

**DETECÇÃO DE CÓDIGO CLONADO USANDO  
SEQUÊNCIA DE CHAMADAS DE MÉTODOS**



ALEXANDRE MARTINS PAIVA

**DETECÇÃO DE CÓDIGO CLONADO USANDO  
SEQUÊNCIA DE CHAMADAS DE MÉTODOS**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: EDUARDO FIGUEIREDO

Belo Horizonte

Maio de 2016



ALEXANDRE MARTINS PAIVA

**ON THE DETECTION OF CODE CLONE WITH  
SEQUENCE OF METHOD CALLS**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: EDUARDO FIGUEIREDO

Belo Horizonte

May 2016

© 2016, Alexandre Martins Paiva.  
Todos os direitos reservados.

Paiva, Alexandre Martins

D1234p On the Detection of Code Clone with Sequence of  
Method Calls / Alexandre Martins Paiva. — Belo  
Horizonte, 2016  
xxiv, 76 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of  
Minas Gerais

Orientador: Eduardo Figueiredo

1. Code clones. 2. detection method. 3. method  
calls. I. Título.

CDU 519.6\*82.10



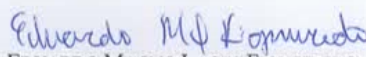
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

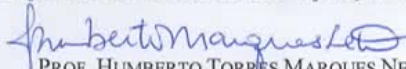
## FOLHA DE APROVAÇÃO

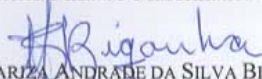
On the detection of code clones with sequence of method calls

**ALEXANDRE MARTINS PAIVA**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. HUMBERTO TORRES MARQUES NETO  
Instituto de Ciências Exatas e Informática - PUC/MG

  
PROFA. MARIZA ANDRADE DA SILVA BIGONHA  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 11 de maio de 2016.





*À minha esposa, Christiane, principal incentivadora deste mestrado, dedico este trabalho. Serei sempre grato pelo seu apoio irrestrito!*

*Aos meus filhos, André e Ana Clara, dedico este trabalho. Peço desculpas pela ausência nos momentos em que tive que dedicar ao mestrado. Como Baby costumava dizer, eu estava escrevendo a história detalhada da minha vida, tão infinito parecia ser este texto aqui escrito. Obrigado pela paciência!*

*À minha mãe, Celina, também dedico este trabalho. Minha mãe jamais chamou minha atenção por qualquer motivo (é uma estratégia de educação meio arriscada, a qual não recomendo a ninguém). Como a única exceção, ela jamais admitiu que eu faltasse à aula. Guardei este exemplo vitorioso: a educação em primeiro lugar, como o único bem material realmente importante.*

*À Tia Glória, minha madrinha, dedico este trabalho. Tia Glória é a pessoa mais agradável que conheço, detentora da maior rede social, do maior número de seguidores e de curtidas, vivendo a vida exclusivamente no mundo real.*

*Por fim, dedico este trabalho ao meu pai, Sérgio, às minhas irmãs, Stella, Tina e Ângela, e demais pessoas que gostam de mim.*

*Que um dia eu possa retribuí-los...*



# Acknowledgments

Quando comecei o mestrado, não imaginava que seria uma trajetória tão boa e tão árdua. Nesta universidade maravilhosa, que me traz agradáveis recordações da minha graduação, reencontrei o privilégio de estar em sala de aula. Concluo aqui uma etapa importante e muito marcante da minha vida. A todos que conviveram comigo neste período, meus sinceros agradecimentos!

A Deus, agradeço por cada dia que me foi concedido.

De maneira especial, gostaria de agradecer ao meu orientador, Dr. Eduardo Figueiredo. Sua dedicação, sua paciência, sua organização, seu incentivo e, sobretudo, sua sabedoria foram fundamentais para que eu pudesse desenvolver e concluir este trabalho.

Agradeço à FITec - Fundação para Inovações Tecnológicas -, especialmente aos meus amigos Leonardo Resende e Eugênio Daher, que me permitiram conciliar as atividades do trabalho com as atividades acadêmicas. Também da FITec, agradeço ao Prof. Antônio Hamilton, que prontamente escreveu minha recomendação ao curso de mestrado do DCC.

Em parágrafo dedicado, agradeço imensamente ao melhor gerente que já tive na vida, Ricardo Alves, pela gentileza e prestatividade, não somente durante o mestrado, mas em todos os momentos que trabalhamos juntos. Sem a sua eterna boa vontade, este trabalho não teria se realizado.

Agradeço ao meu gerente na Ericsson, Daniel Rosa, pelas muitas oportunidades que confiou a mim. Agradeço também por ter permitido que minhas atividades profissionais fossem conciliadas com meus interesses pessoais, em especial, este curso.

Agradeço ao Dr. Marco Túlio Valente por ter me aceitado como seu aluno em disciplinas isoladas e por ter posteriormente me recomendado como aluno de mestrado do DCC.

Agradeço aos professores Dr. Humberto Torres Marques Neto e Dr<sup>a</sup> Mariza Bigonha por ter aceitado o convite para compor a banca de defesa.

Agradeço aos colegas do curso pelos momentos de aprendizado e descontração,

aos colegas do Labsoft pelas valiosas contribuições ao trabalho, deixando um abraço ao amigo e co-autor, Johnatan.

Agradeço ao meu país por ter me dado esta oportunidade e levo comigo a obrigação de devolver o investimento em benefícios para a sociedade.

Peço desculpas àqueles que por ventura eu tenha esquecido de agradecer, mas que contribuíram de alguma forma com este trabalho.

Muito obrigado!

