7, A8, A10, A11, A15, A18, A19, A23, A25, A26, A28, A29, A32, A33, A35, A36, A37, A39, A42, A43, A44, A45, A47, A52, A55, A57, A62, A64, A65, A68, A69, A70, A71, A74, A75, A79, A82, A84, A85, A91, A92, A94, A96, A98, A99, A100, A101, A106, A107, A108, A109, A110, A111, A112, A113, A117, A118, A119, A120, A126, A131, A132, A133, A134, A138, A139, A140, A141, A142, A143, A144, A147, A149, A151, A152, A153, A154, A155, A157, A158
	Total: 13 Linguagens de Programação	

Fonte: Do autor (2020)

Essas linguagens pertencem a 6 paradigmas de programação (Figura 2.8) (estrutural, imperativo, funcional, orientado a objetos, procedural e concorrente). Cabe ressaltar que existem repetições na numeração do artigo em diferentes tipos, justificados pelo fato das informações coletadas serem informadas explicitamente pelo autor, não ficando a critério de interpretação durante a análise.

Figura 2.8 - Paradigmas de Programação Identificados.



Fonte: Do Autor (2020)

2.5.3 Discussão

Clones de código são detectados por abordagens e ferramentas, sendo implementadas de maneiras diferentes. Porém, é possível identificar algumas maneiras de compará-los, por exemplo, o modo em que o código a ser dito “clone” é representado para identificação e os tipos de clones identificados. Os artigos identificados no estudo exploratório possuem intervalo

de publicação de 20 anos (2001 a 2020, inclusive e incompleto). Quanto à quantidade de veículos de divulgação científica identificados (84 *journals*, *workshops*, conferências e simpósios), pode ser considerada alta e mostra a diversidade de contextos/áreas da Computação, nos quais são abordadas a detecção de clones, por exemplo, segurança, inteligência artificial, engenharia reversa, *data mining* e computação aplicada.

Pode-se observar que a maior concentração de artigos foi apresentada no `International Workshop on Software Clones (IWSC)`, cujo objetivo é "reunir pesquisadores e profissionais para avaliar o estado atual da pesquisa, discutir problemas comuns e direções emergentes (como a detecção de clones em modelos de software, análise de clones em reengenharia para reutilização, análise de clones em evolução de software e detecção de clones em direitos autorais e plágio)". Foram reunidas abordagens e suas bases de representação similares em tipos diferentes. Alguns desses tipos são comumente encontrados na literatura como árvores e grafos, cujos resultados foram obtidos em maior quantidade, e outras são representações que apresentam uma abordagem com base de identificação completamente diferente das usuais, por exemplo, *wavelets*, métodos formais e análise concólica, as quais apresentam bom desempenho e escalabilidade na detecção de clones complexos comparados a outras abordagens. Entretanto, algumas dessas técnicas não são explicadas adequadamente pelos autores e/ou passaram por poucas avaliações.

As ferramentas facilitam a identificação de clones para os programadores. Elas comumente oferecem interfaces para visualização dos clones identificados, sejam elas gráficas ou não. Além disso, essas ferramentas podem identificar clones de vários tipos, o que as beneficiam, pois identificá-los manualmente é cansativo, dispendioso e propenso a falhas. Nem todas essas ferramentas de detecção são explícitas quanto a isso ou garantem encontrar um tipo de clone específico, o que acaba por generalizar termos como clones semânticos e sintáticos. Tal convenção pode gerar confusão quanto aos tipos identificados, pois não existe exatidão dos tipos que abrange cada termo. Quanto aos tipos identificados, nota-se divergência entre os autores quanto à utilização ou não da classificação usual que ditam 4 tipos de clones relacionados às similaridades semântica e sintática. Com isso, alguns autores preferem utilizar a nomenclatura convencional de separação, sendo os tipos de clones diferidos por similaridades sintáticas e semânticas.

Mesmo com a variedade de linguagens de programação existentes, é perceptível que, em clonagem de código, destaca-se a utilização de sistemas de software desenvolvidos em Java

(46,06%), C (24,71%) e C++ (12,92%). Tais valores podem estar relacionados à popularidade de utilização da linguagem em outras áreas de pesquisa. Isso também revela o porquê da maioria das abordagens existentes empregarem suas técnicas em ambientes voltados ao desenvolvimento utilizando o paradigma orientado a objetos e procedural. Também, pode-se perceber que os artigos são comumente aplicados em contextos industriais e acadêmicos. No contexto industrial, a detecção de clones em sistemas de software trata principalmente da realização de testes nesses sistemas, visando às melhorias de manutenção. No contexto acadêmico, relativamente maior que o industrial, são destacadas pesquisas buscando o desenvolvimento de ferramentas que proporcionam mais escalabilidade e desempenho. Outros aspectos comumente abordados juntamente com a detecção de clones são refatoração de códigos clonados (com o objetivo de manutenção) e necessidade/propensão a gerência de clones.

Relatados, em sua maioria, no conteúdo dos artigos analisados, clones de código têm impacto direto na manutenção de sistemas de software. Ainda que diretamente tal manutenção englobe família de sistemas, não foram identificadas abordagens que direcionassem ou mencionassem o contexto de LPS. A detecção por meio de ferramentas e de abordagens busca facilitar a sua identificação pelos programadores, a fim de reduzir propagações de erros disseminados por clones não identificados, bem como facilitar inserções/alterações/remoções de funções em sistemas de software.

2.6 Considerações Finais

Neste capítulo, foi apresentado um estudo exploratório utilizando a técnica Mapeamento Sistemático de Literatura sobre a detecção de códigos clonados, no nível de código, em sistemas de software. Foram analisados e coletados dados de 158 artigos relacionados ao tema para responder as questões de pesquisa. Algumas informações foram identificadas a respeito das características gerais coletadas nos artigos selecionados.

Em relação às bibliotecas digitais, em ordem decrescente, a quantidade de artigos analisados está concentrada na IEEE (36,07% - 57 artigos), na ACM (20,25% - 32 artigos), Ei Compendex (20,25% - 32 artigos), Scopus (18,35% - 29 artigos), Springer Link (2,53% - 4 artigos) e Science Direct (2,53% - 4 artigos). Com relação ao ano de publicação, foi considerado o período de 2001 a 2020 (inclusive e incompleto), onde a maior concentração de publicações foi no ano de 2017. Quanto aos autores, quatro autores principais destacaram-se com publicações superiores/iguais a cinco artigos. Além disso, foram identificados dois grupos de

pesquisadores que têm publicado em conjunto; um grupo com cinco pesquisadores que publicaram quatro artigos juntos e outro grupo com dois pesquisadores que publicaram quatro artigos juntos. Também, foi contabilizado ao todo 84 veículos de divulgação de trabalhos científicos diferentes, categorizando a diversificação de aplicação da detecção de clones.

Como resposta à QP1, como resultado foram encontradas 70 ferramentas implementadas baseadas em tipos diferentes. O total de 30 técnicas foram utilizadas para encontrar tipos de clone. Foram encontrados os quatro tipos de clone ao final do estudo das ferramentas e das abordagens analisadas, respondendo a QP2. Para a QP3, foram identificadas 13 linguagens de programação diferentes e 6 paradigmas de programação.

3 LINHA DE PRODUTOS DE SOFTWARE

3.1 Considerações Iniciais

Uma Linha de Produtos de Software (LPS) é formada por um conjunto de sistemas de software desenvolvidos com base em um código comum e buscam atingir um segmento de mercado específico [Apel *et al.*, 2013]. A gerência desse recurso de desenvolvimento da LPS é feita por meio de um modelo de árvore, operadores lógicos e restrições que podem ser implementados por tecnologias baseadas em composição [Vale *et al.*, 2015]. O objetivo da utilização de LPS tange principalmente o aumento do reúso e da qualidade de sistemas de software e diminuição dos custos de manutenção [Marimuthu; Chandrasekaran, 2017].

Este capítulo está estruturado da seguinte maneira. Na Seção 3.2, estão descritos alguns conceitos sobre LPS e uma visão geral sobre o assunto. Na Seção 3.3, é apresentado o processo de desenvolvimento de uma LPS. Na Seção 3.4, é abordada a construção de produtos por meio da composição de características e sua modelagem por meio de um modelo de configuração (*feature model*) representando as variabilidades de uma LPS. Na Seção 3.5, a evolução e a clonagem de código são descritas em termo de LPS.

3.2 Conceitos

O termo “Linha de Produtos de Software” surgiu da necessidade da indústria em promover adaptações a um produto, de acordo com as necessidades particulares de cada cliente [Apel *et al.*, 2013]. Isso possibilitou a variabilidade de produtos que compõe uma família, gerada a partir de um conjunto de características em comum [Laguna; Crespo, 2013]. A variação das características na modelagem de LPS possibilita distinguir os produtos resultantes de uma mesma LPS, quando combinadas e relacionadas.

Aplicada ao contexto de LPS, uma característica ou recurso (*feature*) tem como principal utilização representar comunalidades e diferenças entre produtos. Das várias definições destacadas, em síntese, tem-se:

> "Uma *feature* é um comportamento característico ou visível ao usuário final de um sistema de software. Os recursos são usados na engenharia de linha de produto para especificar e comunicar semelhanças e diferenças dos produtos entre as partes interessadas e para orientar a estrutura, a reutilização e a variação em todas as fases do ciclo de vida do software." [APEL *et al.*, 2013, p. 18, tradução nossa]

Assim sendo, um subconjunto de características selecionadas e seus relacionamentos modelam a estrutura de um produto. Tal composição envolvendo características e relacionamentos é gerenciada por meio de suas dependências, sejam elas mandatórias ou opcionais, cujos processo de seleção e suas restrições compõem o chamado modelo de características [Apel *et al.*, 2013], abordado posteriormente em outra seção deste capítulo.

3.3 Processo de Desenvolvimento

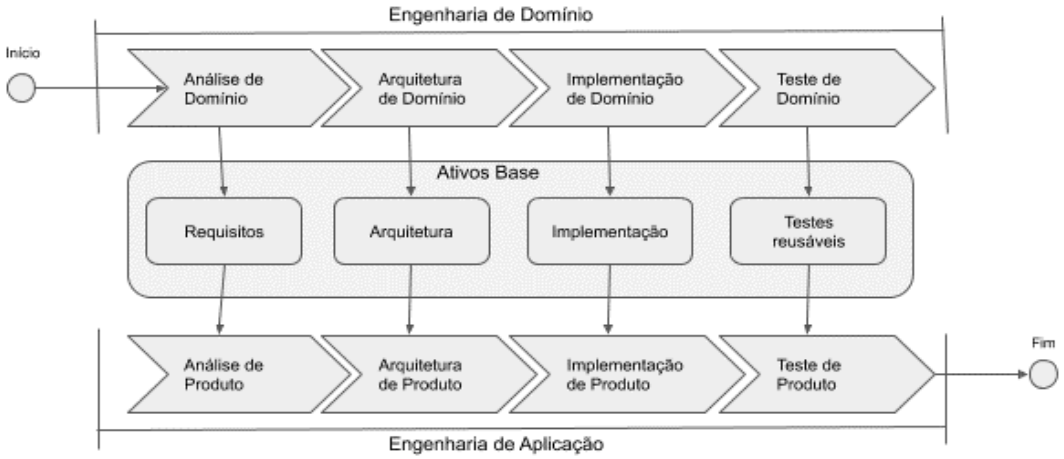
Na engenharia de software tradicional, as etapas que decorrem desde a idealização até a "morte" de um sistema de software específico consistem no que se conhece como ciclo de vida desses sistemas. Essas etapas são pensadas e executadas para um único sistema alvo [Apel *et al.*, 2013]. Por outro lado, a Engenharia de Linha de Produto de Software (ELPS) (*Software Product Line Engineering* - SPLE) deve ser pensada para um domínio semelhante, mas não idêntico de sistemas similares. A fim de estabelecer uma nova LPS, o processo **Engenharia de Domínio** é utilizado, no qual ocorre o desenvolvimento de ativos necessários à construção de produtos variados (visando ao reúso), onde não se tem a intenção nem é possível gerar um produto final [Apel *et al.*, 2013]. Posteriormente, o processo de derivação de produtos dessa LPS é chamado **Engenharia de Aplicação** [Marimuthu; Chandrasekaran, 2017], no qual a necessidade de customização possibilita a derivação de produtos. Em ambos os processos (Figura 3.1), são realizadas atividades de análise, de arquitetura, de implementação e de testes [Apel *et al.*, 2013; Metzger; Pohl, 2014]:

a) O processo **Engenharia de Domínio** consiste no processo de analisar o domínio de uma LPS e construir artefatos reusáveis. Tal processo não resulta em um produto de software específico, mas prepara os artefatos para serem utilizados em um ou diversos produtos [Apel *et al.*, 2013]. As atividades são:

   - **Análise de Domínio**. Pode ser traduzida como uma forma de engenharia de requisitos voltada para a construção da LPS. Nessa atividade, são tomadas decisões voltadas para o escopo pretendido, quais serão os artefatos de reúso e a documentação destas decisões é transcrita no modelo de características;
   - **Arquitetura de Domínio**. Engloba atividades para definir a arquitetura usada na LPS. Ocorre a definição de pontos de variação e quais serão as variantes para a LPS em questão. Dessa maneira, a arquitetura de composição é um exemplo de arquitetura que pode ser utilizada, onde a variabilidade é dada em função de pontos de variações ou composição de componentes;

- **Implementação de Domínio**. Consiste na implementação dos artefatos de domínio para tornarem-se artefatos de reúso na LPS. Envolve a seleção da estratégia de implementação, preparação do *design* e do código reutilizáveis e outros documentos necessários (modelo, casos de teste e outros);
- **Teste de Domínio**. Tem como objetivo garantir a qualidade das *features* que vão compor a LPS por meio de testes;

Figura 3.1 - Engenharia de Domínio e Engenharia de Aplicação.



Fonte: Adaptado de [Silva *et al.*, 2011]

b) No processo **Engenharia de Aplicação**, os artefatos desenvolvidos no processo de Engenharia de Domínio são utilizados para a construção de um produto específico, visando à necessidade do cliente. Esse processo assemelha-se a engenharia de software tradicional, porém com foco em reúso. As atividades são:

- **Análise de Produto**. São determinados os requisitos específicos para uma aplicação utilizando o relacionamento direto entre o requisito de domínio e os recursos da aplicação que a corresponde;
- **Arquitetura de Produto**. São considerados os requisitos da aplicação e a Arquitetura de Domínio para selecionar as variabilidades necessárias para encaixar no escopo pretendido;
- **Implementação de Produto**. O código é ajustado baseado na Arquitetura de Produto e na Análise de Produto. Podem ser usadas técnicas de configuração de software para facilitar a parametrização e a composição dos módulos de código reutilizáveis;
- **Teste de Produto**. O objetivo é testar a LPS. São feitos testes a fim de validar e verificar a LPS com base no que foi especificado e garantir a que estejam corretas as das variantes.

3.4 Modelagem de Características (*Features*)

Inserida no contexto da Análise de Domínio, a modelagem de características representa a variabilidade de uma LPS, tomando como base suas funções. Essa modelagem segue duas vertentes em um modelo de representação [Schulze, 2013]: i) observações das características primitivas; e ii) observações das características compostas.

A modelagem das características de uma LPS pode ser expressa em termos de *features*. Originalmente, o conceito de *features* foi introduzido por meio do método *Feature Oriented Domain Analysis* (FODA) [Kang *et al.*, 1990]. Tal método tem por finalidade utilizar a apresentação das *features* e seus relacionamentos para esboçar um modelo que estabeleça, em função das informações obtidas no domínio, uma coleção de *features* hierarquicamente distribuídas para serem utilizadas de forma combinatória na geração de produtos diferentes.

3.4.1 Variabilidades

O que difere os produtos gerados em uma LPS são as variabilidades acrescentadas a cada um deles no decorrer da seleção do conjunto de características (*features*) específicas que os comporão. A implementação de uma LPS segue uma estrutura de árvore, pelo método FODA [Kang *et al.*, 1990], que representa um modelo de exibição de características (*feature model*). As *features* obrigatórias podem ser acrescidas de *features* opcionais variáveis (caracterizando a variabilidade) e as várias combinações diferentes possibilitadas pelo *feature model*, geram produtos diferentes.

Variabilidades em uma LPS são inseridas a partir do momento da seleção de *features* para cada variante [Schulze, 2013]. Partindo do princípio de que a inserção de variabilidades ocorre mediante a seleção de *features*, as *features* primitivas podem ser descritas como aquelas que não possuem sub-características e *features* compostas são formadas por agrupamento de sub-características, que seguem um conceito em comum. A LPS pode ser constituída de *features* compostas, *features* primitivas ou a combinação de ambas *features*. A adição de novas *features* em um modelo gera variabilidade de produtos.

3.4.2 Modelo de Características (*Feature Model*)

Features obrigatórias estão presentes em todos os produtos da LPS. *Features* opcionais pertencem a alguns produtos. *Features* alternativas, dado um conjunto de *features*, possibilita a escolha de somente uma *feature* [Colanzi, 2014]. Esse agrupamento de *features* possibilita a

formação de um modelo de variabilidades comumente representado por um modelo de características (*feature model*).

O ato de customizar um produto que satisfaça os requisitos de um cliente envolve selecionar e relacionar as *features* necessárias. Assim sendo, são utilizadas para definir o domínio em termo de *features* [Apel *et al.*, 2013]:

a) **Característica Obrigatória (*Mandatory Feature*)**. Indica que, obrigatoriamente, a *feature* deve ser selecionada se seu pai (*feature*) for selecionado e vice-versa. Na Figura 3.2a, observa-se a representação da equivalência lógica por meio de um círculo preenchido na *feature* `f`. A formulação em proposição lógica pode ser definida como a equivalência lógica `mandatory (p, f)`, sendo `p` representando a *feature* pai e `f` representando a sua *feature* filha, na hierarquia:

$$mandatory\,(p, f) \equiv f \Leftrightarrow p$$

b) **Característica Opcional (*Optional Feature*)**. Indica que, obrigatoriamente, a *feature* deve ser selecionada se seu pai (*feature*) for selecionado. Na Figura 3.2b, observa-se a representação da implicação lógica por meio de um círculo vazio na *feature* `f`. A formulação em proposição lógica pode ser definida como a implicação lógica `optional (p, f)`, sendo `p` representando pela *feature* pai e `f` representando a sua *feature* filha, na hierarquia:

$$optional\,(p, f) \equiv f \Rightarrow p$$

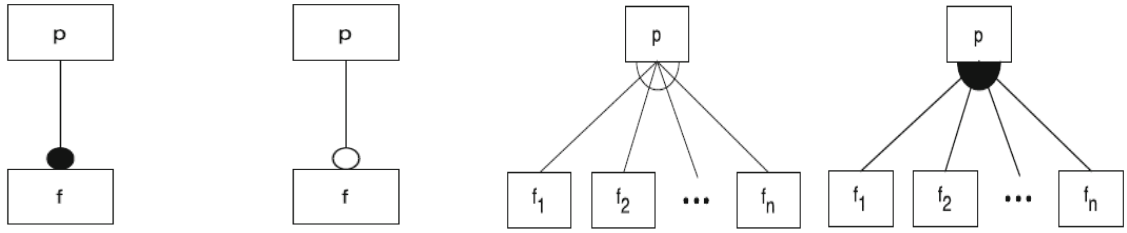
c) **Ou Irrestrito (*Unrestrict Or*)**. Indica que uma ou mais *features* (disjunção) pode ser selecionada para uma *feature* pai. Na Figura 3.2c, observa-se como é representada a notação de disjunção lógica. A formulação em proposição lógica pode ser definida como a disjunção lógica `or (p, f)`, sendo `p` representando a *features* pai e `f` representando a sua *feature* filha, na hierarquia:

$$or\,(p, \{f_1, \ldots, f_n\}) \equiv ((f_1 \vee \ldots \vee f_n) \Leftrightarrow p)$$

d) **Restrição Alternativa (*Alternative Constraint*)**. Indica que não mais que uma *feature* (disjunção exclusiva) pode ser selecionada para uma *feature* pai. Na Figura 3.2d, observa-se como é representada a notação de *features* alternativas. A formulação em proposição lógica pode ser definida como a equivalência lógica `alternative (p, f)`, sendo `p` representando a *feature* pai e `f` representando a sua *feature* filho, na hierarquia:

$$alternative\ (p, \{f_1, \ldots, f_n\}) \equiv \left((f_1 \vee \ldots \vee f_n) \Leftrightarrow p \right) \bigwedge_{i<j} \neg (f_i \wedge f_j)$$

Figura 3.2 - *Mandatory Feature* (a), *Optional Feature* (b), *Alternative Constraint* (c) e *Unrestrict Or* (d).



(a) (b) (c) (d)

Fonte: Apel *et al.* (2013)

O modelo de características (*feature model*) está inserido no contexto de modelagem de variabilidades e tem por finalidade representar os ativos de uma LPS e seus relacionamentos [Apel *et al.*, 2013]. Esse modelo deve ser capaz de conduzir descritivamente a um cliente os seus requisitos, bem como dispor dos recursos solicitados pelos *steakholders*.

3.5 Importância de Detecção de Código Clonado

As LPS foram propostas como uma abordagem melhor estruturada para a reutilização de artefatos de código fonte entre um conjunto de sistemas com características semelhantes, conhecidos como família [Apel *et al.*, 2013]. Nesse contexto, o crescimento exagerado da LPS é justificado por alguns fatores, por exemplo, a existência de clientes diferentes com requisitos diferentes e modificações nos produtos de empresas globais [Mende *et al.*, 2008]. Tal crescimento está diretamente relacionado à Evolução de LPS e a gerência de sistemas variados.

A oscilação que ocorre na evolução e na melhoria de produtos e o fenômeno de mitose de software [Faust; Verhoef, 2003] leva ao problema destacado pela abordagem Crescimento e Poda (*Grown-and-Prune*). Na fase Poda, códigos similares ou clones podem ser utilizados para formar um núcleo ativo de uma LPS [Mende *et al.*, 2008]. Os fenômenos Mitose e Oscilação na configuração de software são [Faust; Verhoef, 2003]:

a) **Mitose** de software ocorre quando, com o passar do tempo, variantes de um mesmo sistema de software surgem em decorrência de modificações locais efetuadas de forma descontrolada ou modificações centrais controladas (gerenciadas). Essas últimas acabam por seguir o mesmo caminho das modificações locais descontroladas aumentando a ocorrência de variantes desse sistema, quando não são feitas com o devido gerenciamento a fim de serem reproduzidas por meio do sistema original. As principais alterações que

propiciam o acontecimento de modificações locais descontroladas são o crescimento da modificação, o controle local de modificações e a localização da organização que ocorre de forma espalhada. Entretanto, tal fenômeno não deve ser considerado algo ruim, visto que pode trazer benefícios às necessidades da empresa;

b) **Oscilação** na configuração ocorre por causa da existência de mitoses de sistemas de software, pois as oscilações de configuração aumentam constantemente em decorrência de serem pressionadas pela alta demanda de customização e de modificação de variantes desses sistemas para atender a necessidades de clientes. Consequentemente, por conta de custo e de prazo, a clonagem de código torna-se prática comum em fenômenos de oscilação na configuração, dificultando a gerência de configurações oscila entre ser genericamente ou especificamente configuráveis.

A abordagem Crescimento e Poda (*Grown-and-Prune Model*) é similar à construção de sistemas de software ao crescimento de árvores, buscando estabelecer boa forma e equilíbrio e objetivando melhorar a robustez e o crescimento de maneira desejada [Faust; Verhoef, 2003]. Assim, para os fenômenos Mitose e Oscilações não terem impacto significativo no crescimento, são estabelecidas épocas de poda em fases diferentes do crescimento, o que nem sempre é trivial, se observada a quantidade de variabilidades em uma floresta de crescimento.

Códigos similares (clones de código) são identificados em características diferentes a fim de tornarem-se uma única característica, ou seja, serem reutilizadas [Mende *et al.*, 2009]. A ocorrência de clones em LPS tem mais incidência em recursos alternativos e independem da abordagem utilizada na implementação [Mende *et al.*, 2008].

3.6 Considerações Finais

Neste capítulo, foram abordados alguns temas importantes para o entendimento e a contextualização de LPS. A combinação de características (*features*) e seus relacionamentos leva à formação de produtos configuráveis de maneira variável, isto é, customizados. Para isso, um modelo de características (*feature model*) foi proposto com a finalidade de estabelecer um catálogo de hierarquias de configurações, bem como definir obrigatoriedades, opcionalidades, disjunções e disjunções exclusivas para essas configurações customizáveis. Nesse contexto, o processo de desenvolvimento de LPS é essencial na identificação, na definição e na criação de *features* e produtos, dando destaque à fase Análise de Domínio para base de todo o processo de estabelecer o domínio dos recursos de um cliente.

4 ABORDAGEM HÍBRIDA

4.1 Considerações Iniciais

Clones de código podem se propagar por várias partes do escopo de implementação de sistemas de software, o que dificulta o processo de manutenção e aumenta a probabilidade de disseminação de erros. Neste capítulo, é descrita uma proposta de abordagem híbrida para detecção de clones em LPS, utilizando a combinação as técnicas "Sequência de Chamadas de Métodos" e "Análise de Assinatura".

Além disso, para (semi) automatizar a abordagem híbrida proposta para detecção de código clonado, foi desenvolvido um apoio computacional combinando as técnicas "Sequência de Chamadas de Métodos" e "Análise Estrutural de Assinaturas de Métodos". Tal apoio foi implementado em Java em forma de *plug-in* para o Eclipse IDE (*Integrated Development Environment*) Jakarta e realiza a detecção de códigos clonados em LPS por meio da manipulação da AST, construção de grafos de dependências das chamadas dos métodos das classes, análise de isomorfias (estruturais) em grafos, análise da assinatura de métodos e validações sintáticas para classificação do tipo de clone.

O restante deste capítulo está organizado da seguinte forma. Na Seção 4.2, são apresentados os conceitos fundamentais ao entendimento da abordagem proposta. Na Seção 4.3, uma visão com mais detalhes da abordagem proposta é descrita. Na Seção 4.4, é apresentada a implementação do apoio computacional.

4.2 Conceitos

Conhecer os conceitos das técnicas "Sequência de Chamadas de Métodos" e "Análise estrutural de Assinaturas de Métodos" é fundamental para o entendimento deste trabalho. A **Sequência de Chamadas de Métodos** pode ser entendida como invocações/acessos a métodos de várias classes de um mesmo sistema de software, possuindo uma ordem sintática de execução semelhante, a fim de estabelecer um resultado pretendido ou atender a uma ordem solicitada. Logo, a ordem de execução dessas chamadas deve ser preservada. A frequência de ocorrências de mesma sequência de chamadas de métodos em diferentes partes do sistema de software tem alto indício da existência de código clonado. Alguns indícios que podem ser considerados clones são [Paiva; Figueiredo, 2016]:

a) Sequência de chamadas de método fazem computação similar;
b) Sequências podem ser formadas por chamadas de métodos de diversas classes;

c) Classes diferentes, provedoras de serviços específicos ou comportamento comum encapsulado que misturam chamadas de métodos tem maior força de computação similar.

A **Análise Estrutural de Assinaturas de Métodos** diz respeito a assinatura de um método, sendo uma forma única de identificá-lo e é empregada para fins de convenção de código em Java. Essa forma única de identificação é composta por identificador e tipos dos parâmetros formais pois são responsáveis por garantir a unicidade de um método dentro de uma classe. A assinatura de um método é empregada em diversas linguagens de programação a fim de estabelecer um formalismo na declaração de métodos no código fonte de sistemas de software [Rot, 1993]. Neste trabalho, a assinatura (A) do método (M) é dada por:

```
A(M) = identificador(tipo(s) parâmetro(s))
```

Estruturas de grafos podem ser utilizadas para armazenar informações que possuem relação entre elas, por exemplo, as chamadas de métodos. Grafos possuem características específicas; nesse contexto, os grafos isomorfos podem ser considerados grafos que possuem uma relação de equivalência estrutural, no que diz respeito a preservação das adjacências entre dois vértices [Picado, 2014]. Neste trabalho, é considerado isomorfia de grafos quando, dados dois grafos, a quantidade de adjacências (ligações) é preservada, nó a nó do grafo e em todos os níveis do grafo.

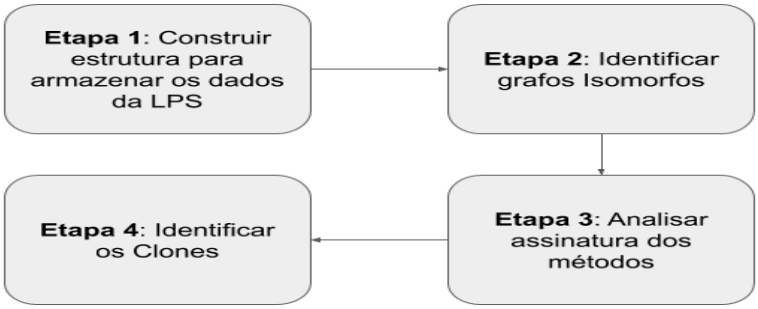
4.3 Abordagem

Clones de código em LPS são frequentemente identificados quando o fenômeno de crescimento exagerado de uma LPS ocorre em função da alta demanda de customizações de produtos para clientes diferentes, em curto período [Schulze *et al.*, 2011]. Tais necessidades tendem a aumentar a prática de copiar e colar códigos, reaproveitando funções por meio da reutilização de trechos de código em virtude da diminuição do tempo de implementação e entrega. Com os resultados obtidos com MSL realizado (Capítulo 2), foi elaborada uma proposta de abordagem híbrida, cujo objetivo é detectar clones de código em LPS, visto que não foram identificadas ferramentas que efetuassem essa detecção no contexto pretendido (LPS desenvolvidas sob o paradigma orientado a características). Uma abordagem híbrida foi adotada a fim de promover mais versatilidade no tratamento de detecção de clones dos Tipos 1 e 2 existentes na literatura. Essa abordagem consiste em quatro etapas (Figura 4.1):

a) **Etapa 1 - Construir estrutura para armazenar os dados da LPS (Figura 4.2)**. Os dados (métodos) encontrados na análise realizada pela abordagem híbrida proposta são

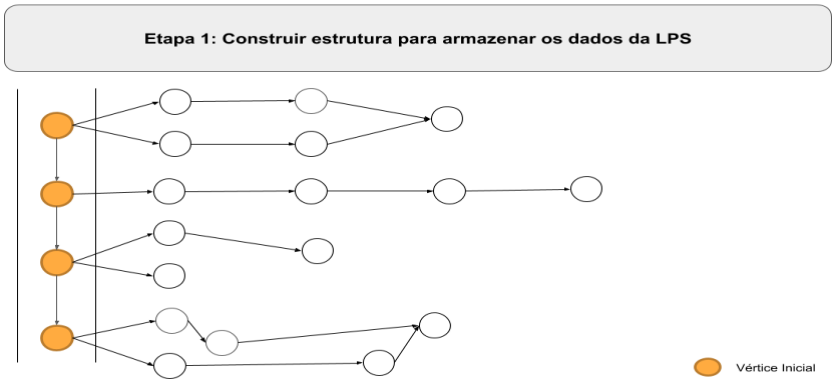
representados por meio de grafos. Cada nó de um grafo é a representação abstrata de um método de uma *feature* da LPS. O intuito é, ao verificar de maneira estática o código fonte de uma LPS, um grafo ser construído seguindo a ordem das chamadas dos métodos existente dentro de cada método que implementa as *features* de uma LPS. Supondo a existência de 4 métodos, cada um desses métodos realiza chamadas (ou não) a outros métodos para serem executados em sua estrutura de implementação e assim sucessivamente, até que o último método tenha sua estrutura de implementação analisada por completo. Partindo da quantidade de métodos total existente em uma LPS, cada um desses métodos torna-se o vértice inicial (círculo preenchido em laranja) de um grafo diferente a ser construído. Analisando sequencialmente as chamadas a outros métodos que um método faz, as demais chamadas/invocações desse método a outros métodos são inseridas no grafo por meio de ligações (arestas) nesse grafo. Cada vértice do grafo representa um método (círculo) na lista principal de grafos;

Figura 4.1 - Etapas da Abordagem Híbrida Proposta.



Fonte: Do autor (2020)

Figura 4.2 - Criação da Estrutura de Grafos.

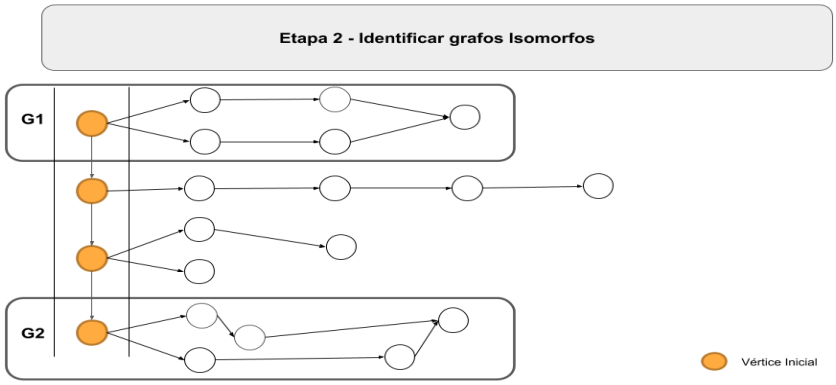


Fonte: Do autor (2020)

b) **Etapa 2 - Identificar grafos isomorfos (Figura 4.3)**. Após a construção dos grafos, todos os métodos são comparados, considerando a estrutura de cada um deles, em busca de grafos isomorfos. Ou seja, sejam `G1` e `G2` dois grafos obtidos após construir a lista principal de

grafos (Etapa 1), se a quantidade de ligações (arestas), analisando vértice a vértice de todos os vértices de `G1` e de `G2`, for a mesma em `G1` e `G2`, eles são grafos isomorfos pois existe a preservação de adjacências de vértices em todos os níveis de cada um desses grafos. Logo, os métodos representados pelos vértices iniciais `G1` e `G2` são considerados candidatos a clones e permanecerão para serem analisados nas demais etapas. Ao analisar o conteúdo presente em cada método, pode-se observar que isomorfias não implicam necessariamente em chamada de métodos idênticas (por mais que elas possam em algum momento serem idênticas) pois não são considerados os métodos em si mas a forma de níveis que configuram a chamada de um método a outros métodos no sistema;

Figura 4.3 - Identificação de Grafos Isomórficos.



Fonte: Do autor (2020)

c) **Etapa 3 - Analisar assinatura dos métodos**. Os grafos isomorfos são analisados na Etapa 3 quanto à assinatura dos métodos (dois a dois) para encontrar semelhanças. Partindo do pressuposto que, se dois métodos possuem coincidências (semelhanças pré-estabelecidas) em sua assinatura, elas podem ser indícios da existência de clonagem de código. Sejam `M1` e `M2` dois métodos de grafos diferentes, se `A(M1) = A(M2)` então `M1` e `M2` são possíveis candidatos a clones. Nesse contexto, a fim de garantir mais precisão, serão considerados dois tipos de composições de informações (coincidências) presentes na assinatura permitidas no nível de programação:

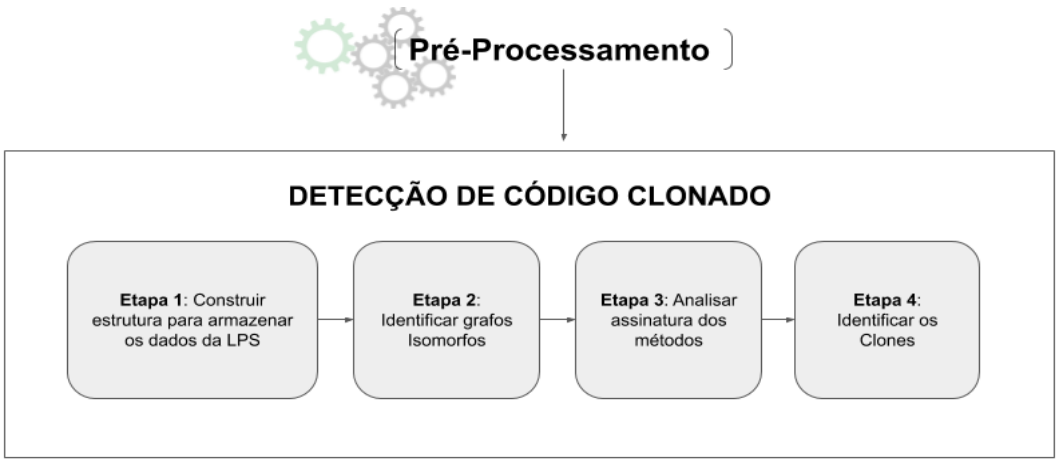
- **Apenas identificador do método igual**. Quando o identificador dos métodos analisados for igual, pois, em uma mesma classe, o identificador de um método torna o método único;
- **Apenas tipos de parâmetros iguais**. Quando o tipo do(s) parâmetro(s) esperado(s) pelo método for igual em ambos os métodos analisados pois, ainda que o identificador dos métodos e do(s) parâmetro(s) seja diferente, dois métodos podem ter computação similar;

d) **Etapa 4 - Identificar os Clones**. Após a identificação de possíveis métodos candidatos a clones, são procurados clones do Tipo 1 e do Tipo 2. Os clones do Tipo 3 e do Tipo 4 não foram considerados tendo em vista que as etapas anteriores reduziram significativamente a quantidade de dados disponíveis para análise, após a realização de testes realizados. Além disso, foi considerada a complexidade da identificação desses tipos onde a maioria das ferramentas disponível atualmente identificam apenas esses dois tipos (os mais frequentemente detectados) [Batista *et al.*, 2019]. Na análise do código fonte dos métodos para encontrar os clones, o objetivo é encontrar semelhanças em estruturas de código de atribuição, de repetição e de decisão entre dois métodos resultantes da Etapa 3. Os clones foram analisados seguindo os critérios: i) **Tipo 1**, clones em estruturas de atribuição, de repetição e de decisão idênticas; e ii) **Tipo 2**, clones em estruturas de atribuição, de repetição e de decisão com pequenas alterações em identificadores, literais, tipos e *layout.*

4.4 Implementação

O apoio computacional desenvolvido utiliza os métodos declarados e chamados/invocados na implementação da LPS como conjunto principal de informações. Para iniciar o processo (Figura 4.4) de detecção de código clonado em LPS, da abordagem híbrida proposta, é preciso realizar um pré-processamento que possibilitará utilizar a AST, uma estrutura base do Eclipse IDE que permite analisar estaticamente o código por meio de "visitas" às estruturas específicas disponíveis em sua biblioteca.