*“Ainda que eu falasse as línguas dos homens e dos anjos, e não tivesse amor, seria como o metal que soa ou como o sino que tine.*

*E ainda que tivesse o dom de profecia, e conhecesse todos os mistérios e toda a ciência, e ainda que tivesse toda a fé, de maneira tal que transportasse os montes, e não tivesse amor, nada seria.”*

*(Coríntios 13:1-2)*



# Resumo

Desenvolvedores de software geralmente copiam e colam código de uma parte do sistema para outro. Essa prática, chamada de clonagem de código, dispersa uma mesma lógica em diferentes pontos do sistema, dificultando as tarefas de manutenção e evolução. Vários métodos têm sido propostos objetivando localizar códigos clonados para posterior eliminação. No entanto, alguns tipos de códigos clonados são difíceis de encontrar, especialmente quando as partes recebem diferentes alterações. Este trabalho propõe um método para detectar códigos clonados. Esse método se baseia em busca de códigos clonados em sistemas de código aberto analisando sequências similares de chamadas de métodos. O método proposto foi implementado em uma nova ferramenta, chamada McSheep. Os resultados obtidos pelo método foram comparados com códigos clonados detectados por uma ferramenta largamente utilizada, chamada PMD. Por fim, um estudo foi realizado com 25 desenvolvedores. Por meio de inspeção de código, os participantes desse estudo analisaram um conjunto específico de códigos clonados detectados pela ferramenta McSheep. Esse estudo mostrou que mais de 90 % dos participantes concordaram com o método com relação aos trechos de código. Portanto, os resultados indicam que a análise de sequência de chamadas de métodos é uma estratégia válida para localização de códigos clonados.





# Abstract

Software developers usually copy and paste code from one part of the system to another. This practice, called code clone, spreads same logic over the system, hindering maintenance and evolution tasks. Several methods were proposed in order to detect code clones for further elimination. However, some types of code clones are hard to detect, specially when clones receive different changes. This work proposes a method for detecting code clones. This method relies on searching code clones in software systems by analyzing similar sequences of method calls. The proposed method was implemented in a new open source tool, called McSheep. Our results were compared with code clones detected by a state of the practice tool, called PMD. The comparison results indicate that the proposed method is able to detect code clones, since McSheep and PMD detected a common set of code clones. Additionally, the comparison showed that both tools are complementary, since each one detected an exclusive set of code clones. In addition, a user study was conducted with 25 developers. By means of code inspection, participants analyzed the code clones detected by our method. This user study showed that more than 90% of subjects agree with the code clones found by the method. Therefore, results indicate that analyzing sequences of method calls is a valid strategy for code clone detection.



# List of Figures

2.1	Original Source Code for Code Clone Types Example. . . . .	6
2.2	Code Clones Type 1 and Type 2. . . . .	6
2.3	Code Clones Type 3. . . . .	7
2.4	Code Clone Type 4. . . . .	8
3.1	Sequence of method calls inside method <i>countAvailablePapers</i> . . . . .	14
3.2	Sequence of method calls in the class scope. . . . .	16
3.3	Variability of method calls in a sequence. . . . .	17
3.4	Snippet adapted from <i>DialogPresenter</i> class of <i>FacebookAndroid</i> . . . . .	17
3.5	Order of method calls in a <i>for</i> statement. . . . .	17
3.6	Methods <i>ToolchainsBuildingException.toMessage</i> and <i>ModelBuildingException.toMessage</i> extracted from Maven. . . . .	19
4.1	McSheep Steps for Code Clone Detection. . . . .	24
4.2	Code clone detected by coincident sequence of method calls. . . . .	26
4.3	Methods <i>GenericCatalogDAO.getCategory</i> and <i>GenericCatalogDAO.getProducts</i> extracted from PetStore. . . . .	31
5.1	Expected set of code clones candidates presented by tool. . . . .	34
5.2	Relation between different configurations. . . . .	34
5.3	Code clone detected in the ArgoUML by both McSheep and PMD. . . . .	38
5.4	Code clone detected in the ArgoUML only by the PMD. . . . .	39
5.5	Code clone detected in the ArgoUML only by the McSheep. . . . .	40
6.1	Survey Case 1. . . . .	45
6.2	Survey First Step. . . . .	46
6.3	Survey Second Step. . . . .	46
6.4	Subject Profiles. . . . .	47
6.5	Number of agreements per Survey Case. . . . .	48

6.6	Code snippet with the lowest level of acceptance. . . . .	49
6.7	Code 1: One of two cases with 100% of agreement. . . . .	50
6.8	Code 2: One of two cases with 100% of agreement. . . . .	51
6.9	Agreement per Object Oriented Programming Expertise. . . . .	52
6.10	Percentage of agreement per Java Expertise. . . . .	52
6.11	Percentage of agreement per Bad Smell Expertise. . . . .	53
6.12	Percentage of agreement per Code Duplication Expertise. . . . .	53
6.13	Percentage of agreement per Development Expertise. . . . .	54
A.1	Characterization Form . . . . .	63
B.1	Question 1. . . . .	65
B.2	Question 2. . . . .	66
B.3	Question 3. . . . .	67
B.4	Question 4. . . . .	68
B.5	Question 5. . . . .	69
B.6	Question 6. . . . .	70
B.7	Question 7. . . . .	71
B.8	Question 8. . . . .	72
B.9	Question 9. . . . .	73
B.10	Question 10. . . . .	74
B.11	Question 11. . . . .	75
B.12	Question 12. . . . .	76

# List of Tables

3.1	Sequences of method calls of <i>ToolchainsBuildingException.toMessage</i> and <i>ModelBuildingException.toMessage</i> . . . . .	20
4.1	Hash list with method call as key and where call occurs as values. . . . .	26
4.2	Hash list with method as key and its method calls as values. . . . .	26
4.3	Systems selected for analysis. . . . .	28
4.4	Method calls scattering. . . . .	28
4.5	Code clones candidates per configuration. . . . .	29
5.1	Code Clones Found: McSheep and PMD. . . . .	37