Figura 4.4 - Processo da Implementação da Abordagem Híbrida Proposta.



Fonte: Do autor (2020)

Esse pré-processamento consiste em modificar o arquivo de configuração do produto gerado pela LPS (criado a partir das *features* base de uma LPS). Inicialmente, devem ser localizados os métodos (localizados em pacotes/pastas) a serem analisados. Ao configurar um

produto de uma LPS no Elipse IDE (por meio do *plug-in* FeatureIDE, que possibilita o desenvolvimento de LPS e a linguagem jakarta de AHEAD), é gerado um arquivo de configuração (`configuration.xml`) composto pelos refinamentos das classes (no formato `.jak`) pertencentes às *features* selecionadas. Como a proposta deste trabalho é coletar os dados provenientes da LPS e não apenas considerar isoladamente os produtos que podem ser gerados, o arquivo de configuração é alterado para conter os refinamentos de classe de todas as *features* sem considerar as restrições do modelo de configuração (composições) impostas pelo *feature model* (para ter acesso ao código). O arquivo de configuração do produto original anteriormente configurado (ou a configuração padrão da LPS caso nenhum produto ainda tenha sido configurado pelo usuário) é armazenado em outro lugar (*backup*) e pode ser acessado pelo caminho `<project_name>/configs/backup/configuration.xml`.

Cabe ressaltar que tal alteração não viola as restrições de composição impostas pelo *feature model* da LPS, onde somente é possível configurar um produto por vez e compilá-lo em produto de software, uma vez que a configuração foi manipulada e tornou-se inválida para o modelo de restrições. O novo arquivo criado tem por única e exclusiva finalidade a seleção do código fonte das classes e dos refinamentos da LPS para análise de código estática, por meio da manipulação da AST. As transformações necessárias são efetuadas automaticamente pelo *plug-in* FeatureIDE em tempo de compilação e são transparentes ao usuário.

O pré-processamento é implementado na classe `CodeFragments.java` onde os fragmentos de código são manipulados e armazenados para posterior utilização. As quatro etapas de detecção de código são Etapa 1 - Construir Estrutura para Armazenar os Dados da LPS, Etapa 2 - Identificar grafos isomorfos, Etapa 3 - Analisar assinatura dos métodos e Etapa 4 - Identificar os Clones (Etapa 4). Para verificar os tipos de clones nessa abordagem, as

4.4.1 Etapa 1 - Construir Estrutura para Armazenar os Dados da LPS

Reunidas as informações necessárias em um mesmo local, foi utilizada a AST do Eclipse IDE para a coleta de dados, por meio da manipulação do código fonte. A AST contém uma classe `ASTParse` que analisa uma *string* contendo um código fonte e retorna uma árvore de sintaxe abstrata com a qual pode-se capturar cada elemento de classes como uma subclasse da classe `ASTNode`. Após reunir o conteúdo necessário no pacote *default*, pode-se implementar métodos de visitas aos nós dessa árvore por meio da classe `ASTVisitor` e realizar a extração dos dados dos métodos que a ferramenta utiliza. Foram utilizados os métodos

`ICompilationUnit()` e `IJavaProject()` para capturar o arquivo do projeto no Eclipse IDE e o arquivo de origem da LPS. Na classe `DependencyVisitor`, foram utilizadas as classes `MethodDeclaration` e `MethodInvocation` que disponibilizam acesso a características específicas dos métodos. Para cada declaração de método encontrada, um novo objeto `MethodData` é instanciado. Na classe `MethodData` (Código 4.1), são armazenados os atributos parâmetros, nome, corpo, métodos invocados e modificadores, que caracterizam os objetos a serem utilizados. Para facilitar em algumas comparações necessárias e não perder a relação da classe criadora do método, o nome do método foi concatenado ao nome de sua classe seguindo o formato `classe.método`.

Código 4.1 - Classe `MethodData`.

```
1.  public class MethodData {
2.
3.      private ArrayList<String> parameters = new ArrayList<String>();
4.      private String methodName;
5.      private MethodDeclaration methodBody;
6.      private ArrayList<String> methodsInvo = new ArrayList<String>();
7.      private String retorno;
8.      private int modificador;
9.      private ArrayList<MethodStructure> methodDependencies;
10.
11.     public MethodData(ArrayList<String> parameters, String methodName, MethodDeclar
    ation methodBody,
12.             ArrayList<String> methodsInvo, String retorno, int modificador) {
13.         this.parameters = parameters;
14.         this.methodName = methodName;
15.         this.methodBody = methodBody;
16.         this.methodsInvo = methodsInvo;
17.         this.retorno = retorno;
18.         this.modificador = modificador;
19.         this.methodDependencies = new ArrayList<MethodStructure>();
20.     }
21.
22.     public void addMethodDependency(MethodStructure methodDependency) {
23.         this.methodDependencies.add(methodDependency);
24.     }
25.
26.     public ArrayList<String> getParameters() {
27.         return parameters;
28.     }
29.
30.     public String getMethodName() {
31.         return methodName;
32.     }
33.
34.     public MethodDeclaration getMethodBody() {
35.         return methodBody;
36.     }
37.
38.     public ArrayList<String> getMethodsInvo() {
39.         return methodsInvo;
40.     }
```

(continua)

Código 4.1 - Classe `MethodData`. (cont.)

```
41.
42.
43.
44.
45.
46.     public String getAssinatura() {
47.         return modificador + "." + retorno + "." + methodBody.getName()
48.     }
49.
50.     public void setEStrelas(int valor) {
51.     }
52.
53.     public String getRetorno() {
54.         return retorno;
55.     }
56.
57.     public int getModificador() {
58.         return modificador;
59.     }
60.
61.     public ArrayList<MethodStructure> getMethodDependency() {
62.         return methodDependencies;
63.     }
64.
65. }
```

Fonte: Do autor (2020)

A estrutura principal utilizada no sistema é uma lista de grafos implementada por meio de lista de adjacências, sendo, a partir da lista dos métodos declarados no sistema, construídas listas de chamadas de métodos cujo nó raiz de cada grafo é o método e as adjacências representam chamadas a outros métodos, em subníveis de um grafo. Se o método que acabou de ser inserido no grafo fizer uma chamada a outro(s) método(s), este(s) são inseridos no grafo. A classe que representa essa estrutura de grafos é a classe `GraphStructure` (Código 4.2). Para construir os grafos iniciais, a lista de métodos é criada como `ArrayList<MethodData> metodos`. As execuções são feitas no contexto da classe `ListOfGraph` composta basicamente pelo construtor, pelos métodos `createGraph()`, `addAdjacencia()`, *getters* e *setters*. No método `createGraph()` (Código 4.3), a lista de métodos é percorrida e o nível inicial é igual a zero para cada novo grafo a ser criado. Os nós adjacentes encontrados são armazenados na lista `ArrayList<GraphStructure> graphs`.

Satisfeita a condição de que o atributo `metodosInvo` (usando o método `getMethodsInvo()`) não seja vazio, isto é, de que o método analisado efetua pelo menos uma chamada a outro método, continua a verificação das possíveis chamadas a outros métodos, inserindo o método que possui chamadas a outros métodos na lista de adjacências `graph`. A

continuidade da verificação de subníveis em um nível é dado no método `addAdjacencia()` (Código 4.4) que adiciona um elemento na lista `graph` a cada vez que um método invocado é encontrado na lista de algum outro método.

Código 4.2 - Classe `GraphStructure`.

```
1.       public GraphStructure(int nivel, ArrayList<String> parameters, String methodN
    ame, MethodDeclaration methodBody, ArrayList<String> methodsInvo, String retorno,
    int modificador, ArrayList<MethodStructure> estrutura) {
2.
3.          this.nivel = nivel;
4.          this.parameters = parameters;
5.          this.nome = methodName;
6.          this.methodBody = methodBody;
7.          this.subnivel = methodsInvo;
8.          this.setRetorno(retorno);
9.          this.setModificador(modificador);
10.         this.setMethodDependencies(estrutura);
11.         this.flag = false;
12.     }
13.
14.     public Boolean getFlag() {
15.         return flag;
16.     }
17.
18.     public void setFlag(Boolean flag) {
19.         this.flag = flag;
20.     }
21.
22.     public int getNivel() {
23.         return nivel;
24.     }
25.
26.     public void setNivel(int nivel) {
27.         this.nivel = nivel;
28.     }
29.
30.     public String getNome() {
31.         return nome;
32.     }
33.
34.     public void setNome(String nome) {
35.         this.nome = nome;
36.     }
37.
38.     public ArrayList<String> getSubnivel() {
39.         return subnivel;
40.     }
41.
42.     public void setSubnivel(ArrayList<String> subnivel) {
43.         this.subnivel = subnivel;
44.     }
45.
46.     public ArrayList<String> getParametros() {
47.         return parameters;
48.     }
49.
50.     public MethodDeclaration getMethodBody() {
51.         return methodBody;
52.     }
```

(continua)

Código 4.2 - Classe `GraphStructure`. (cont.)

```
53.
54.     public String getRetorno() {
55.         return retorno;
56.     }
57.
58.     public void setRetorno(String retorno) {
59.         this.retorno = retorno;
60.     }
61.
62.     public int getModificador() {
63.         return modificador;
64.     }
65.
66.     public void setModificador(int modificador) {
67.         this.modificador = modificador;
68.     }
69.
70.   public ArrayList<MethodStructure> getMethodDependencies() {
71.         return methodDependencies;
72.     }
73.
74.     public void setMethodDependencies(ArrayList<MethodStructure> methodDependencies
    ) {
75.         this.methodDependencies = methodDependencies;
76.     }
77.
78. }
```

Fonte: Do autor (2020)

Código 4.3 - Método `createGraph()`.

```
1.      private void createMethod() {
2.          for (Entry<String, ArrayList<Dependency>> entry : code.entrySet()) {
3.              for (int i = 0; i < entry.getValue().size(); i++) { // classe
4.                  metodos.addAll(entry.getValue().get(i).getMethods());
5.              }
6.          }
7.      }
```

Fonte: Do autor (2020)

Código 4.4 - Método `addAdjacencia()`.

```
1.  private void addAdjacencia(String invocadoX) { // adiciona uma lista de adjacencia
     a grafo
2.      for (int k = 0; k < metodos.size(); k++) {
3.
4.          String[] nomeEstruturaGrafo1 = metodos.get(k).getMethodName().split("\\.")
    ;
5.          String[] nomeEstruturaGrafo2 = invocadoX.split("\\.");
6.
7.          if (nomeEstruturaGrafo1[1].equals(nomeEstruturaGrafo2[1])) {

8.              graphs.add(new GraphStructure(nivel, metodos.get(k).getParameters(), m
    etodos.get(k).getMethodName(),
```

(continua)

Código 4.4 - Método `addAdjacencia()`. (cont.)

```
9.                  metodos.get(k).getMethodBody(), metodos.get(k).getMethodsInvo(),
10.                 metodos.get(k).getRetorno(), metodos.get(k).getModificador(),
11.                 metodos.get(k).getMethodDependency()));
12.                 nivel++;
13.         }
14.     }
15. }
```

Fonte: Do autor (2020)

4.4.2 Etapa 2 - Identificar Grafos Isomorfos

Com a lista de adjacências pronta, caracterizando a estrutura de grafos utilizada, pode-se identificar grafos isomorfos, a característica de similaridade considerada para identificar possíveis oportunidades de código clonado. Na classe `IsomorfoEstrutural`, as verificações entre grafos acontecem. A verificação é feita, inicialmente, no método `verificar()` (Código 4.5), percorrendo a lista de grafos e verificando a posição atual e a próxima, se seus subníveis não são vazios e se o método `verificaSubNivel()` tem retorno verdadeiro (`TRUE`) para prosseguir sua execução chamando o método `zerar()`, passando como parâmetro os dois elementos em análise.

Código 4.5 - Método `verificar()`.

```
1.  private void verificar() {
2.      int contTipo1, contTipo2;
3.      contTipo1 = contTipo2 = 0;
4.
5.      for (int i = 0; i < (graphs.size()); i++) { // i compara com i + 1
6.          for (int x = i + 1; x < (graphs.size() - 1); x++) {
7.              if (!graphs.get(i).getSubnivel().isEmpty()) {
8.                  if (!graphs.get(x).getSubnivel().isEmpty()) {
9.                      if (!graphs.get(i).getNome().equals(graphs.get(x).getNome())) {
10.                         if (verificaSubNivel(graphs.get(i), graphs.get(x))) {
11.                             zerar(graphs.get(i));
12.                             zerar(graphs.get(x));
13.                             if (verificarItensAssinatura(graphs.get(i), graphs.get(
    x))) {
14.                                 if (verificarCloneTipo1(graphs.get(i), graphs.get(x
    ))) {
15.                                     if (!verificaImpressoesDuplicadas(graphs.get(i)
    .getNome(),
16.                                             graphs.get(x).getNome(), imprimidosC1 {
17.                                         contTipo1++;
18.                                         System.out.println(" ");
19.                                         System.out.println("---------
    - CLONE TIPO 1 ------------");
20.                                         System.out.println("Classe$$Feature.método
    : " + graphs.get(i).getNome());
21.                                         System.out.println("body do método:");
22.                                         System.out.println(graphs.get(i).getMethodB
    ody().getBody());
```

(continua)

Código 4.5 - Método `verificar()`. (cont.)

```
23.                                     System.out.println("Classe$$Feature.método
   :" + graphs.get(x).getNome());
24.                                     System.out.println("body do método:");
25.                                     System.out.println(graphs.get(x).getMethodB
   ody().getBody());
26.                                     System.out.println("tamanho:" + countNumLin
   has(
27.                                             graphs.get(x).getMethodBody().getBo
   dy().toString()));
28.                                     System.out.println(" ");
29.                                     imprimidosC1.add(graphs.get(i).getNome());
30.                                     imprimidosC1.add(graphs.get(x).getNome());
31.                                 }
32.                             }
33.                             if (verificarCloneTipo2(graphs.get(i), graphs.get(x
   ))) {
34.                                 if (!verificaImpressoesDuplicadas(graphs.get(i)
   .getNome(),
35.                                         graphs.get(x).getNome(), imprimidosC2))
    {
36.                                     contTipo2++;
37.                                     System.out.println(" ");
38.                                     System.out.println("---------
   - CLONE TIPO 2 ------------");
39.                                     System.out.println("Classe$$Feature.método
   : " + graphs.get(i).getNome());
40.                                     System.out.println("body do método:");
41.                                     System.out.println(graphs.get(i).getMethodB
   ody().getBody());
42.                                     System.out.println("Classe$$Feature.método
   :" + graphs.get(x).getNome());
43.                                     System.out.println("body do método:");
44.                                     System.out.println(graphs.get(x).getMethodB
   ody().getBody());
45.                                     System.out.println("tamanho:" + countNumLin
   has(
46.                                             graphs.get(x).getMethodBody().getBo
   dy().toString()));
47.                                     System.out.println(" ");
48.                                     imprimidosC2.add(graphs.get(i).getNome());
49.                                     imprimidosC2.add(graphs.get(x).getNome());
50.                                 }
51.                             }
52.                         }
53.                     }
54.                 }
55.             }
56.         }
57.     }
58. }
```

Fonte: Do autor (2020)

Para melhor entender a lógica de controle aplicada no método `verificar()` no contexto da classe `GraphStructure`, existe um atributo de controle (`flag`) cujo objetivo é efetuar o controle das verificações feitas sendo que, a cada "combinação" encontrada com outro nó da árvore que preserve o isomorfismo do grafo, esse atributo possui o valor `TRUE`. O método recursivo de controle `verificaSubNivel()` (Código 4.6) tem retorno verdadeiro (`TRUE`)

ou falso (FALSE). Retorno igual a FALSE significa que o atributo flag de ambos nós encontram combinações corretas (flag = TRUE), isto é, não ocorreram situações em que a estrutura alterou com a mudança de nível nas comparações nível a nível.

Código 4.6 - Método `verificaSubnivel()`.

```
1.  private Boolean verificaSubNivel(GraphStructure estruturaGrafo, GraphStructure estr
    uturaGrafo2) {
2.          if (((estruturaGrafo != null) && (estruturaGrafo2 != null))
3.                  && ((estruturaGrafo.getFlag().equals(false)) && (estruturaGrafo2.ge
    tFlag().equals(false)))) {
4.              if ((!estruturaGrafo.getSubnivel().isEmpty()) && (!estruturaGrafo2.getS
    ubnivel().isEmpty())) {
5.                  GraphStructure grafo1 = null;
6.                  GraphStructure grafo2 = null;
7.                  estruturaGrafo.setFlag(true);
8.                  estruturaGrafo2.setFlag(true);
9.                  for (int i = 0; i < estruturaGrafo.getSubnivel().size(); i++) {
10.                     for (int k = 0; k < graphs.size(); k++) {
11.                         if (estruturaGrafo.getSubnivel().get(i).equals(graphs.get(k
    ).getNome())
12.                                 && !estruturaGrafo.getSubnivel().isEmpty()) {
13.                             grafo1 = graphs.get(k);
14.                         }
15.                     }
16.                 }
17.                 for (int i = 0; i < estruturaGrafo2.getSubnivel().size(); i++) {
18.                     for (int k = 0; k < graphs.size(); k++) {
19.                         if (estruturaGrafo2.getSubnivel().get(i).equals(graphs.get(
    k).getNome())
20.                                 && !estruturaGrafo2.getSubnivel().isEmpty()) {
21.                             grafo2 = graphs.get(k);
22.                         }
23.                     }
24.                 }
25.                 if (grafo1 == null || grafo2 == null) {
26.                     return true;
27.                 } else {
28.                     verificaSubNivel(grafo1, grafo2);
29.                 }
30.             }
31.         }
```

Fonte: Do autor (2020)

Com valor de retorno para o método igual a TRUE, significa que a condição para continuar a execução não é satisfeita e a verificação recursiva pode ser abortada. Se ao final da execução o retorno for FALSE, os próximos itens na lista que contém o atributo flag = FALSE prosseguem na verificação, descartando os verdadeiros. Caso o retorno seja TRUE, o método zerar() é chamado para atribuir novamente aos nós do grafo o valor de flag = FALSE. O método zerar() percorre a estrutura em análise, alterando o atributo flag para FALSE dos dois grafos analisados, desde o item analisado até sua raiz. Ao final dessa etapa, os

grafos considerados isomorfos são eleitos a prosseguirem para a Etapa 3, onde são feitas verificações de similaridades por meio da análise nas assinaturas dos métodos.

4.4.3 Etapa 3 - Analisar Assinatura dos Métodos

Os métodos identificados na etapa anterior são analisados para encontrar coincidências em sua assinatura. Assim, a implementação inicia-se tomando como base duas estruturas de grafos sendo validadas, considerando que existem possibilidades de serem clones tendo como base sua assinatura, ou seja:

a) Se a quantidade de parâmetros for zero e o identificador for diferente, deve-se retornar `TRUE` pois não existe indícios suficientes de que seu conteúdo não é igual. Por exemplo, tem-se a assinatura `A(M)` de dois métodos hipotéticos `A(escreveTexto) = escreveTexto()`" e `A(retornaValorProcurado) = retornaValorProcurado()` onde os identificadores `escreveTexto` e `retornaValorProcurado` são diferentes;

b) Se os identificadores do método forem iguais deve-se retornar `TRUE`. Por exemplo, tem-se a assinatura `A(M)` de dois métodos hipotéticos `A(comparaValor) = comparaValor()` e `A(comparaValor) = comparaValor (int A, int B)` onde os identificadores `comparaValor` e `comparaValor` são iguais;

c) Se a quantidade de parâmetros e seus tipos coincidirem, deve-se retornar `TRUE`.