# Contents

<b>Acknowledgments</b>	<b>xi</b>
<b>Resumo</b>	<b>xv</b>
<b>Abstract</b>	<b>xvii</b>
<b>List of Figures</b>	<b>xix</b>
<b>List of Tables</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation, Problem Description, and Goal . . . . .	1
1.2 The Proposed Solution . . . . .	2
1.3 The Method Evaluation . . . . .	3
1.4 Dissertation Outline . . . . .	3
<b>2 Background and Related Work</b>	<b>5</b>
2.1 Code Clones . . . . .	5
2.2 Code Clone Detection Techniques . . . . .	8
2.3 Related Work . . . . .	10
2.4 Final Remarks . . . . .	11
<b>3 The Proposed Method</b>	<b>13</b>
3.1 The Method Strategy . . . . .	13
3.2 The Sequence of Method Calls . . . . .	14
3.3 The Sequences Grouped by Class or Method . . . . .	15
3.4 The Order of Method Calls in Java . . . . .	16
3.5 The Size of Coincident Sequences . . . . .	18
3.6 The Code Clone Candidates . . . . .	21
3.7 Final Remarks . . . . .	21

<b>4</b>	<b>Automated Method in Action</b>	<b>23</b>
4.1	Tool Support . . . . .	23
4.2	Selected Systems . . . . .	27
4.3	Code Clone Candidates . . . . .	27
4.4	Final Remarks . . . . .	32
<b>5</b>	<b>Comparative Evaluation</b>	<b>33</b>
5.1	Study Settings . . . . .	33
5.2	The PMD Tool . . . . .	35
5.3	Results and Discussion . . . . .	36
5.4	Threats to Validity . . . . .	39
5.5	Final Remarks . . . . .	41
<b>6</b>	<b>User Study</b>	<b>43</b>
6.1	Study Settings . . . . .	44
6.2	Results and Discussion . . . . .	47
6.3	Threats to Validity . . . . .	55
6.3.1	Construct Validity . . . . .	55
6.3.2	Internal Validity . . . . .	55
6.3.3	External Validity . . . . .	55
6.3.4	Conclusion Validity . . . . .	56
6.4	Final Remarks . . . . .	56
<b>7</b>	<b>Conclusions and Future Work</b>	<b>57</b>
	<b>Bibliography</b>	<b>59</b>
	<b>Appendix A Characterization Form</b>	<b>63</b>
	<b>B Survey Questions</b>	<b>65</b>



# Chapter 1

## Introduction

A common decision made by a software developer when coding is to reuse existing code as reference [Rattan et al., 2013]. Reasons vary and are not mutual exclusive. They include getting an idea of how to solve a specific problem or using something already tested [Ducasse et al., 1999]. Copying existing code fragments and pasting them with or without modifications into other parts of the system is called code clone, an important area of software engineering research [Rattan et al., 2013; Ducasse et al., 1999; Roy et al., 2009]. However, this practice is a bad smell [Fowler, 1999] and leads to problems during software development and maintenance tasks. The reason is that code clones duplicate logic and, therefore, increases points of refactoring and error fixing.

In order to find and eliminate code clones, several methods and tools have been proposed. The most common strategies are based on tree [Wahler et al., 2004], program dependency graph (PDG) [Krinke, 2001; Komondoor and Horwitz, 2001], text [Johnson, 1993], token [Hummel et al., 2010], metrics [Marinescu, 2004; Kontogiannis, 1997], and hybrid techniques [Göde and Koschke, 2011]. This variety of strategies allows detection of different types of code clones, offering methods that best fit different projects. Although the number of different methods for code clone detection is considerable, the research of new methods remains an open research field.

### 1.1 Motivation, Problem Description, and Goal

One of the reasons for code clone detection being an active software engineering research field is that there are virtually endless possibilities of changes in a copied and pasted code, generating code clones not detected by existing strategies. Another reason is that programming languages have been created and evolved. These programming languages can introduce new syntax or change existing ones. These and other reasons lead to code

clones not covered by existing methods [Khan et al., 2014]. Therefore, the motivation of this work is to contribute in this open research field: code clone detection.

In order to contribute, this work relies on the following observation of the code clone problem. When a code is copied from one part of the system and pasted into another, all the instructions (declarations, statements, operations, method calls, etc.) of the original code come together. Therefore, a code clone is a repetition of sequence of instructions in different points of one (or more) systems. After the code clone operation, the parts can evolve independently, receiving different changes. These changes make harder to detect the code clone. However, part of the original sequence of method calls is supposed to be preserved, since the original computation is one of the reasons for code cloning.

*The main goal of this dissertation is to propose and evaluate a new method for code clone detection based on similar sequences of method calls.* The method strategy is to detect code clones comparing similar sequences of method calls in different parts of the target system. Our method analyzes the correlation between code clones and sequences of method calls.

## 1.2 The Proposed Solution

The first step of our method is to collect all the method calls of a target system. For a given system, the method calls occur in a scope of a class or a method. Therefore, the collected method calls are grouped by class or by method, according to the scope where they can be found. Once method calls are collected, our strategy is to find out which similar sequences occur in different parts of the system. These sequences, in order to be considered as a code clone candidate, must contain a minimum number of coincident method calls. In addition, part of this sequence must be continuous.

The proposed method relies on detecting code clones in Java systems. The detection method is supported by a tool, called McSheep, which aims to extract code clones comparing sequences of method calls. First, the software artifacts are downloaded from software repositories and the Java files are extracted. These Java files are then parsed in order to extract all method calls. Each individual method call with more than one point of occurrence is a starting point to look for code clones. Finally, McSheep outputs the code clone candidates, that is, the code snippets with the same sequence of method calls (with a minimum size) found in different locations.