O retorno `FALSE` significa que as estruturas não são consideradas possíveis de duplicação e o retorno `TRUE` significa que as estruturas são consideradas possíveis de duplicação e são consideradas ou "elegíveis" para a próxima etapa. Essa explicação é implementada e executada pela função `verificarItensAssinatura()` (Código 4.7), utilizando as variáveis `nomeEstruturaGrafo1` e `nomeEstruturaGrafo2` representando cada método de cada um dos grafos analisados.

Código 4.7 - Método `verificarItensAssinatura()`.

```
1.  private Boolean verificarItensAssinatura(GraphStructure estruturaGrafo, GraphStruct
    ure estruturaGrafo2) {
2.      if (estruturaGrafo == null || estruturaGrafo2 == null)
3.          return false;
4.
5.      String[] nomeEstruturaGrafo1 = estruturaGrafo.getNome().split("\\.");
6.      String[] nomeEstruturaGrafo2 = estruturaGrafo2.getNome().split("\\.");
7.
```

(continua)

Código 4.7 - Método `verificarItensAssinatura()`. (cont.)

```
8.      if ((estruturaGrafo.getParametros().isEmpty() && estruturaGrafo.getParametros(
    ).isEmpty())
9.              && !nomeEstruturaGrafo1[1].equals(nomeEstruturaGrafo2[1]))
10.         return true;
11.
12.     return ((estruturaGrafo.getParametros().size() == estruturaGrafo2.getParametro
    s().size()
13.             && verificarTipoParametro(estruturaGrafo.getParametros(), estruturaGra
    fo2.getParametros()))
14.             || nomeEstruturaGrafo1[1].equals(nomeEstruturaGrafo2[1]));
15.
16. }
```

Fonte: Do autor (2020)

4.4.4 Etapa 4 - Identificar os Clones

Para verificar os tipos de clones nessa abordagem, as implementações:

a) **Clone do Tipo 1**. São verificadas igualdades dentro do *body* de cada método que por sua vez caracterizam uma cópia literal de todo o trecho de código presente em um método. Ou seja, todo o método é igual em ambos os métodos analisados. A implementação dessa verificação é feita pela função `verificarCloneTipo1()` (Código 4.8) que verifica se o corpo das estruturas analisadas é idêntico desconsiderando espaços e endentação;

b) **Clone do Tipo 2**. São verificadas, dentro das estruturas de cada método, linha a linha, comparando se teve alguma alteração em um dos parâmetros que identificam acessos, atribuições e declarações, por exemplo, por meio de um `regex` (Código 4.9). A função `verifyDiffs()` por sua vez possui um `regex` de separadores

```
separators = [\\.\\;\\n\\=\\+\\-
\\(\\)\\{\\}\\[\\]\\\\,\\<\\>\\!\\\"\\'\\/\\*\\&\\|]
```

que delimitam a contagem e verificação das coincidências correspondendo as verificações necessárias.

Código 4.8 - Método `verificarCloneTipo1()`.

```
1.  private Boolean verificarCloneTipo1(GraphStructure estruturaGrafo, GraphStructure e
    struturaGrafo2) {
2.      if (estruturaGrafo == null || estruturaGrafo2 == null)
3.          return false;
4.
5.      if (estruturaGrafo.getMethodBody().getBody() == null || estruturaGrafo2.getMeth
    odBody() == null)
6.          return false;
7.
```

(continua)

Código 4.8 - Método `verificarCloneTipo1()`. (cont.)

```
8.      if (estruturaGrafo.getMethodBody().getBody().toString().replaceAll("[\\n\\t ]",
    "")
9.              .equals(estruturaGrafo2.getMethodBody().getBody().toString().replaceAll
    ("[\\n\\t ]", "")))
10.         return true;
11.     else
12.         return false;
13. }
```

Fonte: Do autor (2020)

Código 4.9 - Método `verificarCloneTipo2()`.

```
1.      private Boolean verificarCloneTipo2(GraphStructure estruturaGrafo, GraphStructu
    re estruturaGrafo2) {
2.          if (estruturaGrafo == null || estruturaGrafo2 == null)
3.              return false;
4.            if (estruturaGrafo.getMethodDependencies() == null || estruturaGrafo2.get
    MethodDependencies() == null)
5.              return false;
6.
7.          if ((estruturaGrafo.getMethodBody().getBody().toString().replaceAll("[\\n\\
    t ]", "")
8.                  .equals(estruturaGrafo2.getMethodBody().getBody().toString().replac
    eAll("[\\n\\t ]", ""))))
9.              return false;
10.         else {
11.             return verifyDiffs(estruturaGrafo.getMethodBody().getBody().toString()
12.                     estruturaGrafo2.getMethodBody().getBody().toString());
13.         }
14.     }
```

4.5 Considerações Finais

Neste trabalho, o objetivo foi elaborar uma abordagem híbrida que utiliza detecção de clones por meio da combinação as técnicas “Sequência de Chamadas de Métodos” e “Análise Estrutural de Assinaturas de Métodos”. Para tanto, foram descritas ambas técnicas e suas peculiaridades na composição, a utilização de isomorfias de grafos em relação a estrutura e a estrutura final a ser analisada. Além disso, foram apresentadas as etapas do processo de construção da abordagem, explicando quais seriam as ações executadas e o próximo passo até a finalização do processo.

Com a proposta de abordagem definida, foi desenvolvido um apoio computacional para (semi) automatizar o processo de detecção de clones de código. A implementação foi feita em forma de *plug-in* para o Eclipse IDE na linguagem Java. A abordagem proposta detecta isomorfia de grafos por meio da verificação em níveis considerando a preservação da quantidade de adjacências de em cada nível, de nó para nó analisado, sendo os dados de entrada

para prosseguir na análise dos métodos chamados, por meio da estrutura construída. Em termos de implementação, foram desenvolvidas as detecções de clones do Tipo 1 e do Tipo 2.

5 AVALIAÇÃO DA ABORDAGEM

5.1 Considerações Iniciais

O processo de avaliação da abordagem de detecção de clones proposta neste trabalho tem por objetivo realizar a análise e a avaliação dos resultados obtidos após utilizar a abordagem por meio do apoio computacional desenvolvido. Para isso, foram conduzidas duas avaliações: i) **avaliação interna**, na qual foram calculados os valores das medidas de precisão e *recall* por meio da utilização de um oráculo; e ii) **avaliação externa**, na qual a ferramenta, foi comparada com outra abordagem disponível na literatura. Ao longo desse capítulo, para fins de facilidade de nomenclatura e escrita, a `LPS Tank War` será abreviada como `LPS TW`. O projeto que contém o código fonte com as implementações das *features* da `LPS TW` pode ser obtido via importação de um novo projeto, utilizando os exemplos disponibilizados no FeatureIDE para AHEAD.

Este capítulo está estruturado da seguinte forma. Na Seção 5.2, são mensurados os valores de precisão e *recall* para a abordagem proposta por meio da realização da avaliação com dados esperados (onde a saída deve ser analisada em relação a um valor esperado) nas análises. Na Seção 5.3, é realizada uma comparação e discussão dos resultados obtidos ao realizar a análise da `LPS TW` pelo apoio computacional desenvolvido e pela ferramenta `CPD` (*Copy/Paste Detector*) do `PMD`[12] (*Programming Mistake Detector*).

5.2 Avaliação Interna

Como parte da avaliação da abordagem, foi realizada a avaliação interna, que consistiu em obter e avaliar, com base em um resultado manipulado/esperado proveniente de um oráculo, os valores das medidas de precisão e *recall* na identificação de clones do Tipo 1 e do Tipo 2, utilizando o apoio computacional desenvolvido. A `LPS TW` foi escolhida como oráculo. O oráculo, nesse contexto, representa um conjunto de dados de referência inalterada da LPS composta por todas as *features* e seus refinamentos. A partir dele, são construídos cenários de testes desejados para melhor realizar a avaliação onde são inseridos os trechos manipulados e utilizados como trechos de clones do Tipo 1 e do Tipo 2 para realizar a avaliação interna.

[12] https://pmd.github.io/latest/pmd_userdocs_cpd.html

5.2.1 Como Mensurar a Precisão

A primeira parte do processo de avaliação interna consiste em mensurar qual a precisão da detecção de código clonado realizada pelo apoio computacional desenvolvido. A precisão é uma medida que determina o grau de proximidade das medições de uma quantidade (resultados encontrados) em relação ao valor real dessa quantidade. No contexto de detecção de código clonado, um valor de precisão alto pode ser relacionado a capacidade de detectar maior quantidade de clones de código.

O cálculo da precisão (***p***) da detecção de código clonado é dado pela razão entre a quantidade de clones detectados corretamente (***cC***) e a quantidade de clones detectados (***cD***) [Dang; Wani, 2015]. Assim sendo, o valor de ***p*** é calculado por meio da seguinte fórmula

$$\boldsymbol{p} = \frac{\boldsymbol{cC}}{\boldsymbol{cD}}$$

O valor de ***cD*** é determinado com base na execução da ferramenta e na coleta da quantidade total de código clonado identificado. Dessa quantidade, é obtida a quantidade que realmente se caracteriza como um clone do Tipos 1 e do Tipo 2 (***cC***). A quantidade de clones detectados corretamente requer a análise manual de cada trecho visando ao aumento da identificação de falsos positivos. Logo, quanto mais distante (menor) for o valor de ***cC*** do valor de ***cD***, menor é a precisão.

5.2.2 Calculando a Precisão

Ao executar o apoio computacional desenvolvido sobre a `LPS TW`, foram encontrados o total de 160 pares de métodos clonados sendo eles 9 pares do Tipo 1 e 151 pares do Tipo 2. Todos os métodos foram analisados quanto ao seu Tipo manualmente após a identificação pela análise do apoio computacional desenvolvido para confirmar se continham de fato as características para enquadrarem-se nos tipos analisados e diminuíssem a quantidade de falsos positivos identificados. Nessa análise, foi constatado que os 160 pares de clones identificados correspondiam de fato a pares de clones dos tipos identificados pela ferramenta. Substituindo na fórmula da precisão os valores ***cD* = 160** e ***cC* = 160**, tem-se:

$$\boldsymbol{p} = \frac{\mathbf{160}}{\mathbf{160}} = \mathbf{1}\ \boldsymbol{x}\ \mathbf{100} = \mathbf{100}\%$$

Como o valor da precisão indica qual o grau de proximidade da medição em relação ao seu valor real, tem-se valor de precisão igual a 1, que, em porcentagem, refere-se ao máximo de precisão (100%) na detecção de clone do Tipo 1 e do Tipo 2 pelo apoio computacional desenvolvido. Analisando individualmente os valores contabilizados em relação a seu tipo, na `LPS TW`, tem-se proporção de ocorrências de 1 clone do Tipo 1 a cada 16,77 clones do Tipo 2, o que equivale ao total de 5,62% de clones do Tipo 1 e 94,37% dos clones sendo do Tipo 2 encontrados na `LPS TW`. Com precisão de 100% nos resultados analisados, observa-se que a frequência de ocorrências da detecção de clones do Tipo 2 foi maior em relação aos clones do Tipo 1 na implementação da `LPS TW`.

Como clones do Tipo 2 são cópias com alterações em comentários, *layout*, literais, identificadores e tipos e a implementação da `LPS TW` é, em sua maioria, refinamentos de métodos, observa-se que esses refinamentos têm sido feitos por cópias com pequenas alterações nos métodos de outras *features*. Em contra partida, pode-se observar que a baixa ocorrência de clones do tipo um em métodos, isto é, cópias literais de métodos que possuem semelhança, tem baixa frequência de ocorrência.

5.2.3 Como Mensurar o *Recall*

A outra parte destinada ao processo de avaliação interna consiste em mensurar qual é o *recall* da detecção de código clonado encontrado com a utilização do apoio computacional desenvolvido. O *recall* é uma medida que determina a capacidade de encontrar a maioria dos clones existentes no sistema após a busca. No contexto de detecção de código clonado, um valor de *recall* alto pode ser relacionado à confiabilidade e à credibilidade na escolha da ferramenta de detecção de clones pelos usuários interessados, considerando que ela detecta grande parte da quantidade total de trechos de código classificados como clonados.

O cálculo do recall ($\boldsymbol{r}$) na detecção de código clonado é dado pela razão entre a quantidade de clones detectados corretamente ($\boldsymbol{cC}$) e a quantidade de possíveis clones existentes ($\boldsymbol{cE}$) [Dang; Wani, 2015]. Assim sendo, o valor de r é calculado por meio da seguinte fórmula:

$$\boldsymbol{r = \frac{cC}{cE}}$$

O valor de $\boldsymbol{cC}$ é determinado pela contabilização das coincidências identificadas nos clones de $\boldsymbol{cE}$, após a utilização do apoio computacional desenvolvido e obtenção de seus

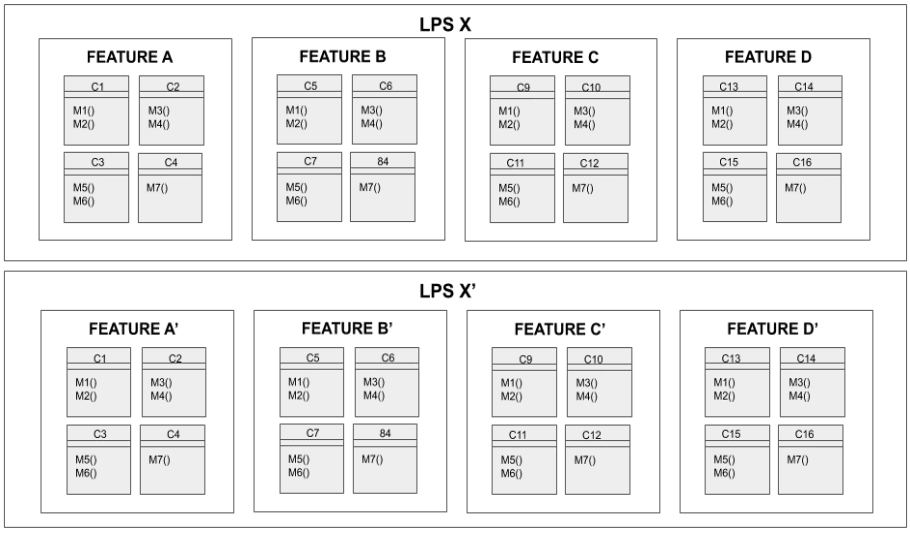
resultados. Ou seja, o valor da interseção dos conjuntos de clones de $\boldsymbol{cE}$ e o conjunto de clones identificados pela ferramenta dado por:

$$\boldsymbol{cC} = \{\boldsymbol{cE}\} \cap \{\boldsymbol{clones\ identificados\ pelo\ apoio\ computacional}\}$$

O valor de $\boldsymbol{cE}$ é determinado com base na quantidade de cópias de trechos de código (clones) manipulados. Essa manipulação pode ser justificada pelo fato de que o objetivo deste estudo é encontrar clones de código do Tipo 1 e do Tipo 2 (levando em consideração suas características descritas na literatura) e a abordagem proposta ser dividida em passos onde a finalidade é determinar se ela trouxe benefícios em sua escolha e na aplicação desses passos na identificação desses tipos de clones.

Para obter tal valor, o primeiro passo foi utilizar uma LPS como oráculo e manipular cópias de trechos de códigos para construir uma LPS derivada onde seria possível saber exatamente a quantidade de código clonado existente na LPS. Como o objetivo é detectar clones na `LPS TW`, foi criada uma LPS derivada (cópia da LPS original acrescida de alterações nas *features*) chamada `LPS TW'` utilizada neste trabalho como auxílio para realizar a avaliação interna. A `LPS TW'` é exatamente uma cópia da `LPS TW` acrescida de algumas outras cópias de alguns métodos de refinamentos de *features*. Tal alteração pode ser observada na ilustração da Figura 5.1. Nessa figura, é mostrado, por meio de um exemplo, que a `LPS X'` é derivada da `LPS X` contendo exatamente suas respectivas *features*, classes e métodos.

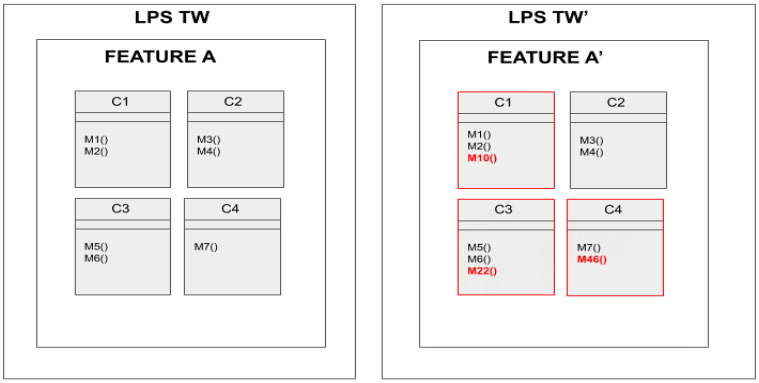
Figura 5.1 - Derivação de LPS.



Fonte: Do autor (2020)

Na Figura 5.2, a `FEATURE A` e a `FEATURE A'` da `LPS A` e da `LPS A'`, respectivamente, possuem os métodos idênticos. Entretanto, na `FEATURE A'`, são destacadas as classes `C1`, `C3` e `C4` acrescidas de cópias dos métodos `M10()`, `M22()` e `M46()` que, por sua vez, foram copiados intencionalmente de outras *features* da `LPS TW'` para representarem clones e, posteriormente, serem alvo da detecção de clones pela ferramenta de detecção de clone desenvolvida.

Figura 5.2 - `LPS TW` x `LPS TW'`.



Fonte: Do autor (2020)

Os métodos `M10()`, `M22()` e `M46()` (Figura 5.2) após copiados de outras *features*, foram posteriormente modificados (ou não), dependendo da necessidade de sua caracterização, a fim de se caracterizarem clones dos tipos procurados (Tipo 1 e Tipo 2). No caso do clone do Tipo 1 (clones idênticos), não foram necessárias alterações visto que a prática de copiar e colar esses métodos de uma classe para outra sem realizar alterações em seu conteúdo caracteriza o clone como clone idêntico. Para clone do Tipo 2, algumas alterações intencionais foram feitas para se caracterizarem como alterações em literais e variáveis, por exemplo, dentro do conteúdo (*body*) dos métodos.

O próximo passo foi determinar a quantidade de clone de código que a LPS derivada deveria ter para calcular o *recall*. A ação de manipular intencionalmente os trechos de código ocorreu de forma manual e lenta. Foram realizadas pesquisas na literatura a fim de encontrar trabalhos que fizeram pesquisas que demonstrasse/estimasse qual seria o percentual ideal de código a ser replicado ou apresentasse alguma metodologia de como obter esse resultado:

a) Em um trabalho [Couto, 2018] , com o objetivo de construir uma ferramenta que auxilia na tomada de decisão arquitetural com base na movimentação de métodos utilizando medidas de qualidade de software e, consequentemente, avaliar sua eficácia, uma das avaliações realizadas foi a estratégia de utilizar um sistema de software como oráculo e a criação de

um sistema derivado onde os métodos foram movidos aleatoriamente para outras classes a fim de calcular o valor de *recall* da ferramenta proposta. Tal estratégia pode ser utilizada para calcular o *recall*.

b) Em outro trabalho [Dang; Wani, 2015], o objetivo foi efetuar avaliações por meio da comparação de diferentes ferramentas de detecção de clone de código em sistemas orientados a objetos onde foram escolhidas quatro ferramentas, considerando os tipos de clones que eles identificavam para analisar valores de precisão e de *recall*. As quatro ferramentas escolhidas identificam clones do Tipo 1 e do Tipo 2. O método utilizado para identificar a quantidade total de clones existente no cálculo de recall foi considerar que todo o código, em linhas de código (*Line of code* - LOC), representava trechos de clones de código.