## 1.3 The Method Evaluation

The results gathered by the detection method are compared to a widely used code clone detection tool, called PMD. Both tools ran against the same 14 systems (ArgoUML, Ecommerce 2, Ecommerce 21, Facebook Android, Health Watcher, JBoss, Junit, Learn Engh, Maven, Mobile Media, PetStore, Restaurant Open, Restaurantr, and Telestrada) and code clones are collected. The code clones found by our method are compared to code clones found by the PMD. The agreement between these different methods reveals that both are complementary. Since they have different strategies, different results were found.

There are three groups of code clones as follows. The first group are the code clones detected by the PMD and not detected by our method. This group indicates that some types of code clones are not possible to be detected using our strategy. The second group are the code clones detected by our method and by the PMD. This group indicates that the proposed strategy is able to detect code clones found by other strategies. Since PMD has found the same group of clones, there is a higher expectation of being true positives. The third group is the most important for this study. It includes the code clones found by our method and not found by the PMD.

In a controlled environment, code clones of the third group (found by our method and not found by the PMD) were evaluated by 25 developers by means of code inspection. The preliminary results show that, in general, more than 90% of subjects agree with the code clones found by our method. Subjects did not confirm only 2 out of 12 code clone candidates. Even these cases, the decision whether they are code clones or not was not a consensus. Indeed, in these two cases, subjects were divided half to half in their opinions. Therefore, results so far indicate that analysis of method calls is a prominent strategy for detecting code clones.

## 1.4 Dissertation Outline

This dissertation is structured in seven chapters as follows.

Chapter 2 provides a background overview of code clone detection.

Chapter 3 explains the proposed method in order to detect code clones using sequence of method calls.

Chapter 4 presents the 14 systems analyzed using the proposed method along with data collected.

Chapter 5 presents a comparison of the proposed method and PMD results.

Chapter 6 describes a user study conducted in order to evaluate our results by means of code inspection.

Chapter 7 summarizes the results and indicates future work.

# Chapter 2

## Background and Related Work

Code clone detection is an active research area of software engineering. The crescent number of publications and conferences dedicated to this theme confirms its importance. This chapter overviews background information for this dissertation and discusses some related work on code clone detection. The main goal of this work is to propose a new method for code clone detection based on similar sequences of method calls. Therefore, this chapter intends to detail related background before present the proposal method in Chapter 3. This chapter additionally reviews some related work in the literature regarding code clone detection and compares them with our work.

This chapter is structured as follows. Section 2.1 provides a background overview of code clone, introducing its concept, classification, importance in software engineering, and needs for detection. Section 2.2 presents some current code clone detection techniques, outlining the most adopted detection strategies. Section 2.3 discusses some selected works related to code clone detection and compares them with our work. Finally, final remarks of this chapter are presented in Section 2.4

### 2.1 Code Clones

Code clones are exactly or nearly similar code fragments within a single software system [Mondal et al., 2014] or in cross-project context [Oliveira et al., 2015]. Programmers copy and paste existing code as helper practice on new development or maintenance of existing features. Usually, copied code contains tested algorithm, clear solution, good ideas, and suchlike reasons for cloning. In addition, some external factors, such as time pressure or productivity evaluation in lines of code, lead to cloning practice [Ducasse et al., 1999].

There are two main kinds of similarity between code fragments: textual similar and functional similar code clones. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text) [Roy and Cordy, 2009; Keivanloo et al., 2014]. The first kind of clone is often the result of copying a code fragment and pasting into another location. A code fragment *frag2* is a clone of another code fragment *frag1* if they are similar by some given definition of similarity. That is,  $f(\text{frag1}) = \text{frag2}$  where  $f$  is the similarity function. Two fragments that are similar to each other form a clone pair (*frag1*, *frag2*). Whenever more than two fragments are similar, they form a clone class or clone group [Koschke et al., 2006].

Different types proposed in the literature express the degree of similarity [Göde and Koschke, 2011]. One of the most common classification is Types 1, 2, and 3 for textual similar [Bellon et al., 2007] and Type 4 for functional similar [Gabel et al., 2008] code clones. Figure 2.1 shows a code snippet which transverses a database cursor. For each record, a new Project object is added to a list. Finally, this list is returned. This code snippet, taken from one of our case studies, is used as original source code of all four types of code clones.

```

Original
ResultSet rs = stmt.executeQuery(query);
boolean ok;
for (ok = rs.first(); ok; ok = rs.next())
{
    results.add(new Project(rs.getInt(1)));
}
return results;
```

**Figure 2.1.** Original Source Code for Code Clone Types Example.

<pre style="margin: 0;"> Type 1 ResultSet rs == stmt.executeQuery(query); boolean ok; // Cursor control; for (ok == rs.first(); ok; ok == rs.next()) {     results.add(new Project(rs.getInt(1))); } return results;</pre>	<pre style="margin: 0;"> Type 2 ResultSet e = stmt.executeQuery(query); boolean hasV; for (hasV=e.first(); hasV; hasV=e.next()) {     projects.add(new Project(e.getInt(1))); } return projects;</pre>
--	--