Tendo em vista os dois trabalhos analisados, que consideraram o cálculo de *recall*, mais especificamente o valor de $\boldsymbol{cE}$ por meio da utilização do oráculo, o procedimento seguido foi baseado nos dados obtidos no segundo trabalho [Dang; Wani, 2015].

Nesse trabalho, foram utilizadas as ferramentas `PMD`, `CCFinderX`, `CP-Miner` e `Bahaus` para calcular o *recall*. Os valores obtidos foram entre 0,59 (maior valor) e 0,48 (menor valor). Assim sendo, foi considerado o maior valor de *recall* (0,59) para representar o melhor valor de *recall* obtido por uma ferramenta de detecção de código clonado cujo objetivo é encontrar clones de código do Tipo 1 e do Tipo 2. Ao substituir os valores na fórmula, tem-se:

$$\mathbf{0,59} = \frac{\mathbf{957}}{\boldsymbol{cE}} \Rightarrow \boldsymbol{cE} = \frac{\mathbf{957}}{\mathbf{0,59}} \Rightarrow \boldsymbol{cE} = \simeq \mathbf{1.622,03}$$

Com o valor de $\boldsymbol{cE}$ obtido, é conclusivo que, para esse caso, a quantidade de possíveis clones existentes é 1.622 linhas de código. Isto é, para o total de 13K LOC analisadas em um programa, a ferramenta PMD consegue inferir que 12,47% de um sistema de software é geralmente classificado como clone de código. Como o objetivo é identificar o percentual adequado de trechos de clones a serem copiados para realizar a avaliação do valor de *recall*, foi utilizado um percentual de cópias de clones de aproximadamente 13%. Para utilizar tal valor, foi considerado o estudo realizado sobre ferramentas com características semelhantes a esse trabalho (identificação de clones do Tipo 1 e do Tipo 2) onde a melhor ferramenta identifica que o total de 12,47% de código é clonado em sistemas de software.

5.2.4 Calculando o *Recall*

A `LPS TW` possui 88 classes distribuídas em 31 *features* e seus refinamentos com total de 5.807 linhas de código (incluindo quebras de linhas e comentários, isto é, em sua formatação original). Assim sendo, para o cálculo da precisão, o total de aproximadamente 724,13 (=12,47% de 5.807) linhas de código devem ser clonadas para obter o valor da precisão da detecção de clone pela ferramenta desenvolvida. Como os trechos de clones precisam representar um cenário de caracterização do trecho de código em relação a classificação do seu tipo e essa abordagem trata especificamente de análise de clones em métodos, as linhas de código foram inseridas em forma de novos métodos em algumas *features*, para cobrir os seguintes cenários de análise:

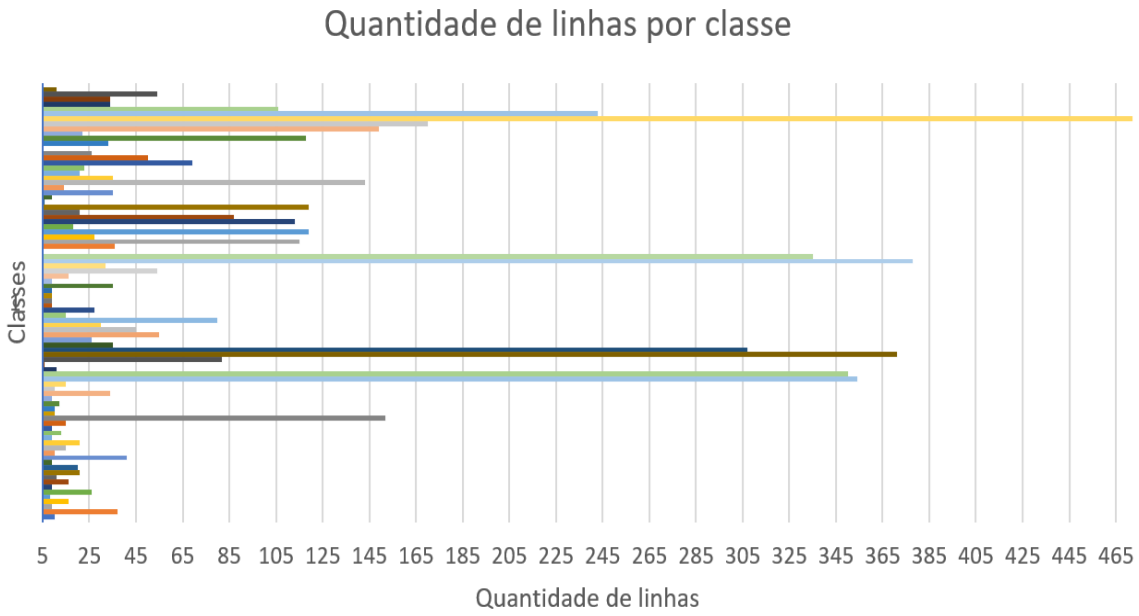
a) C1: Métodos na mesma *feature* e mesma classe;
b) C2: Métodos na mesma *feature* e classes diferentes;
c) C3: Métodos em *features* diferentes e mesma classe;
d) C4: Métodos em *features* diferentes e classes diferentes;

Na Figura 5.3, estão representadas as classes da `LPS TW` e suas respectivas quantidades de linhas (tamanho em LOC). Pode-se observar a concentração das barras no eixo vertical onde a maioria dessas classes possui entre 5 e 25 linhas de código. Considerando os dados exibidos nessa figura, algumas características observadas na própria `LPS TW` (por exemplo *layout*, quebra de linhas e importações) e características da linguagem Jakarta em relação ao refinamento de *features*, foram criados e clonados métodos de 15 linhas de código, totalizando 725 linhas de código[13] clonados (equivalente a aproximadamente 48 métodos). Para representar os clones do Tipo 1, foi utilizado como base o método `testeX()` (Código 5.1).

A `LPS TW`' foi acrescida de 24 clones do método `testeX()`, sendo X substituído pelos números de 1 a 24 para representar métodos de clone do Tipo 1 em que cópias literais desses métodos foram inseridas em *features* aleatórias na `LPS TW`'. Para representar os clones do Tipo 2, 24 clones do método `testeY()`, sendo Y substituído pelos números de 25 a 48, foram inseridos na `LPS TW'` . Para representar os clones do Tipo 2, foi utilizado como base o método `testeY()` (Código 5.2).

[13] Arredondamento de 724,13 (=12,47% de 5.807) linhas de código.

Figura 5.3 - Quantidade de Linhas de Código por Classe na `LPS TW`.



Fonte: Do autor (2020)

Código 5.1 - Método `testeX()`.

```
1.  public void testeX() {
2.      Super().helpItemErstellen();
3.      menu.add(Sprach.HelpItem, loadImage("transparent.png",a,0),loadImage("Bombe.png
    ",0,0), 0);
4.      teste3();
5.      int a = 2;
6.      if (tankManager.status == GameManager.PAUSE || tankManager.status == GameManage
    r.EXIT) {
7.          if (beschleunigung) {
8.              beschleunigungTimer += elapsedTime;
9.          }
10.     }
11.     if (beschleunigung && System.currentTimeMillis() - beschleunigungTimer > 1) {
12.         geschwindigkeit -= 3;
13.         beschleunigung = false;
14.     }
15.  }
```

Fonte: Do autor (2020)

Código 5.2 - Método `testeY()`.

```
1.  public void testeY() {
2.  Super().helpItemErstellen();
3.      menu.add(Sprach.HelpItem, loadImage("black.png",a,0),loadImage("fuer_PC.png",0,
    0), 5);
4.      teste3();
5.      int a = 6;
6.      if (tankManager.status == GameManager.PLAY || tankManager.status == GameManager
    .PAUSE) {
```

(continua)

Código 5.2 - Método `testeY()`. (cont.)

```
7.          if (beschleunigung) {
8.              beschleunigungTimer += elapsedTime;
9.          }
10.     }
11.     if (beschleunigung && System.ucurrentTimeMillis() - abeschleunigungTimer > 899)
    {
12.         geschwindigkeit1 -= 20;
13.         beschleunigung2 = false;
14.     }
15. }
```

Fonte: Do autor (2020)

Como exemplo, foram feitas 15 possíveis combinações de clones com os métodos `teste1()`, `teste2()`, `teste3()`, `teste25()`, `teste26` e `teste27()` para avaliar os dados obtidos. Optou-se por utilizar menos métodos (15 métodos) do que o total (48 métodos) por causa da quantidade de combinações de 1.680 pares de clones (552 do tipo 1 e 1.128 do Tipo 2) ser muito alta para listar neste texto. Contudo, foram analisadas todas as combinações. O cálculo dessa quantidade de itens foi dado pela soma de T1 e T2, sendo T a fórmula da combinação T = $(C_{n,2})$, sendo n a quantidade total de métodos analisados:

a) **Para o clone do Tipo 1 (T1)**: combinação de 24 métodos (pertencentes ao método `testeX()`) agrupados dois a dois acrescidos de combinação de 24 métodos (pertencentes ao método `testeY()`) (n = 24):

$$\text{T1} = \left(C_{n,2}\right) + \left(C_{n,2}\right) = 2 * \left(C_{n,2}\right) = 2 * \left(C_{24,2}\right) = 2 * \frac{24!}{2! * (24-2)!} = 2 * 276 = 552$$

b) **Para o clone do Tipo 2 (T2)**: combinação de 48 métodos (método `testeX()` = 24 métodos e método `testeY()` = 24) agrupados dois a dois (n = 48):

$$\text{T1} = \left(C_{n,2}\right) = \left(C_{48,2}\right) = \frac{48!}{2! * (48-2)!} = 1.128$$

Na Tabela 5.1, é apresentada a relação dos cenários cobertos por cada uma das 15 combinações de métodos para o cálculo do *recall*, os clones que caracterizam cada combinação e se foi identificado corretamente pela ferramenta ou não (determina o valor de $\boldsymbol{cC}$). Para facilitar a visualização de cada cenário, os métodos foram representados nessa tabela por meio da composição da classe, *feature* e nome do método analisado.

Para mensurar a confiabilidade e a credibilidade da utilização do apoio computacional desenvolvido por meio do valor de *recall*, sendo a quantidade de possíveis clones ($\boldsymbol{cE}$) igual a 1.680 e a quantidade de clones detectada corretamente ($\boldsymbol{cC}$) igual a 1.680, tem-se a substituição dos valores na fórmula e o valor de **r**:

$$r = \frac{1.680}{1.680} = 1\ x\ 100 = 100\%$$

Tabela 5.1 - Relação de Combinações de Clones Identificados para o Cálculo do *Recall*.

#	Cenário	Classe$$Feature.método1	Classe$$Feature.método2	Tipo de Clone	Tipo de Clone Detectados Corretamente (cC)
1	C4	Sprach$$DE.teste1	Maler$$einfrieren.teste2	1	sim
2	C4	Sprach$$DE.teste1	Tank$$Energie.teste3	1	sim
3	C3	Tank$$Beschleunigung.teste2	Tank$$Energie.teste3	1	sim
4	C4	Maler$$Beschleunigung. teste25	Tank$$Bombe. teste26	1	sim
5	C3	Maler$$Beschleunigung. teste25	Maler$$ChinaType99. teste27	1	sim
6	C1	Tank$$Bombe. teste26	Tank$$Bombe. teste27	1	sim
7	C4	Sprach$$DE.teste1	Maler$$Beschleunigung. teste25	2	sim
8	C4	Sprach$$DE.teste1	Tank$$Bombe. teste26	2	sim
9	C4	Sprach$$DE.teste1	Tank$$Bombe. teste27	2	sim
10	C2	Tank$$Beschleunigung.teste2	Maler$$Beschleunigung. teste25	2	sim
11	C3	Tank$$Beschleunigung.teste2	Tank$$Bombe. teste26	2	sim
12	C3	Tank$$Beschleunigung.teste2	Tank$$Bombe. teste27	2	sim
13	C4	Tank$$Energie.teste3	Maler$$Beschleunigung. teste25	2	sim
14	C3	Tank$$Energie.teste3	Tank$$Bombe. teste26	2	sim
15	C3	Tank$$Energie.teste3	Tank$$Bombe. teste27	2	sim

Fonte: Do autor (2020)

Ou seja, para o teste executado, foi obtido um valor 1, que em porcentagem significa o total de 100% dos casos cobertos de maneira assertiva. Assim sendo, pode-se concluir que a utilização do apoio computacional desenvolvido para a detecção de clones tem alta confiabilidade e credibilidade nos resultados em que apresenta. Tais resultado podem ser melhor compreendidos quando se observa que esse apoio computacional cobre os quatro possíveis cenários em que o clone de um método pode existir durante o desenvolvimento de uma LPS. Mesmo que, de modo funcional, os quatro cenários analisados não aconteçam na prática, pois só é possível refinar métodos que existam na classe principal, a detecção de clones proposta por essa ferramenta tem por objetivo identificar clones na LPS e não apenas em produtos da LPS.

5.3 Avaliação Externa

De maneira semelhante à avaliação interna, objetivando avaliar os resultados obtidos após o desenvolvimento da proposta de abordagem, foi realizada a avaliação externa do apoio computacional desenvolvido. Essa avaliação consistiu em comparar os resultados obtidos com a ferramenta `PMD`.

5.3.1 Escolha da Ferramenta

Para comparar os resultados obtidos por duas ferramentas distintas, deve-se ter cuidado ao escolhê-las. Assim, inicialmente, foram buscadas as ferramentas listadas como as frequentemente encontrada nos artigos analisados [Batista *et al.*, 2019]. Entretanto, por motivos de não encontrar ferramentas disponíveis para uso por causa de problemas técnicos como falta de atualização e suporte para versões do Eclipse IDE e necessidade de execução de outros complementos/ferramentas para obtenção dos dados, não foi possível utilizá-las para a avaliação. Assim sendo, optou-se por realizar testes com a ferramenta `PMD` utilizada no processo de avaliação da ferramenta de detecção de clones desenvolvida em um dos trabalhos relacionados [Paiva; Figueiredo, 2016] e utilizada para coleta de informações sobre valores de precisão e recall em clones de código [Dang; Wani, 2015] (Seção 5.2).

O `PMD` é uma ferramenta gratuita de análise de código fonte utilizada para encontrar falhas comuns de programação por meio da disponibilização de *plug-ins* e inclui o `CPD` para as linguagens Java, C, C ++, C #, Groovy, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Scala, Objetivo C, Matlab, Python, Go, Swift e Salesforce.com Apex e Visualforce. Sua instalação no Eclipse IDE pode ser feita diretamente via Eclipse Marketplace ou pela opção de instalar novo software.

Para a análise funcionar utilizando `PMD`, foi necessária a manipulação da LPS para analisar código Java (as *features* são implementadas na linguagem Jakarta), uma das opções de análise disponibilizadas por `CPD` do `PMD` (o arquivo de configuração da LPS disponibiliza um arquivo Java com a configuração selecionada). Essa manipulação foi feita utilizando do pré-processamento descrito na seção 4.3 (Abordagem) onde o código da LPS é manipulado por meio da seleção forçada de todas as *features* do modelo, sem considerar suas restrições, o que ocasiona quebra conceitual da LPS na geração de produtos, porém satisfaz a condição de que sejam procurados clones de código em toda a LPS. Antes de executar `CPD` do `PMD` na `LPS TW`, foi executado o apoio computacional desenvolvido para criar o arquivo de configuração da LPS. Esse arquivo de configuração foi analisado pelo `PMD` no diretório `src` criado no projeto da `LPS TW` no Eclipse IDE (esse diretório é o diretório em que o `PMD` realiza as análises de código fonte nos projetos no Eclipse IDE).

5.3.2 Realizando a Avaliação Externa

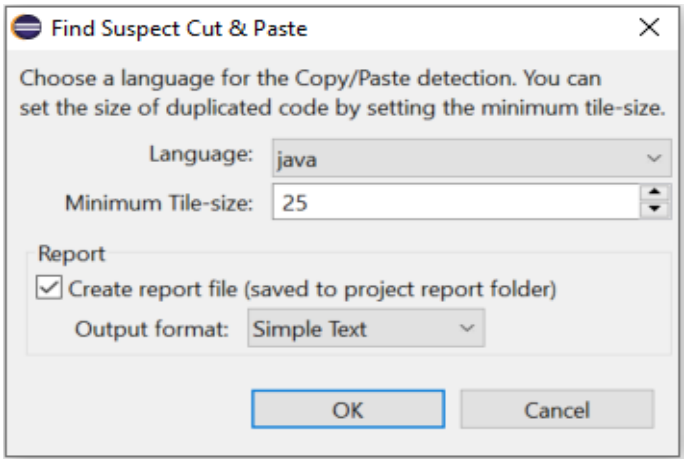
A avaliação externa foi realizada na LPS TW como informado anteriormente. Após garantir que a LPS TW tenha o arquivo de configuração contendo todas as *features* para procurar clones de código em toda a LPS, foi necessário elaborar uma estratégia para analisar os resultados encontrados pelo apoio computacional desenvolvido e PMD, pois utilizam técnicas diferentes para detecção dos clones de código e, consequentemente, da forma de coleta dos mesmos.

A técnica de análise e detecção de clones de CPD do PMD é baseada em *token* que identifica trechos de clones idênticos (clone do Tipo 1). Para utilizar a fermenta após configura-la no Eclipse IDE via interface, é necessário selecionar a linguagem que deseja analisar e definir o tamanho do código duplicado, ou seja, a quantidade mínima de *tokens* que deve existir em um bloco de código (definido em quantidade de linhas) para esse bloco ser considerado clone. O resultado obtido na análise realizada pode ser melhor avaliado pelo *report* gerado onde os blocos de código analisado e suas respectivas quantidades de *token* encontradas são acompanhados do nome da classe de refinamento de *feature* da LPS. A mesma geração do *report* pode ser feita via linha de comando, seguindo as orientações disponibilizadas para auxiliar na documentação e uso do CPD[14].

Como exemplo de utilização de CPD do PMD, via interface, para os parâmetros de entrada *Language* e *Minimum Tile-size*, foram selecionados a linguagem *java* e o valor 25, respectivamente (Figura 5.4). Para o resultado de saída, foram encontrados vários blocos de código agrupados por quantidade de linhas, quantidade de *tokens* e arquivos em que eles foram encontrados. Na Figura 5.5, é apresentada a estrutura do arquivo de saída gerado na detecção de clone que contém a quantidade de linhas no bloco de clone identificado, a quantidade de clones identificada, a linha do arquivo e o arquivos onde foram encontrados tais trechos bem como o próprio trecho de código que caracteriza o clone encontrado. Assim sendo, foram encontradas 3 linhas idênticas (clones com 26 *tokens coincidentes*) em 7 trechos de código iniciados em linhas diferentes. Essa estrutura se repete por todo *report* gerado, mostrando todos os clones encontrados que contém desde a quantidade mínima de *tokens* informada em diante (no caso do exemplo são gerados todos os blocos de código que conseguirem ser formados com 25 *tokens* idênticos em um trecho idêntico, desde que eles sejam iniciados em pelo menos duas linhas diferentes) para o projeto analisado.