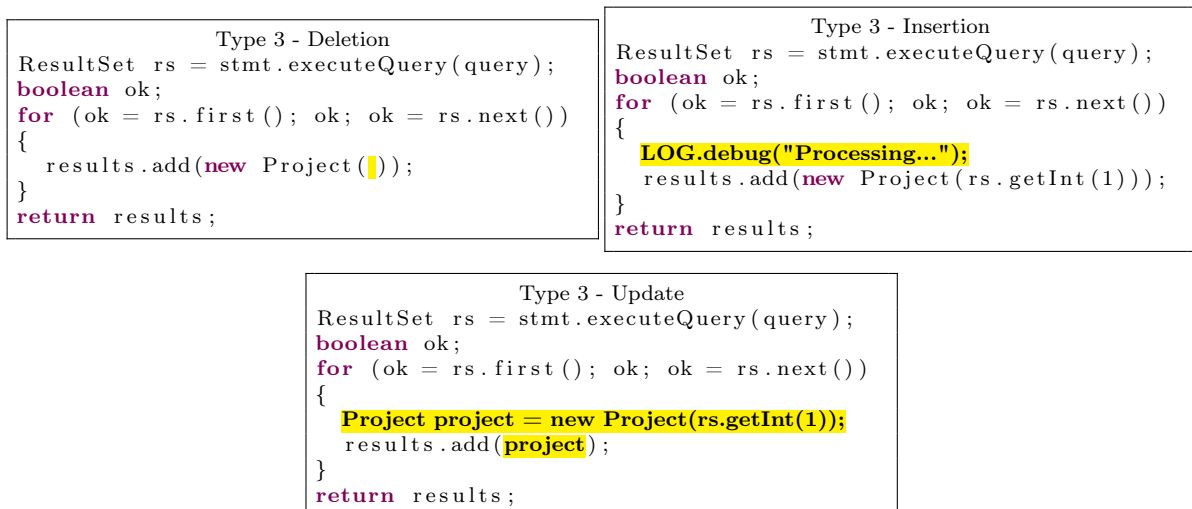
**Figure 2.2.** Code Clones Type 1 and Type 2.

The four specific types are commonly defined as [Roy et al., 2009]:

- **Type 1:** identical code fragments except for variations in white space, tabs, layout, and comments. Figure 2.2 shows in the left box an example of code clone Type 1. Comparing to original code (Figure 2.1), the highlighted areas present

some removed spaces, a new comment, and changed curly brace position. These changes are irrelevant for the compiler, characterizing the code clone Type 1;

- **Type 2:** structurally/syntactically similar code fragments, except for changes in identifiers, literals, types, layout, and comments. Figure 2.2 shows in the right box an example of code clone Type 2. Comparing to original code (Figure 2.1), the highlighted areas present variables renamed, along with some removed blank spaces, and changed curly brace position. The generated code is identical since the code fragment is structurally/syntactically similar, characterizing the code clone Type 2;
- **Type 3:** code fragments that have been copied with further modifications like statement insertions/deletions in addition to changes in identifiers, literals, types, and layouts. Figure 2.3 shows in the top left box an example of code clone Type 3. Comparing to original code (Figure 2.1), the highlighted area presents an instruction removed, *rs.getInt(1)*. The code fragment is similar to the original one, except for the removed piece. Figure 2.3 also shows in the top right box an example of code clone Type 3. Comparing to original code (Figure 2.1), the highlighted area presents an instruction added, *LOG.debug()*. The code fragment is similar to the original one, except for the added piece. Finally, Figure 2.3 shows in the bottom box a third example of code clone Type 3. Comparing to original code (Figure 2.1), the highlighted area presents an instruction changed, *Project project = new Project(rs.getInt(1))*. The code fragment is similar to the original one, except for the changed piece including a new variable. All these three



**Figure 2.3.** Code Clones Type 3.

```

Type 4
ResultSet rs = stmt.executeQuery(query);
if (!rs.first()) {
    return results;
}
do {
    results.add(new Project(rs.getInt(1)));
} while (rs.next());
return results;

```

**Figure 2.4.** Code Clone Type 4.

boxes represent code fragments modified regarding one or more instructions in comparison with original code, characterizing the code clone Type 3;

- **Type 4:** functionally similar code fragments without being textually similar. Figure 2.4 show an example of code clone Type 4. Comparing to original code (Figure 2.1), the highlighted areas present the rewritten code which has the same functionality of original one, but implemented with a different syntax. This new implementation is computational identical of its original counterpart, although syntactically different, characterizing the code clone Type 4.

In this work, we focus on textual similar code clones, although our method, detailed in Chapter 3, should be able to detect some types of functional similar code clones.

## 2.2 Code Clone Detection Techniques

Several works have been proposed in the literature for code clone detection [Ducasse et al., 1999; Oliveira et al., 2015; Gabel et al., 2008; Koschke et al., 2006; Krinke, 2001; Wahler et al., 2004; Hummel et al., 2010; Komondoor and Horwitz, 2001; Baxter et al., 1998; Paiva and Figueiredo, 2014]. Although the techniques vary considerably, previous work [Roy et al., 2009] presents the most adopted code clone detection strategies and compares the methods. The most common code clone detection strategies are based on tree [Wahler et al., 2004], program dependency graph (PDG) [Krinke, 2001; Komondoor and Horwitz, 2001], text [Johnson, 1993], token [Hummel et al., 2010], metrics [Marinescu, 2004; Paiva and Figueiredo, 2014; Kontogiannis, 1997], and hybrid techniques [Göde and Koschke, 2011].

Two techniques that use graphs are tree-based and program dependency graph-based. In the tree-based clone detection strategy [Baxter et al., 1998], a program under observation is first transformed into an abstract syntax tree (AST). For every



sub-tree in the AST, a hash value is computed and identical sub-trees are identified via identical hash values. In order to detect similar (not identical) sub-trees, the sub-trees have to be pairwise compared. Variables, statements, operations, among other elements that compose AST nodes, have influence on clone detection. In the program dependency graph-based clone detection strategy (PDG), starting from every pair of matching nodes, isomorphic sub graphs for ideal clones are constructed, which can be replaced by function calls automatically. This strategy cannot analyze big programs due to limitations of the underlying PDG generating infrastructure [Komondoor and Horwitz, 2001].

Other two techniques for code clone detection are text-based and token-based. A text-based clone detection strategy aims to find clones that differ at most in code format layout (spacing, tabs, and alignment) and comments. This technique uses textual extraction, hash calculation, and further comparison in order to detect similar fragments of code as candidates to code clone. Upon text extraction, some normalization is applied. Typically, transformation of source code in a pretty-print format making layout unified before analysis [Johnson, 1993; Ducasse et al., 1999]. A token-based clone detection strategy [Hummel et al., 2010] claims to provide cloning information for clone management. An index is created by the first time and new code can be added later. This strategy provides a way to handle real-time monitoring of code clones during development.

Some code clone detection strategies are applied alone, such as metrics based strategy, or mixed with others as hybrid clone detection strategies. In the metrics-based clone detection strategy [Roy et al., 2009], metrics for code fragments are gathered and sorted in a specific order. The code fragments can be units such as files, classes, functions, and statements. Searching for clones is the second part of the process, where similar metrics indicate code clone. A hybrid clone detection strategy mixes two (or more) techniques in order to use the best of both worlds in a complementary way, for example, mixing tree-based and metrics-based strategies. In this hybrid clone detection strategy, metrics are calculated on top of an AST generated by parsing the source code.

Type 4 code clones are well known hard to detect by existing techniques. It appears that only PDG-based techniques are likely to produce good results with this kind of code clones. PDG-based techniques use data and control flow information, which remains unchanged across reordering of declarations and data independent statements. However, even PDG-based tools are unlikely to detect some Type 4 code clone scenarios without exhaustive source transformation [Roy et al., 2009]. In addition, practical detection of code clones remains an open research field. For this reason, this work proposes a detection method inspired by the PDG-based techniques. However, in-

stead of creating a complete graph of dependencies, we focus on sequences of similar dependencies by means of method calls.

## 2.3 Related Work

Several works were proposed in the literature for code clone detection [Roy et al., 2009]. Although the techniques vary considerably, as presented in Section 2.2, the most common strategies are based on program dependency graph (PDG) [Krinke, 2001; Komondoor and Horwitz, 2001], text based [Johnson, 1993], tree based [Wahler et al., 2004], token-based [Hummel et al., 2010], and hybrid techniques [Göde and Koschke, 2011].

One of the first experiments in terms of code clone compared three state-of-the-art clone detection and two plagiarism detection tools [Burd and Bailey, 2002]. The authors were able to verify all clone candidates. The case study used only one modest size system. The validation subjectivity makes their findings less than definitive.

A token-based strategy [Hummel et al., 2010] claims to provide real-time cloning information for clone management of very large systems. Their work relies on being, at the same time, incremental and scalable. An index is created by the first time and new code can be added later. This provides a way to handle real-time monitoring of code clones during development. We have such characteristic in the proposed method (Chapter 3) because we use a hash list that can be incremented upon new code analysis. However, token-based methods are somehow dependent of the same type and number of variables, statements, and all elements transformed in tokens. This limitation is not the case of our strategy because only method calls are considered. The method calls are also tokens because they have exactly the same full name all over the code.

Another work available in the literature is based on PDG [Komondoor and Horwitz, 2001]. Starting from every pair of matching nodes, they construct isomorphic sub graphs for ideal clones, which can be replaced by method calls automatically. They cannot analyze big programs due to limitations of the underlying PDG generating infrastructure. The main similarity with our method calls analysis strategy is that clones found are also likely to be meaningful computations, thus candidates for extract method refactoring. As an important advantage, our method has no restrictions regarding the analyzed program size.

In another structure comparing work [Baxter et al., 1998], a program under observation is first transformed on an abstract syntax tree (AST). For every sub-tree in the AST, a hash value is computed and identical sub-trees are identified via identical

hash values. In order to detect similar (not identical) sub-trees, the sub-trees have to be pairwise compared. Variables, statements, operations, among other elements that compound AST nodes, have influence on the clone detection. Changes of implementation style in the copied code, such as *return*, *continue*, or *break*; *for* or *while* lead to code clones Type 4, hardening the detection. Our method ignores those elements, increasing the detection of this kind of intricate code clone scenarios.

Most of those clone detection strategies presented in Section 2.2 are implemented as open source, freeware, or commercial tools. Although the tools have been tested against single software systems, the detection accuracy became worse when cross-projects detection take place. Oliveira et al. [2015] have shown that, in cross-project contexts, reasonable detection rates are only achieved for Type 1 code clones and by a limited number of tools. As examples, cross-projects detection is important for library creation across projects or even for plagiarism detection. However, previous work has shown that existing techniques fail to detect cross-project code clones [Oliveira et al., 2015].

## 2.4 Final Remarks

This chapter presented concept of code clone, its common classification, and research importance for software engineering. The code clone background structure started exposing reasons leading developers to create code clones, practice sometimes unavoidable. In a sequence, this chapter presented the two main kind of similarities, textual and functional, and the most common classifications regarding degree of similarity, Types 1, 2, 3, and 4. Complementing the background, this chapter presented the most adopted code clone detection techniques, which are text, tree, token, metrics, PDG, and hybrid. Finally, this chapter discussed some related work in the literature and compared with our proposed code clone detection method.

The goal of this chapter was to prepare the reader with concepts used in this dissertation, producing insights regarding code clone detection. In particular, this chapter prepares the reader for Chapter 3, which relates all the concepts in order to outline the detection method proposed in this dissertation work.



# Chapter 3

## The Proposed Method

The proposal of a new method for detecting code clone worth pursuing. The parts that compound a code clone can evolve independently, hindering further detection. This chapter describes the method proposed in this work for detecting code clones using sequence of method calls. Since this dissertation proposes a new method for detecting code clones, this chapter explains the method, methodology, and implementation, before report this method in action in Chapter 4.

This chapter is structured as follows. Section 3.1 provides the strategy behind detection of code clone using sequence of method calls. Section 3.2 describes the sequence of method calls concept. Section 3.3 introduces how the sequences are grouped. Section 3.4 details some aspects of the order of method calls in Java. Section 3.5 describes the size of coincident sequences. Section 3.6 concludes what the method considers as code clone candidate. Finally, Section 3.7 presents the final remarks of this chapter.

### 3.1 The Method Strategy

One of the reasons for a developer to copy and paste a code snippet from a part of a software system to another, creating a code clone, is to get the original computation. The original code does something that solves totally or partially the problem in analysis by the developer. After the code clone action, the developer can apply some changes. However, part of the original computation is supposed to be preserved, since the developer wanted the computation provided by original code. That is, the computation provided by the original order of method calls.