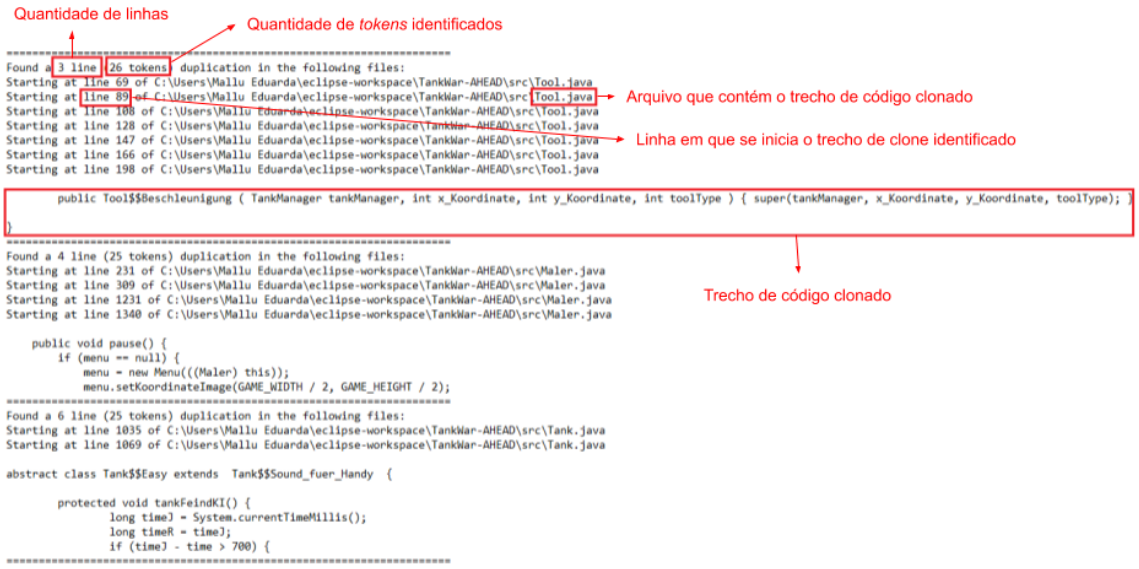
[14] https://pmd.github.io/latest/pmd_userdocs_installation.html

Figura 5.4 - Executando CPD do PMD.



Fonte: Do autor (2020)

Figura 5.5 - Resultado da Análise de Clone em CPD do PMD.



Fonte: Do autor (2020)

5.3.3 Descrição da Análise Realizada

Após avaliar o processo de detecção que o CPD do PMD a avaliação externa dos resultados foi feita considerando a quantidade de clones identificada, baseado na quantidade de linhas que eles possuem, e discutindo sobre esses resultados também no contexto qualitativo. Ao todo, CPD conseguiu identificar o total de 3.795 clones de código em seu arquivo de *report* gerado ao analisar a LPS TW. Na Tabela 5.2, esses clones foram separados por tamanho (em LOC). Assim sendo, foram identificados 20 grupos diferentes de clones de código classificados por tamanho, em relação a quantidade de clones com aqueles tamanhos.

Ao observar a quantidade de clones identificados em relação ao tamanho em LOC desses clones (Tabela 5.2), é possível ver queda quase linear na quantidade de clones quando o tamanho desses clones aumenta em quantidade de linhas. Tal fenômeno pode ser justificado pela análise realizada por `CPD` se tratar de trechos de código idênticos (clones do Tipo 1) e um mesmo trecho de código pode ser (e foi) encontrado em vários desses clones, por exemplo, um clone de tamanho 1 pode compor os clones de tamanho 2 e clones de tamanho 2 podem compor clones de tamanho 3, assim por diante. Essa análise mostra que, em LPS's, a prática de copiar e colar trechos de código, isto é, introduzir clones do Tipo 1, é frequentemente realizada com quantidade menor de linhas de código.

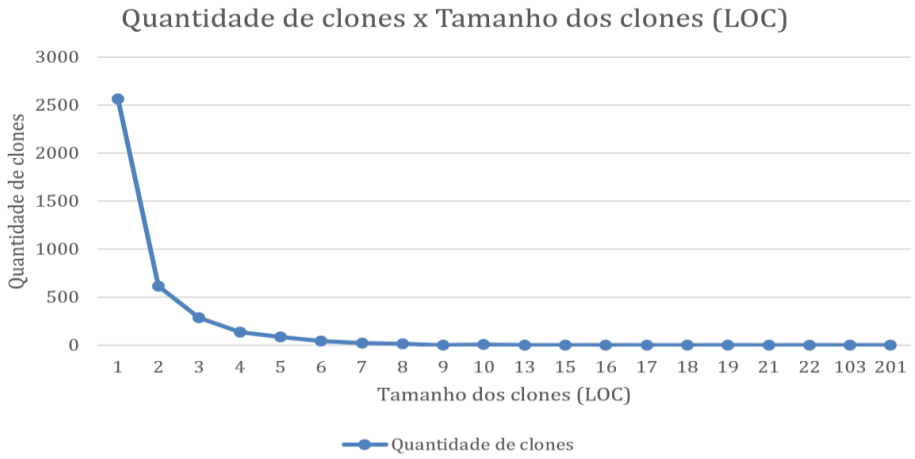
Tabela 5.2 - Tamanhos dos Clones Identificados pelo `CPD` na `LPS TW`.

Tamanho do Clone (LOC)	Quantidade de Clones	Tipo do Clone
1	2.568	1
2	615	1
3	285	1
4	133	1
5	86	1
6	41	1
7	23	1
8	16	1
9	3	1
10	8	1
13	1	1
15	2	1
16	1	1
17	2	1
18	2	1
19	1	1
21	2	1
22	1	1
103	1	1
201	1	1
Total	3.795	

Fonte: Do autor (2020)

Diante disso, pode-se observar uma desvantagem de analisar trechos de código idênticos considerando a quantidade de linhas, visto que um mesmo clone (trecho pequeno) pode ser identificado várias vezes em uma única análise, mostrando uma quantidade elevada de trechos de clones na LPS que, analisando o arquivo de *report*, agrega informações apenas a nível de quantidade. No apoio computacional desenvolvido, foi considerada a relação entre as chamadas de métodos dentro de cada *feature* e os clones são identificados dentro desses métodos que, por sua vez, possuem um contexto lógico de execução do código e de uma análise da sua assinatura. Logo, conclui-se que qualitativamente, as informações coletadas pelo apoio computacional também podem auxiliar no entendimento e na análise dos clones por parte dos desenvolvedores capazes de tomar as medidas que achar necessárias em cada caso.

Figura 5.6 - Relação da Quantidade de Clones Identificados por CPD e seus Tamanhos em LOC.



Fonte: Do autor (2020)

Com relação a análise realizada pelo apoio computacional desenvolvido neste trabalho, foi possível identificar na LPS TW o total de 160 clones sendo eles 9 clones do Tipo 1 e 151 clones do Tipo 2. Na Tabela 5.3, é possível observar 5 grupos diferentes de clones de código classificados por tamanho, em relação a quantidade de clones com aqueles tamanhos.

Tabela 5.3 - Tamanho dos Clones Identificados pelo Apoio Computacional Desenvolvido na LPS TW.

Tamanho do Clone (LOC)	Quantidade de Clones	Tipo do Clone
3	18	1 e 2
4	62	1 e 2
5	1	2
6	73	2
7	6	2
Total	160	

Fonte: Do autor (2020)

Comparando os resultados obtidos, nota-se divergência no total de clones identificados, pois CPD identificou 3.795 clones e o apoio computacional desenvolvido neste trabalho identificou 160 clones. Em relação ao tipo de análise realizado e aos valores da quantidade de clones obtidos, é importante considerar que existem diferenças entre as técnicas e que os objetivos de ambas, por mais que sejam detectar clones, são voltadas para detectar clones em contextos diferentes. Assim sendo, existe diferença na maneira de agrupar e apresentar os clones em ambas as ferramentas, pois CPD trabalha com localização baseada em *tokens* e a busca por clones é agrupada em n locais diferentes em que existe um mesmo clone e o apoio computacional desenvolvido localiza pares de métodos construídos por semelhanças em chamadas de métodos e assinatura desses métodos. Por se tratarem de técnicas diferentes, é justificável a grande disparidade entre a quantidade de clones identificadas.

Em relação ao tamanho em LOC dos 20 grupos de linhas de código contendo clones, comparados com os 5 grupos de LOC identificados pela ferramenta, alguns pontos importantes precisam ser considerados ao analisar tais resultados. Realizando uma análise visual dos métodos implementados da LPS e considerando boas práticas de programação, é importante reduzir o acoplamento para o código ser mais coeso, não é interessante que os métodos de uma classe tenham "muitas responsabilidades", o que acaba tendenciosamente diminuindo o tamanho dos métodos para que eles sejam mais coesos. Tais fatos contribuem para a quantidade de apenas 5 tamanhos de clones diferentes serem identificados pela ferramenta deste trabalho.

Outro ponto a destacar na alta quantidade de clones identificados é a análise ser feita considerando trechos de código existentes em uma mesma *feature.* Porém, quando o objetivo é identificar clones existentes entre *features* de uma LPS (proposta da abordagem descrita neste trabalho), os dados não representam uma realidade acerca das características existentes na implementação das *features*, mas apenas sobre uma quantidade numérica relacionada a duplicação de código. Isso posto, é notório que a discrepância em quantidade de clones identificados está altamente relacionada aos objetivos propostos por cada abordagem durante sua identificação.

Ainda em relação ao tamanho em LOC, todos os tamanhos de clones identificados pelo apoio computacional desenvolvido foram também identificados pelo CPD. Esse fato pode ser justificado pelas considerações em relação ao tamanho médio dos métodos mencionados na avaliação interna para o calcula da precisão onde a maior frequência de tamanho de métodos na `LPS TW` é de 5 a 15 linhas de código.

Outro dado (valor) importante a ser discutido é o apoio computacional não ter detectado clone com tamanho nas faixas de 1 ou 2 linhas de código, sendo que `CPD` teve mais quantidades de clones nessas faixas (2.568 clones e 615 clones, respectivamente). Ao observar as características desses clones, em sua maioria, são linhas de código relacionadas à declaração do refinamento da classe (por exemplo, se o projeto tiver 100 classes refinadas, 100 clones seriam detectados), endentações e início de instruções que por sua vez não considera análise lógica, apenas a coincidências (a análise é baseada em *tokens*) do trecho na mesma classe ou em outras no projeto. Em contra partida, dificilmente (ou quase nunca) seriam encontrados clones nessa faixa em uma abordagem que realiza análise de métodos onde necessariamente uma das regras é ter pelo menos uma sequência de chamada de instruções (isto é, pelo menos um grafo com mais de um nível).

Em relação aos tipos de clones detectados, na Figura 5.6, percebe-se que `CPD` identificou apenas clones do Tipo 1 e, na Tabela 5.3, percebe-se que o apoio computacional desenvolvido identificou clones do Tipo 1 e Tipo 2, sendo 9 clones do Tipo 1 e, em sua maioria, 151 do Tipo 2. Ao realizar uma análise detalhada sobre a relação da quantidade de clones identificadas pelo apoio computacional desenvolvido e seus tipos (Tabela 5.4), é perceptível que a presença de clones com poucas modificações (Tipo 2) em relação a outro método é predominantemente maior do que a relação de métodos clonados na íntegra (Tipo1) quando se parte do pressuposto de que métodos que realizam computação semelhantes e possuem coincidências em sua assinatura, tem probabilidade de serem métodos clonados no contexto da LPS.

Tabela 5.4 - Relação da Quantidade de Clones Identificados pelo Apoio Computacional Desenvolvido e Tipos de Clone.

Tamanho do Clone (LOC)	Quantidade de Clones	Quantidade de Clones Tipo 1	Quantidade de Clones Tipo 2
3	18	4	14
4	62	5	57
5	1	0	1
6	73	0	73
7	6	0	6
Total	160	9	151

Fonte: Do autor (2020)

5.4 Considerações Finais

O apoio computacional desenvolvido neste trabalho propõe, por meio da técnica elaborada, implementada e descrita ao longo deste trabalho, detectar clones no contexto de desenvolvimento da LPS onde a composição de características nas *features* é uma lógica fundamental para configurar qualquer um dos produtos em uma LPS. Assim sendo, a qualidade das informações apresentadas vai muito além de uma quantidade, mas, principalmente, acerca da qualidade dos dados apresentados, possibilitando um auxílio de qualidade aos desenvolvedores de sistemas de LPS. Os valores 100% de precisão e *recall* obtidos na avaliação interna mensuram a qualidade e a confiabilidade dos dados e a escolha da utilização do apoio computacional por parte dos desenvolvedores como auxilio na detecção de clones. Na avaliação externa, foi possível observar características predominantes da abordagem em relação ao contexto aplicado, onde observou-se a alta frequência de clones do Tipo 2 na implementação de LPS que possuem métodos que realizam computação semelhantes.

6 TRABALHOS RELACIONADOS

Nesta seção, são descritos contextos de aplicação de detecção de clones em sistemas de software em conjunto ou não com Linhas de Produtos de Software, de modo a descrever a relação deste trabalho com outros existentes na literatura.

Em um trabalho [Paiva; Figueiredo, 2016], há a proposta de desenvolvimento e de avaliação de um método de detecção de clone de código baseado em sequências similares de chamada de métodos. A ferramenta desenvolvida para detecção (McSheep) é comparada com a ferramenta PMD[15] para detectar clones não identificados. Participaram de uma pesquisa 25 desenvolvedores para avaliar o método e mais de 90% deles concordam com sua validade para detecção de clones. O resultado da comparação das ferramentas mostra que ambas podem ser utilizadas de forma híbrida, pois existe um conjunto de interseção de casos detectados e, para cada ferramenta, um conjunto exclusivo. Nesse trabalho, são considerados aspectos importantes, por exemplo, a importância desempenhada por métodos em um sistema, bem como sua sequência de execução por chamada visto que tal alteração não é benéfica ao escopo do projeto, onde será obtido um resultado totalmente diferente em relação ao pretendido inicialmente.

Em outro trabalho [Schulze *et al.*, 2011], há descrição de razões para existência de clones relacionados à Programação Orientada a Características (POC) e como lidar com eles. Para isso, aplicar refatorações em clones sintáticos para remoção de clones em LPS é a abordagem proposta, visando à permanência dos benefícios de programação orientada a características como aumento da coesão e a reutilização de códigos desses recursos. Os resultados foram: i) cerca de 10% a 15% do código de LPS analisadas são clones; ii) entre 9% a 12% desses podem ser refatorados com base na sintaxe; iii) mais ou menos 9% dos clones são relacionados a POC e 2% a 3% relacionados a POO; e iv) nas etapas de análise, a quantidade de clones é maior em LPS desenvolvidas do zero do que em LPS decompostas de sistemas legados. Assim observados, nesse trabalho de mestrado, foi considerada a identificação de percentual significativamente alto que pode impactar na manutenção de LPS, bem como a relacionatividade com o paradigma orientado a características.

[15] PMD é uma ferramenta de análise que "varre" código Java e dispõe da implementação da análise CPD (Copy/Paste Detector), que detecta código duplicado.

Em outro trabalho [Roy, 2009], foi proposta uma série de evoluções a respeito da tratativa sobre análise e detecção de clones. Proposta em cinco etapas concretas, tal tratativa propõe: i) desenvolvimento de uma abordagem híbrida de detecção de clones chamada NICAD focada na detecção de clones exatos e com trocas mínimas (Tipo 1 e Tipo 2); ii) taxonomia para a definição de clones; iii) comparação qualitativa a respeito de técnicas e ferramentas de clonagem disponíveis visando melhorar a técnica híbrida implementada em NICAD; iv) desenvolvimento de uma estrutura baseada em mutações que mede automaticamente a recuperação e a precisão de ferramentas de detecção de clones para diferentes tipos de clones refinados da taxonomia de edição proposta; e v) realização de um estudo empírico da clonagem de código em softwares abertos para avaliar o NICAD e estudar as características de clonagem desses sistemas em dimensões diferentes.

Assim como os trabalhos citados, esta investigação propõe a detecção de clones de código por meio da utilização de análise estática de código fonte. Mas, há diferenças a serem consideradas, tais como, o método abordado de como é feita a detecção de clones em paradigmas diferentes (programação orientada a objetos e programação orientada a características), propor uma abordagem diferente das atuais considerando aspectos da modelagem do sistema em relação a função desempenhada pelos métodos. Diferente das demais propostas, (semi) automatizar a abordagem definida por meio de um apoio computacional para detectar clones diretamente no código fonte do modelo de características da LPS e analisar os resultados dessa abordagem com relação à manutenibilidade, analisando seu desempenho e precisão em relação a outras ferramentas e os tipos de clones existentes.

7 AMEAÇAS A VALIDADE

A validade é uma medida na qual uma conclusão é bem fundamentada de acordo com a concepção e a análise de um experimento [BARROS *et al.*, 2011]. As ameaças a validade são organizadas em quatro tipos:

a) **Validade Interna**. É a relação observada entre os fatores que afetam o processo de experimentação. As principais limitações são referentes à construção da *string* de busca, construção da técnica utilizada e os tipos de clones detectados. A *string* de busca pode conter falhas ou estar incompleta para o contexto podendo gerar alteração no resultado obtido nas buscas nas bibliotecas digitais utilizadas, entretanto sua construção passou por vários processos de refinamento durante um mês, para que se chegasse na *string* final. A técnica híbrida abordada poderia ter sido composta por outras e outra quantidade de técnicas diferentes, porém optou-se por escolher tais técnicas baseada em bons resultados obtidos por outro trabalho e pelo contexto de construção de assinaturas de métodos, ambos voltados ao contexto e granularidade de métodos sendo assim a quantidade suficiente para se obter bons resultados. Por fim, outros tipos de clones poderiam ter sido implementados, o que foi limitado pelas características e as dificuldades na implementação dos demais tipos de clones limitaram a apenas o Tipo 1 e o Tipo 2, entretanto na literatura foi possível observar que esses clones aparecem com mais frequência do que o Tipo 3 e o Tipo 4;

b) **Validade Externa**. Consiste na capacidade de generalizar a descoberta. A coleta e a análise dos dados foram feitas em apenas uma LPS e em relação a apenas uma fermenta de detecção de clones. Outras LPS poderiam ser escolhidas para realizar a análise bem como outras ferramentas de detecção de clones, uma vez que existem repositórios de LPS desenvolvidas com finalidades acadêmicas e na literatura outras ferramentas são mencionadas para detectar clones. Porém, foram encontradas dificuldades na disponibilização e na instalação dessas fermentas, pois, a maioria possui dependências de execução;

c) **Validade de Construção**. Consiste em considerar se os fatores que afetam o processo de experimentação refletem na causa e o resultado reflete no efeito. Vários trabalhos analisados não fundamentam completamente (ou nenhuma vez) como/com base em que são as técnicas de detecção utilizadas bem como os tipos de clones identificados pela ferramenta/abordagem criada e os dados acerca da linguagem e do paradigma de programação utilizada são uma lacuna a ser preenchida com informações não contidas nos trabalhos selecionados, porém várias características desses trabalhos foram respondidas como questões de pesquisa afim de aumentar o nível de informação. O apoio computacional

para detecção de clones foi desenvolvido por apenas um pesquisador, entretanto durante esse processo várias validações e casos de testes foram feitos entre ambos. As medidas precisão e *recall* consideram análises manuais de visualização e manipulação dos resultados, respectivamente, o que está propenso à falhas porém ainda é uma forma eficaz de se obter as medidas utilizadas, a gama de dados a serem analisados não é muito extensa e as características a serem analisadas segundo o tipo são pequenas e as dificuldades em encontrar semelhanças para análise das ferramentas leva a uma análise mais genérica e não pontual do conteúdo dos métodos obtidos. Porém, foi possível realizar análises de características da LPS e seus tipos devido ao temo e a complexidade dos dados existentes na LPS escolhida;

d) **Validade de Conclusão**. Relacionada à capacidade de alcançar uma conclusão correta considerando a relação entre os fatores que afetam o processo de experimentação e o resultado obtido no experimento. No MSL os relacionamentos encontrados entre as linguagens de programação, tipo de clone e paradigma de programação podem conter falhas não percebidas durante as conclusões tomadas, porém a análise manual foi revisada algumas vezes antes da contabilização. Neste trabalho, são analisados os resultados em relação à capacidade de encontrar clones do Tipo 1 e Tipo 2. Os valores das medidas precisão e *recall* são mensurados em relação a um resultado esperado e isso induz refinamentos na técnica até obter um valor ideal alto. Entretanto, como dizem respeito à assertividade e à confiança de utilização do apoio computacional, tais refinamentos se fazem necessários. A relação entre duas ferramentas, quando analisadas, precisam considerar a técnica utiliza em cada uma o que reduz a qualidade dos dados obtidos quando comparados, pois, cada uma tem o foco em escopo de detecção, como análises de métodos ou classes, por exemplo. Porém, é necessária a análise para ter uma visão do ambiente estudado composto por possibilidades de escolhas de outras ferramentas.