In order to achieve a specific goal, a method coordinates statements, handles data, and calls other methods. The order of these method calls usually matters and should be preserved. In other words, it is not possible to change the order of most method

calls; otherwise, the final computation results in something completely different. This work proposes to detect code clones identifying a sequence of method calls in the exact same order in different parts of a software system.

## 3.2 The Sequence of Method Calls

Figure 3.1 shows an example of sequence of method calls inside the method *countAvailablePapers*, which belongs to the class *PaperManager*. In the scope of this method, three method calls occur: *papers.size()*, *papers.get()*, and *paper.isAvailable()*. The methods *size()* and *get()* belong to the *List* class, since the parameter *papers* is declared as *List<Paper>*. The method *isAvailable()* belongs to the *Paper* class, since the variable *paper* is declared as *Paper*.

```
1 package com.example.managers;
2
3 import java.util.List;
4 import com.example.Paper;
5
6 public class PaperManager {
7
8     int countAvailablePapers(List<Paper> papers){
9         int total = 0;
10        for (int i = 0; i < papers.size(); i++){
11            Paper paper = papers.get(i);
12            if (!paper.isAvailable())
13                continue;
14            total++;
15        }
16        return total;
17    }
18
19    ...
```

**Figure 3.1.** Sequence of method calls inside method *countAvailablePapers*.

Our method identifies coincident sequences of method calls in different system locations and considers that these locations are performing similar computation. This coincidence could be a code clone if the locations have exact the same method calls. However, a system could have methods with the same names in different classes. For example, the method *toString()* is commonly implemented by various classes in a Java system. Therefore, method calls should be uniquely identified in the entire system. In

order to achieve this uniqueness, the complete class name, i.e., including the Java package, is added as prefix to the method name. For instance, the method calls of the example in Figure 3.1 are stored as *java.util.List.size*, *java.util.List.get*, and *com.example.Paper.isAvailable*. For simplification purposes, we omit packages in the examples of this dissertation.

### 3.3 The Sequences Grouped by Class or Method

The sequence of method calls are grouped by location where it is found. The most common group is a method inside a class, as shown in Figure 3.1. The sequence is stored as belonging to method *com.example.managers.PaperManager.countAvailablePapers*. However, the Java language allows method calls outside a method, although inside the class declaration scope. As an example, Figure 3.2 contains a snippet of the *EvictionResourceDefinition* class, part of JBoss [JBoss, 2016]. In the scope of this class, there are the method calls *pathElement*, *SimpleAttributeDefinitionBuilder*<sup>1</sup>, *getLocalName*, *setXmlName*, *setAllowExpression*, *setFlags*, *EnumValidator*<sup>1</sup>, *setValidator*, *ModelNode*<sup>1</sup>, *name*, *set*, *setDefaultValue*, and *build*. All these method calls occur outside a method of the *EvictionResourceDefinition* class.

Occurrences of the same sequence of method calls at different points of a system launches a suspicion of code clone. Our assumption is based on the fact that the same sequences of method calls do similar computation. As the number of identical method calls becomes higher, more likely a code clone is. To reinforce this idea, the fact that the sequence has method calls from different classes in the same order increases the probability of being a code clone. Since a class provides a specific service or encapsulates common behaviors, mixing method calls of different classes should lead to a more strong similar computation. Figure 3.3 shows an example. The (a) side in Figure 3.3 is a snippet of the ArgoUML system. The constructor *Lexer* (class *org.argouml.language.csharp.importer.csparser.main.Lexer*) calls the method *Hashtable.put* multiple times in a sequence. The (b) side in Figure 3.3 is a snippet of the PetStore system. The method *extractCreditCard* (class *com.sun.j2ee.blueprints.petstore.controller.web.actions.CustomerHTMLAction*) calls in a sequence the methods *HttpServletRequest.getParameter()*, *String.trim()*, *String.equals()*, *ArrayList.ArrayList()*, and *ArrayList.add()*. The (b) side sequence is a more complex computation and has a higher probability of being code clones when found duplicated, although the (a) side could be part of a code clone as well. This

---

<sup>1</sup>In this work, we consider object instantiation as a method call to the class constructor.

```

1 package org.jboss.as.clustering.infinispan.subsystem;
2
3 import org.jboss.as.controller.PathElement;
4 import org.jboss.as.controller
    ↪ SimpleAttributeDefinitionBuilder;
5 ...
6
7 public class EvictionResourceDefinition extends
    ↪ SimpleResourceDefinition {
8
9     static final PathElement PATH = PathElement.pathElement(
    ↪ ModelKeys.EVICTION, ModelKeys.EVICTION_NAME);
10
11     // attributes
12     static final SimpleAttributeDefinition STRATEGY = new
    ↪ SimpleAttributeDefinitionBuilder(ModelKeys.STRATEGY,
    ↪ ModelType.STRING, true)
13         .setXmlName(Attribute.STRATEGY.getLocalName())
14         .setAllowExpression(true)
15         .setFlags(AttributeAccess.Flag
    ↪ RESTART_ALL_SERVICES)
16         .setValidator(new EnumValidator<>(EvictionStrategy.
    ↪ class, true, false))
17         .setDefaultValue(new ModelNode().set(
    ↪ EvictionStrategy.NONE.name()))
18         .build();
19 ...

```

**Figure 3.2.** Sequence of method calls in the class scope.

work does not investigate whether calls to different methods are more likely to be a code clone. The example here intends to present our assumption that the same sequence of method calls does similar computation, leading to code clones.