8 CONSIDERAÇÕES FINAIS

8.1 Conclusão

Este trabalho propôs uma abordagem híbrida para detecção de código clonado em LPS baseada em sequência de chamadas de métodos para detectar clones do Tipo 1 e Tipo 2. Clones do Tipo 1 são conhecidos por serem cópias literais de trechos de código com alterações em comentários e espaçamentos, em que é comum a prática de copiar e colar. Clones do Tipo 2 são identificados por cópias de trechos com pequenas alterações em literais, identificadores, *layouts* e comentários.

Para identificar esses dois tipos de clones, foi proposta uma abordagem híbrida composta por etapas cujo objetivo é analisar métodos que possuem relação de chamadas a outros métodos, isto é, realizam computações semelhantes e análise de estruturas nas assinaturas desses métodos. Feito isso, o conteúdo (*body*) desses métodos são analisados para encontrar clones do Tipo 1 ou Tipo 2, seguindo suas características identificadas na literatura.

Foram identificados 160 clones de códigos na `LPS TW`, sendo 9 clones do Tipo 1 e 151 clones do Tipo 2. Além disso, mostrou-se viável e confiável a utilização do apoio computacional desenvolvido por meio da obtenção de valores das medidas precisão e *recall* de 100%. Foi realizada uma comparação entre os clones obtidos entre o apoio computacional desenvolvido e `CPD` do `PMD` onde o `CPD` identificou clones apenas do Tipo 1 e o apoio computacional identificou clones do Tipo 1 e Tipo 2. Em resumo, as características da LPS utilizada na análise mostram a viável utilização do apoio computacional principalmente em relação a frequência de clones do Tipo 2 identificados e que realmente parece ser o contexto aplicado na implementação dessa LPS.

8.2 Contribuições

A principal contribuição deste trabalho foi construir uma abordagem de detecção de clones do Tipo 1 e Tipo 2 em LPS orientada a características e um apoio computacional (semi) automatizado que implementa essa abordagem. Em resumo, as contribuições deste trabalho foram:

a) Realização de um MSL sobre o estado da arte da detecção de código clonado englobando as técnicas utilizadas, ferramentas desenvolvidas, linguagens de programação aplicadas e os frequentes paradigmas de programação em que são empregadas essas detecções;
b) Revisão *ad-hoc* da literatura sobre o contexto de LPS e a clonagem de código nesse meio;

c) O desenvolvimento de um apoio computacional (*plug-in* para o Eclipse) para detecção de clones de código do Tipo 1 e Tipo 2 para LPS orientada a características;
d) Mensuração das medidas precisão e *recall*;
e) Comparação dos clones obtidos quando utilizados o apoio computacional desenvolvido e outra ferramenta com o mesmo objetivo de detecção, mas em outro paradigma de programação.

8.3 Divulgação do Trabalho

Artigo "Uma Abordagem Híbrida para Detecção de Código Clonado em Linhas de Produtos de Software". O artigo foi aceito no XVII Workshop de Teses e Dissertações em Qualidade de Software (WTDQS 2018) do XVII Simpósio Brasileiro de Qualidade de Software (SBQS 2018).

Artigo "*An Exploratory Study on Detection of Cloned Code in Information Systems*". O artigo foi aceito no XV Simpósio Brasileiro de Sistemas de Informação (SBSI 2019).

8.4 Trabalhos Futuros

Após a realização deste trabalho foi possível observar algumas propostas de trabalhos futuros que podem acrescentar valor literário e novas descobertas no que diz respeito ao envolvimento de clonagem de código em LP Software. Dentre essas propostas destacam-se:

a) Aprofundamento nos métodos propostos durante as etapas da abordagem que por sua vez podem mensurar de uma forma melhorada os benefícios da qualidade dos dados obtidos na utilização do plug-in ao detectar clones em uma LPS;
b) Análise de desempenho para LPS escaláveis;
c) Implementação do para detecção de clones do Tipo 3 e Tipo 4;
d) Refatoração dos clones detectados, uma vez que na literatura existem trabalhos que buscam essa solução, mas muitos propõe uma outra técnica de detecção;
e) Comparação com mais ferramentas de detecção de clone visando comparações mais aprofundadas.

REFERÊNCIAS

APEL, S.; BATORY, D.; KÄSTNER, C.; SAAKE, G. Feature-Oriented Software Product Lines. 1. ed. New York, NY, USA: Springer Science & Business Media, 2013. 315 p.

BARROS, M.; NETO, A. Threats to Validity in Search-based Software Engineering Empirical Studies. Relatórios Técnicos do DIA/UNIRIO, No. 0006/2011 Editor: Prof. Sean W. M. Siqueira Abril, 2011.

BATISTA, M.; PEREIRA, A.; COSTA, H. 2019. An Exploratory Study on Detection of Cloned Code in Information Systems. In Proceedings of the XV Brazilian Symposium on Information Systems (SBSI'19). Association for Computing Machinery, New York, NY, USA, Article 67, 1-8.

COLANZI, T. E. Uma Abordagem de Otimização Multiobjetivo para Projeto Arquitetural de Linha de Produto de Software. Tese (Programa de Pós-Graduação em Informática _ Setor de Ciências Exatas, Universidade Federal do Paraná, Curitiba - PR).

CORRÊA, H. L.; CORRÊA, C. A. Administração de Produção e Operações: Manufatura e Serviços: Uma Abordagem Estratégica. [S.l.]: Editora Atlas SA, 2000.

COUTO, C. M. S. A quality-oriented approach to recommend move method refactoring. Dissertação (Mestrado em Ciência da Computação _ Departamento de Ciência da Computação) - Universidade Federal de Lavras, Lavras - MG, 2018.

DANG, S.; WANI, S. Performance Evaluation of Clone Detection Tools. International Journal of Science and Research (IJSR). n. 4, p.1903-1906, Apr 2015.

DUALA-EKOKO, E.; ROBILLARD, M. P. Tracking Code Clones in Evolving Software. In: 29th International Conference on Software Engineering (ICSE'07). [S.l.: s.n.], 2007. p. 158-167.

FAUST, D.; VERHOEF, C. Software Product Line Migration and Deployment. Software: Practice and Experience, Wiley Online Library, v. 33, n. 10, p. 933-955, 2003.

FÖRDÖS, V.; TÓTH, M. Identifying Code Clones with RefactorErl. In: Journal Acta Cybernetica, v. 22, n. 3, p. 553-571, 2016.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software. 1. ed. Boston: Ed. Pearson Education, 1994.

GAUTAM, P.; SAINI, H. Various Code Clone Detection Techniques and Tools: A Comprehensive Survey. In: SPRINGER. International Conference on Smart Trends for Information Technology and Computer Communications. [S.l.], 2016. p. 655-667.

JANG, J.; BRUMLEY, D. Bitshred: Fast, Scalable Code Reuse Detection in Binary Code (cmu-cylab-10-006). CyLab, p. 28, 2009.

KAMIYA, T.; KUSUMOTO, S.; INOUE, K. Ccfinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In: IEEE Transactions on Software Engineering, v. 28, n. 7, p. 654-670, Jul 2002. ISSN 0098-5589.

KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. [S.l.], 1990.

KITCHENHAM, D. B. B.; BRERETON, P. The Value of Mapping Studies - A Participant-Observer Case Study. In: 14th International Conference on Evaluation and Assessment in Software Engineering, British Computer Society. [S.l.: s.n.], 2010. p. 25-33.

LAGUNA, M. A.; CRESPO, Y. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. Science of Computer Programming, Elsevier, v. 78, n. 8, p. 1010-1034, 2013.

LIN, Y.; XING, Z.; XUE, Y.; LIU, Y.; PENG, X.; SUN, J.; ZHAO, W. Detecting Differences Across Multiple Instances of Code Clones. In: 36th International Conference on Software Engineering. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 164-174.

MANUAL. Joanna Briggs Institute. Joanna Briggs Institute reviewers' manual: 2014 edition. Australia: The Joanna Briggs Institute, 2014. In: Chapter Two: Qualitative protocol and title development. Link: https://nursing.lsuhsc.edu/JBI/docs/ReviewersManuals/ReviewersManual.pdf. Acessado: 18/11/2020.

MARIMUTHU, C.; CHANDRASEKARAN, K. Systematic Studies in Software Product Lines: A Tertiary Study. In: 21st International Systems and Software Product Line Conference - Volume A. New York, NY, USA: ACM, 2017.

MENDE, T.; BECKWERMERT, F.; KOSCHKE, R.; MEIER, G. Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In: European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008. [S.l.], 2008. p. 163-172.

MENDE, T.; KOSCHKE, R.; BECKWERMERT, F. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. In: Journal of Software Maintenance and Evolution: Research and Practice, Wiley Online Library, v. 21, n. 2, p. 143-169, 2009.

METZGER, A.; POHL, K. Software Product Line Engineering and Variability Management: Achievements and Challenges. In: Future of Software Engineering. New York, NY, USA: ACM, 2014. (FOSE'14), p. 70-84.

MORESI, E. Metodologia da Pesquisa. Brasília: Universidade Católica de Brasília, v. 108, p. 24, 2003.

MURDOCH UNIVERSITY. Systematic Reviews - Research Guide. 2020. Link: https://libguides.murdoch.edu.au/systematic/PICO. Acessado em: 18/11/2020.

NORTHROP, L.; CLEMENTS, P. C.; BACHMANN, F.; BERGEY, J.; CHASTEK, G.; COHEN, S.; DONOHOE, P.; JONES, L.; KRUT, R.; LITTLE, R.; MCGREGOR, J. O'BRIEN, L. A Framework for Software Product Line Practice, version 5.0. SEI.-2007-http://www.sei.cmu.edu/productlines/index.html, 2007.

PAIVA, A. M.; FIGUEIREDO, E. ON THE DETECTION OF CODE CLONE WITH SEQUENCE OF METHOD CALLS. Dissertação (Masters Thesis) - University of Minas Gerais, Belo Horizonte, may 2016.

PETERSEN R. FELDT, S. M. K.; MATTSSON, M. Systematic Mapping Studies in Software Engineering. In: 12th International Conference on Evaluation and Assessment in Software Engineering. [S.l.: s.n.], 2008. p. 1.

PICADO, J. ESTRUTURAS DISCRETAS. Textos de Apoio. Segunda Edição. Departamento de Matemática. Universidade de Coimbra: 2014. 167. Disponível em: <http://www.mat.uc.pt/~picado/ediscretas/2018/apontamentos/TextosApoio.pdf> Acesso em: 20 fev 2019.

RATTAN, D.; KAUR, J. Systematic Mapping Study of Metrics Based Clone Detection Techniques. In: International Conference on Advances in Information Communication Technology & Computing. New York, NY, USA: ACM, 2016. (AICTC '16), p. 76:1-76:7.

ROT, Procedure; INICIA, Constructor; TERMINA, Destructor. Programação orientada a objetos. 1993.

ROY, C.K; DETECTION AND ANALYSIS OF NEAR-MISS SOFTWARE CLONES. Thesis (Doctor Thesis) - Queen's University, Kingston, Ontario, Canada, August 2009.

SCHUGERL, P. Scalable Clone Detection Using Description Logic. In: 5th International Workshop on Software Clones. New York, NY, USA: ACM, 2011. (IWSC '11), p. 47-53.

SCHULZE, S. Analysis and Removal of Code Clones in Software Product Lines. Dissertation, University of Magdeburg, School of Computer Science, 2012.

SCHULZE, S.; APEL, S.; KÄSTNER, C. Code Clones in Feature-Oriented Software Product Lines. ACM SIGPLAN Notices, v. 46, n. 2, p. 103-112, 2011.

SEGURA, S.; HIERONS, R. M.; BENAVIDES, D.; RUIZ-CORTES, A. Automated Test Data Generation on the Analyses of Feature Models: A Metamorphic Testing Approach. In: Third International Conference on Software Testing, Verification and Validation. Washington, DC, USA: IEEE Computer Society. (ICST'10), p. 35-44. 2010.

SILVA, F. A. P. da; NETO, P. A. d. M. S.; GARCIA, V. C.; MUNIZ, P. F. Linhas de Produtos de Software: Uma Tendência da Indústria. 2011.

SOLANKI, K.; KUMARI, S. Comparative Study of Software Clone Detection Techniques. In: Management and Innovation Technology International Conference (MITicon), 2016. [S.l.], 2016. p. MIT-152.

TORRES, J. J. B.; JUNIOR, M. C. R.; FARIAS, M. A. D. F. Procedural x OO: A Corporative Experiment on Source Code Clone Mining. In: International Conference on Enterprise Information Systems (ICEIS 2017) - V. 2 , pp395-402. 2017.

VALE, G.; ABÍLIO, R.; FREIRE, A.; COSTA, H. Criteria and Guidelines to Improve Software Maintainability in Software Product Lines. In: 12th International Conference on Information Technology - New Generations. [S.l.: s.n.], 2015. p. 427-432.

WIERINGA, R.; MAIDEN, N.; NANCY, M. ROLLAND, C. Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion. In: Journal Requirements Engineering. V. 11, Issue 1, pp102-107. 2005.

YUAN, Y.; GUO, Y. Cmcd: Count Matrix Based Code Clone Detection. In: 18th Asia-Pacific Software Engineering Conference. [S.l.: s.n.], 2011. p. 250-257. ISSN 1530-1362.

YUKI, Y.; HIGO, Y.; KUSUMOTO, S. A Technique to Detect Multi-Grained Code Clones. In: 11th International Workshop on Software Clones (IWSC). [S.l.: s.n.], 2017. p. 1-7.

APÊNDICE A

Tabela A.1 - Artigos Selecionados.

ID	Título	Autores	Fonte
A01	Detection of Near-Miss Clones Using Metrics and Abstract Syntax Trees	Vishwachi Gupta, Sonam	Ei Compendex
A02	Semantic Clone Detection Using Machine Learning	Sheneamer, A. Kalita, J.	IEEE
A03	Identifying Clones in the Linux Kernel	Casazza, G. Antoniol, G. Villano, U. Merlo, E. Penta, M. Di	IEEE
A04	LICCA: A Tool for Cross-Language Clone Detection	Vislavski, T. RakiÄ‡, G. Cardozo, N. Budimac, Z.	IEEE
A05	An Approach to Identify Duplicated Web Pages	Lucca, G. A. Di Penta, M. Di Fasolino, A. R.	IEEE
A06	SDD: High Performance Code Clone Detection System for Large Scale Source Code	Lee, S. Jeong, I.	ACM
A07	On Detection of Gapped Code Clones Using Gap Locations	Ueda, Y. Kamiya, T. Kusumoto, S. Inoue, K.	Ei Compendex
A08	Clone Detection Using Time Series and Dynamic Time Warping Techniques	Abdelkader, M. Mimoun, M.	IEEE
A09	XIAO: Tuning Code Clones at Hands of Engineers in Practice	Dang, Yingnong Zhang, Dongmei Ge, Song Chu, Chengyun Qiu, Yingjun Xie, Tao	ACM
A10	To Enhance the Code Clone Detection Algorithm by Using Hybrid Approach for Detection of Code Clones	Roopam Singh, G.	IEEE
A11	DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones	Jiang, L. Misherghi, G. Su, Z. Glondu, S.	IEEE
A12	A Code Clone Oracle	Krutz, D. E. Le, W.	Ei Compendex
A13	Detecting Differences Across Multiple Instances of Code Clones	Lin, Y. Xing, Z. Xue, Y. Liu, Y. Peng, X. Sun, J. Zhao, W.	ACM
A14	CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code	Kamiya, T. Kusumoto, S. Inoue, K.	IEEE
A15	Software Clone Detection Using Clustering Approach	Joshi, B. Budhathoki, P. Woon, W. L. Svetinovic, D.	Ei Compendex
A16	Hybridizing Evolutionary Algorithms and Clustering Algorithms to Find Source-Code Clones	Sutton, A. Kagdi, H. Maletic, J. I. Volkert, L. G.	ACM

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A17	Scalable and Incremental Clone Detection for Evolving Software	Nguyen, T. T. Nguyen, H. A. Al-Kofahi, J. M. Pham, N. H. Nguyen, T. N.	Ei Compendex
A18	CMCD: Count Matrix Based Code Clone Detection	Yuan, Y. Guo, Y.	IEEE
A19	Is Code Cloning in Games Really Different?	Al-Omari, F. Roy, C. K.	ACM
A20	Scalable Clone Detection Using Description Logic	Schugerl, P.	ACM
A21	DebCheck: Efficient Checking for Open Source Code Clones in Software Systems	Cordy, J. R. Roy, C. K.	IEEE
A22	SPAPE: A Semantic-Preserving Amorphous Procedure Extraction Method for Near-Miss Clones	Bian, Y. Koru, G. Su, X. Ma, P.	Science Direct
A23	Clone Detection Via Structural Abstraction	Evans, W. S. Fraser, C. W. Ma, F.	Ei Compendex
A24	Ctcompare: Code Clone Detection Using Hashed Token Sequences	Toomey, W.	Science Direct
A25	CloneWorks: A Fast and Flexible Large-scale Near-miss Clone Detection Tool	Svajlenko, J. Roy, C. K.	ACM
A26	Examining the Effectiveness of Using Concolic Analysis to Detect Code Clones	Krutz, D. E. Malachowsky, S. A. Shihab, E.	ACM
A27	Code Clone Detection Experience at Microsoft	Dang, Y. Ge, S. Huang, R. Zhang, D.	ACM
A28	Instant Code Clone Search	Lee, M.-W. Roh, J.-W. Hwang, S.-Won Kim, S.	ACM
A29	Fast and Flexible Large-scale Clone Detection with CloneWorks	Svajlenko, J. Roy, C. K.	ACM
A30	Detection of Code Clones in Software Generators	Lillack, M. Bucholdt, C. Schilling, D.	ACM
A31	Phoenix-based Clone Detection Using Suffix Trees	Tairas, R. Gray, J.	ACM
A32	A Scalable and Accurate Approach Based on Count Matrix for Detecting Code Clones	Yuan, Y.	ACM
A33	ClemanX: Incremental Clone Detection Tool for Evolving Software	Nguyen, T. T. Nguyen, H. A. Pham, N. H. Al-Kofahi, J. M. Nguyen, T. N.	IEEE
A34	Tree-Pattern-Based Duplicate Code Detection	Lee, H.-S. Doh, K.-G.	ACM
A35	Clone-Aware Configuration Management	Nguyen, T. T. Nguyen, H. A. Pham, N. H. Al-Kofahi, J. M. Nguyen, T. N.	IEEE
A36	Code Relatives: Detecting Similarly Behaving Software	Su, F.-H. Bell, J. Harvey, K. Sethumadhavan, S. Kaiser, G. Jebara, T.	ACM

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A37	Detecting Code Clones with Gaps by Function Applications	Matsushita, T. Sasano, I.	ACM
A38	CCSharp: An Efficient Three-Phase Code Clone Detector Using Modified PDGs	Wang, M. Wang, P. Xu, Y.	IEEE
A39	Deep Learning Code Fragments for Code Clone Detection	White, M. Tufano, M. Vendome, C. Poshyvanyk, D.	ACM
A40	Constructing Universal Version History	Chang, H.-F. Mockus, A.	ACM
A41	Efficient Token Based Clone Detection with Flexible Tokenization	Basit, H. A. Jarzabek, S.	ACM
A42	CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE	Jablonski, P. Hou, D.	ACM
A43	Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets	Chen, K. Liu, P. Zhang, Y.	ACM
A44	A Hybrid Approach (Syntactic and Textual) to Clone Detection	Funaro, M. Braga, D. Campi, A. Ghezzi, C.	ACM
A45	IDE-based Real-time Focused Search for Near-miss Clones	Zibran, M. F. Roy, C. K.	ACM
A46	Clone Detection and Removal for Erlang/OTP Within a Refactoring Environment	Li, H. Thompson, S.	ACM
A47	Practical Language-independent Detection of Near-miss Clones	Cordy, J. R. Dean, T. R. Synytskyy, N.	ACM
A48	Clone Detection and Elimination for Haskell	Brown, C. Thompson, S.	ACM
A49	VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery	Kim, S. Woo, S. Lee, H. Oh, H.	IEEE
A50	SCVD: A New Semantics-Based Approach for Cloned Vulnerable Code Detection	Zou, D. Qi, H. Li, Z. Wu, S. Jin, H. Sun, G. Wang, S. Zhong, Y.	Ei Compendex
A51	Implementation of Analytical Hierarchy Process in Detecting Structural Code Clones	Aktas, M. S. Kapdan, M.	Ei Compendex
A52	Code Clone Genealogy Detection on e-Health System Using Hadoop	Tekchandani, R. Bhatia, R. Singh, M.	Ei Compendex
A53	Scalable Code Clone Search for Malware Analysis	Farhadi, M. R. Fung, B. C. M. Fung, Y. B. Charland, P. Preda, S. Debbabi, M.	Ei Compendex
A54	Scalable and Accurate Detection of Code Clones	Sargsyan, S. Kurmangaleev, Sh. Belevantsev, A. Avetisyan, A.	Ei Compendex

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A55	Clone-Based and Interactive Recommendation for Modifying Pasted Code	Lin, Y. Peng, X. Xing, Z. Zheng, D. Zhao, W.	Ei Compendex
A56	Structural Code Clone Detection Methodology Using Software Metrics	Aktas, M. S. Kapdan, M.	Ei Compendex
A57	CLORIFI: Software Vulnerability Discovery Using Code Clone Verification	Li, H. Kwon, H. Kwon, J. Lee, H.	Ei Compendex
A58	Identifying Code Clones with Refactorerl	Fordos, V. Toth, M.	Ei Compendex
A59	A Novel Approach to Effective Detection and Analysis of Code Clones	Rajakumari, K. E. Jebarajan, T.	IEEE
A60	Modular Heap Abstraction-Based Code Clone Detection for Heap-Manipulating Programs	Dong, L. Wang, J. Chen, L.	IEEE
A61	Incremental Clone Detection and Elimination for Erlang Programs	Li, H.; Thompson, S.	Ei Compendex
A62	A Novel Detection Approach for Statement Clones	Shi, Q. Q. Zhang, L. P. Meng, F. J. Liu, D. S.	IEEE
A63	Folding Repeated Instructions for Improving Token-Based Code Clone Detection	Murakami, H. Hotta, K. Higo, Y. Igaki, H. Kusumoto, S.	Ei Compendex
A64	Incremental Code Clone Detection: A PDG-Based Approach	Higo, Y. Yasushi, U. Nishino, M. Kusumoto, S.	IEEE
A65	Clone Detection through Process Algebras and Java Bytecode	Santone, A.	Ei Compendex
A66	SHINOBI: A Tool for Automatic Code Clone Detection in the IDE	Kawaguchi, S. Yamashina, T. Uwano, H. Fushida, K. Kamei, Y. Nagura, M. Iida, H.	IEEE
A67	Cross-Language Clone Detection	Kraft, N. A. Bonds, B. W. Smith, R. K.	Ei Compendex
A68	Threshold-Free Code Clone Detection for a Large-Scale Heterogeneous Java Repository	Keivanloo, I. Zhang, F. Zou, Y.	IEEE
A69	Code Clone Detection Using Parsing Actions	Lazar, F. M. Banias, O.	IEEE
A70	Boreas: An Accurate and Scalable Token-Based Approach to Code Clone Detection	Yuan, Y. Guo, Y.	IEEE
A71	Code Clone Detection Using Wavelets	Karus, S. Kilgi, K.	IEEE
A72	CCCD: Concolic Code Clone Detection	Krutz, D. E. Shihab, E.	IEEE
A73	An Execution-Semantic and Content-and-Context-Based Code-Clone Detection and Analysis	Kamiya, T.	IEEE
A74	An Efficient Code Clone Detection Model on Java Byte Code Using Hybrid Approach	Raheja, K. Tekchandani, R. K.	IEEE

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A75	Code Clone Detection Using Decentralized Architecture and Code Reduction	Patil, R. V. Joshi, S. D. Shinde, S. V. Ajagekar, D. A. Bankar, S. D.	IEEE
A76	LLVM-Based Code Clone Detection Framework	Avetisyan, A. Kurmangaleev, S. Sargsyan, S. Arutunian, M. Belevantsev, A.	IEEE
A77	Selecting a Set of Appropriate Metrics for Detecting Code Clones	Bansal, G. Tekchandani, R.	IEEE
A78	ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions	Jang, J. Agrawal, A. Brumley, D.	IEEE
A79	CCLearner: A Deep Learning-Based Clone Detection Approach	Li, L. Feng, H. Zhuang, W. Meng, N. Ryder, B.	IEEE
A80	Scalable Detection of Semantic Clones	Gabel, M. Jiang, L. Su, Z.	IEEE
A81	Code Clone Detection on Specialized PDGs with Heuristics	Higo, Y. Kusumoto, S.	IEEE
A82	A Technique to Detect Multi-Grained Code Clones	Yuki, Y. Higo, Y. Kusumoto, S.	IEEE
A83	Code Syntax-Comparison Algorithm Based on Type-Redefinition-Preprocessing and Rehash Classification	Cui, B. Guan, J. Guo, T. Han, L. Wang, J. Ju. Y.	Scopus
A84	WSIM: Detecting Clone Pages Based on 3-Levels of Similarity Clues	Jung, W. Wu, C. Lee, E.	IEEE
A85	Gapped Code Clone Detection with Lightweight Source Code Analysis	Murakami, H. Hotta, K. Higo, Y. Igaki, H. Kusumoto, S.	IEEE
A86	Clone Detection Meets Semantic Web-Based Transitive Closure Computation	Keivanloo, I. Rilling, J.	IEEE
A87	Towards Slice-Based Semantic Clone Detection	Alomari, H. W. Stephan, M.	IEEE
A88	VFDETECT: A Vulnerable Code Clone Detection System Based on Vulnerability Fingerprint	Liu, Z. Wei, Q. Cao, Y.	IEEE
A89	BinClone: Detecting Code Clones in Malware	Farhadi, M. R. Fung, B. C. M. Charland, P. Debbabi, M.	IEEE
A90	SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch Mode and During Software Development	Saini, V. Sajnani, H. Kim, J. Lopes, C.	IEEE
A91	Code Clone Detection Using Parsing Actions	Maeda, K.	IEEE

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A92	Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics	Kodhai, E. Kanmani, S. Kamatchi, A. Radhika, R. Saranya, B. V.	IEEE
A93	Semantic Code Clone Detection Using Parse Trees and Grammar Recovery	Tekchandani, R. Bhatia, R. K. Singh, M.	IEEE
A94	A Data Mining Approach for Detecting Higher-Level Clones in Software	Basit, H. A. Jarzabek, S.	IEEE
A95	Implementing a 3-Way Approach of Clone Detection and Removal Using PC Detector Tool	Mahajan, G. Bharti, M.	IEEE
A96	A Novel Approach Based on Formal Methods for Clone Detection	Cuomo, A. Santone, A. Villano, U.	IEEE
A97	CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization	Semura, Y. Yoshida, N. Choi, E. Inoue, K.	IEEE
A98	Scalable Code Clone Detection and Search Based on Adaptive Prefix Filtering	Nishi, M. A. Damevski, K.	IEEE
A99	Tree-Pattern-Based Clone Detection with High Precision and Recall	Lee, H.-S. Choi, M.-R. Doh, K.-G.	Scopus
A100	Interface Driven Code Clone Detection	Patil, R. V. Joshi, S. D. Shinde, S. V. Khanna, V.	Scopus
A101	Method-Level Code Clone Detection for Java Through Hybrid Approach	Kodhai, E. Kanmani, S.	Scopus
A102	Non-Trivial Software Clone Detection Using Program Dependency Graph	Gautam, P. Saini, H.	Scopus
A103	An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies	Saha, R. K. Roy, C. K. Schneider, K. A.	Scopus
A104	Code Clones Detection Using Machine Learning Technique: Support Vector Machine	Jadon, S.	Scopus
A105	Generic Code Cloning Method for Detection of Clone Code in Software Development	Haque, S. M. F. Srikanth, V. Reddy, E. S.	Scopus
A106	Method-Level Code Clone Detection Through LWH (Light Weight Hybrid) Approach	Kodhai, E. Kanmani, S.	Springer Link
A107	Code Clone Detection Using Coarse and Fine-Grained Hybrid Approaches	Sheneamer, A. Kalita, J.	Scopus
A108	Method-Level Incremental Code Clone Detection Using Hybrid Approach	Kodhai, E. Kanmani, S.	Scopus
A109	A Measurement of Similarity to Identify Identical Code Clones	Mythili, S. S. Sarala, S.	Scopus
A110	An Effective Approach Using Dissimilarity Measures to Estimate Software Code Clone	Patil, R. V. Joshi, S. D. Shinde, S. V. Khanna, V.	Scopus
A111	Enhancing Generic Pipeline Model for Code Clone Detection Using Divide and Conquer Approach	Mubarak-Ali, A.-F. Syed-Mohamad, S. Sulaiman, S.	Scopus
A112	Semantic Code Clone Detection for Internet of Things Applications Using Reaching Definition and Liveness Analysis	Tekchandani, R. Bhatia, R. Singh, M.	Springer Link
A113	Structural Similarity Detection Using Structure of Control Statements	Sudhamani, M. Rangarajan, L.	Scopus

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A114	A Metric Space Based Software Clone Detection Approach	Li, Z. Sun, J.	Scopus
A115	An Iterative, Metric Space Based Software Clone Detection Approach	Li, Z. Sun, J.	Scopus
A116	A Hybrid Technique in Pre-Processing and Transformation Process for Code Clone Detection	Mubarak Ali, A.-F. Sulaiman, S.	Scopus
A117	Code Process Block Based Reduction Technique for Software Code Clone Detection	Patil, R. V. Joshi, S. D. Shinde, S. V. Khanna, V.	Scopus
A118	Index-Based Code Clone Detection: Incremental, Distributed, Scalable	Hummel, B. Juergens, E. Heinemann, L. Conradt, M.	Scopus
A119	An Improved Method for Tree-Based Clone Detection in Web Applications	Li, C. Sun, J. Chen, H.	Scopus
A120	Detection of Recurring Clones Using Weighted Frequent Itemset Mining	Mythili, S. Sarala, S.	Scopus
A121	KLONOS: Similarity-Based Planning Tool Support for Porting Scientific Applications	Ding, W. Hsu, C.-H. Hernandez, O. Chapman, B. Graham, R.	Scopus
A122	An Effective Software Clone Detection Using Distance Clustering	Devi, D. G. Punithavalli, M.	Scopus
A123	Models are Code Too: Near-Miss Clone Detection for Simulink Models	Alalfi, M. H. Cordy, J. R. Dean, T. R. Stephan, M. Stevenson, A.	Scopus
A124	An Efficient New Multi-Language Clone Detection Approach from Large Source Code	Rehman, S. U. Khan, K. Fong, S. Biuk-Aghai, R.	Scopus
A125	Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection	Nguyen, H. A. Nguyen, T. T. Pham, N. H. Al-Kofahi, J. M. Nguyen, T. N.	Springer Link
A126	Tree Slicing: Finding Intertwined and Gapped Clones in One Simple Step	Akhin, M. Itsykson, V.	Springer Link
A127	Detect Functionally Equivalent Code Fragments Via k-Nearest Neighbor Algorithm	Kong, D. Su, X. Wu, S. Wang, T. Ma, P.	Scopus
A128	Parallel Code Clone Detection Using MapReduce	Sajnani, H. Ossher, J. Lopes, C.	Scopus
A129	CLCDSA: Cross Language Code Clone Detection Using Syntactical Features and API Documentation	Li, Guanhua; Wu, Yijian; Roy, Chanchal K.; Sun, Jun; Peng, Xin; Zhan, Nanjie; Hu, Bin; Ma, Jingyi	ACM
A130	Neural Detection of Semantic Code Clones via Tree-Based Convolution	Xue, Hongfa; Mei, Yongsheng; Gogineni, Kailash; Venkataramani, Guru; Lan, Tian	ACM

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A131	CCAligner: A Token Based Large-Gap Clone Detector	Fang, Chunrong; Liu, Zixi; Shi, Yangyang; Huang, Jeff; Shi, Qingkai	ACM
A132	Intelligent Token-Based Code Clone Detection System for Large Scale Source Code	Ragkhitwetsagul, C.; Krinke, J.; Marnette, B.	ACM
A133	Semantic Code Clone Detection for Enterprise Applications	Yang, Y.; Ren, Z.; Chen, X.; Jiang, H.	ACM
A134	Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis	Perez, D.; Chiba, S.	Ei Compendex
A135	Semantic code clone detection using abstract memory states and program dependency graphs	Wang, Y.; Liu, D.	Ei Compendex
A136	A topic modeling approach for code clone detection	Yuan, Y.; Kong, W.; Hou, G.; Hu, Y.; Watanabe, M.; Fukuda, A.	Ei Compendex
A137	Cloned buggy code detection in practice using normalized compression distance	Hung, Y.; Takada, S.	Ei Compendex
A138	CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones	Dong, W.; Feng, Z.; Wei, H.; Luo, H.	Ei Compendex
A139	Siamese: scalable and incremental code clone search via multiple code representations	Capiluppi, Andrea; Di Ruscio, Davide; Di Rocco, Juri; Nguyen, Phuong T.; Ajienka, Nemitari	Ei Compendex
A140	Comparison of Token-Based Code Clone Method with Pattern Mining Technique and Traditional String Matching Algorithms In-terms of Software Reuse	Liu, J.; Wang, T.; Feng, C.; Wang, H.; Li, D.	Ei Compendex
A141	IBFET: Index-based features extraction technique for scalable code clone detection at file level granularity	Nafi, Kawser Wazed; Roy, Banani; Roy, Chanchal K.; Schneider, Kevin A.	Ei Compendex
A142	TECCD: A Tree Embedding Approach for Code Clone Detection	Wu, M.; Wang, P.; Yin, K.; Cheng, H.; Xu, Y.; Roy, C. K.	Ei Compendex
A143	DroidSD: An efficient indexed based android applications similarity detection tool	Zeng, J.; Ben, K.; Li, X.; Zhang, X.	Ei Compendex
A144	SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration	Li, Guanhua; Wu, Yijian; Roy, Chanchal K.; Sun, Jun; Peng, Xin; Zhan, Nanjie; Hu, Bin; Ma, Jingyi	Ei Compendex
A145	Twin-Finder: Integrated Reasoning Engine for Pointer-Related Code Clone Detection	Xue, Hongfa; Mei, Yongsheng; Gogineni, Kailash; Venkataramani, Guru; Lan, Tian	Ei Compendex
A146	Functional code clone detection with syntax and semantics fusion learning	Fang, Chunrong; Liu, Zixi; Shi, Yangyang; Huang, Jeff; Shi, Qingkai	Ei Compendex
A147	A picture is worth a thousand words: Code clone detection based on image similarity	Ragkhitwetsagul, C.; Krinke, J.; Marnette, B.	IEEE
A148	Structural Function Based Code Clone Detection Using a New Hybrid Technique	Yang, Y.; Ren, Z.; Chen, X.; Jiang, H.	IEEE

(continua)

Tabela A-1 - Artigos Selecionados. (cont.)

ID	Título	Autores	Fonte
A149	Cross-Language Clone Detection by Learning Over Abstract Syntax Trees	Perez, D.; Chiba, S.	IEEE
A150	Image-Based Clone Code Detection and Visualization	Wang, Y.; Liu, D.	IEEE
A151	From Local to Global Semantic Clone Detection	Yuan, Y.; Kong, W.; Hou, G.; Hu, Y.; Watanabe, M.; Fukuda, A.	IEEE
A152	CPPCD: A Token-Based Approach to Detecting Potential Clones	Hung, Y.; Takada, S.	IEEE
A153	A Novel Code Stylometry-based Code Clone Detection Strategy	Dong, W.; Feng, Z.; Wei, H.; Luo, H.	IEEE
A154	Detecting Java software similarities by using different clustering techniques	Capiluppi, Andrea; Di Ruscio, Davide; Di Rocco, Juri; Nguyen, Phuong T.; Ajienka, Nemitari	Science Direct
A155	A Large-Gap Clone Detection Approach Using Sequence Alignment via Dynamic Parameter Optimization	Liu, J.; Wang, T.; Feng, C.; Wang, H.; Li, D.	Scopus
A156	A universal cross language software similarity detector for open source software categorization	Nafi, Kawser Wazed; Roy, Banani; Roy, Chanchal K.; Schneider, Kevin A.	Science Direct
A157	LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach	Wu, M.; Wang, P.; Yin, K.; Cheng, H.; Xu, Y.; Roy, C. K.	Scopus
A158	Fast code clone detection based on weighted recursive autoencoders	Zeng, J.; Ben, K.; Li, X.; Zhang, X.	Scopus

Fonte: Do autor (2020)