## 3.4 The Order of Method Calls in Java

The order of method calls should be preserved in a sequence. This order should follow the exact order of code execution, since we are looking for similar computation. The identification of similar computations is not always trivial when parsing Java code, which requires understanding specific cases. The code execution usually follows the order it was written. That is, lines from top to bottom and according to the English



```

public Lexer (...) {
    ...

    keywords.put("byte", TokenID.Byte);
    keywords.put("bool", TokenID.Bool);
    keywords.put("char", TokenID.Char);
    keywords.put("double", TokenID.Double);
    keywords.put("decimal", TokenID.Decimal);
    keywords.put("float", TokenID.Float);
    keywords.put("int", TokenID.Int);
    keywords.put("long", TokenID.Long);
    keywords.put("object", TokenID.Object);
    keywords.put("sbyte", TokenID.SByte);

    ...
}

private CreditCard extractCreditCard(...) {
    ArrayList missingFields = null;
    String creditCardNumber = request.
        ↪ getParameter("credit_card_number")
        ↪ .trim();
    if (creditCardNumber.equals("")) {
        if (missingFields == null) {
            missingFields = new ArrayList();
        }
        // this need to be internationalized
        missingFields.add("Credit Card");
    }
    ...
}

```

(a) Sequence of the same method call multiple times.

(b) Sequence with method calls of different classes.

**Figure 3.3.** Variability of method calls in a sequence.

reading, from left to right. However, some execution cases do not match the order that the code was written. The two following examples illustrate cases like that.

As a first example, Figure 3.4 contains a snippet of the *DialogPresenter* class, part of FacebookAndroid. As shown in this code, the order of method calls is *getCallId* (1), *toString* (2), *getLatestKnownVersion* (3), and *setupProtocolRequestIntent* (4). This order is not the written order from top to bottom and from left to right. The reason is that the parameters of the method *setupProtocolRequestIntent* need first to be built. In addition, English reading is kept when *getCallId* is called before *toString*, since *toString* is a method of the object returned by *getCallId*.

```

NativeProtocol.setupProtocolRequestIntent(4)(appCall.getCallId()(1).toString()(2),
    NativeProtocol.getLatestKnownVersion()(3));

```

**Figure 3.4.** Snippet adapted from *DialogPresenter* class of *FacebookAndroid*.

As a second example, Figure 3.5 contains a *for* statement with numbered order of method calls execution. As shown in this code, the order of method calls is *first* (1), *hasNext* (2), *getInt* (3), and *next* (4). The Java language states that *next()* is called after all method calls inside the *for* scope, i.e., inside braces when present or next line when there are no braces. In this case, there is only one call inside the braces, *getInt()*.

```

for (rs.first()(1); rs.hasNext()(2); rs.next()(4)) {
    rs.getInt(1)(3);
}

```

**Figure 3.5.** Order of method calls in a *for* statement.

### 3.5 The Size of Coincident Sequences

Our work is based on searching and comparing the same sequences of method calls in different points of a system. In order to characterize this coincident sequence, we have established two parameters to determine its size:

1. The total number of coincident method calls;
2. The maximum size of continuous coincident method calls.

The first coincident method call is the sequence start and the last coincident method call is the sequence end. Therefore, the total number of coincident method calls is the count of all matches, starting from the first coincidence until the last coincidence. However, the sequences are not necessarily continuous when comparing to each other because the code where the coincident sequence was extracted can have method calls that do not belong to this sequence. That is, method calls between the sequence start and the sequence end code that are not coincident. The maximum size of continuous coincident method calls is the size of the longest sequence with all elements matching without any different call.

Figure 3.6 shows two methods with the same name *toMessage* extracted from (a) *ToolchainsBuildingException* and (b) *ModelBuildingException* classes of Maven. The coincident method calls are highlighted. The sequence starts at line 3 of both methods with the method call *StringWriter* and ends at line 23 with the method call *toString* in both methods. The size of the coincident sequence of method calls between the two methods is 17. In addition, we observe that the longest sequence of continuous coincident method calls is 7. This value occurs two times. First occurrence starts at line 3 and ends at line 7 of both methods, interrupted by the method call of *length* at line 8 of (b) side. The second occurrence of the longest sequence of continuous coincident method calls starts at line 8 of *ToolchainsBuildingException.toMessage* and at line 13 of *ModelBuildingException.toMessage*. It ends at line 13 of *ToolchainsBuildingException.toMessage* and at line 19 of *ModelBuildingException.toMessage*, interrupted by method call *getLocation* of (a) side.

Table 3.1 shows two sequences of method calls extracted from the code of Figure 3.6. The first and third columns are the lines where the method call was found. The second and fourth columns indicate the method calls found. The first two columns are data from the *ToolchainsBuildingException.toMessage* method. The last two columns are data from the *ModelBuildingException.toMessage* method. The method calls highlighted in bold are the coincident ones. That is, the ones found in both methods. For instance, the method call *StringWriter* is highlighted in bold because it was found on both

```

1 private static String toMessage( List<Problem> problems )
2 {
3     StringWriter buffer = new StringWriter( 1024 );
4     PrintWriter writer = new PrintWriter( buffer );
5     writer.print( problems.size() );
6     writer.print( ( problems.size() == 1 ) ? " problem was " : " problems were " );
7     writer.print( "encountered while building the effective toolchains" );
8     writer.println();
9     for ( Problem problem : problems )
10    {
11        writer.print( "[" );
12        writer.print( problem.getSeverity() );
13        writer.print( "]" );
14        writer.print( problem.getMessage() );
15        String location = problem.getLocation();
16        if ( !location.isEmpty() )
17        {
18            writer.print( "@" );
19            writer.print( location );
20        }
21        writer.println();
22    }
23    return buffer.toString();
24 }

```

(a) Method *toMessage* of *ToolchainsBuildingException* class.

```

1 private static String toMessage( String modelId, List<ModelProblem> problems )
2 {
3     StringWriter buffer = new StringWriter( 1024 );
4     PrintWriter writer = new PrintWriter( buffer );
5     writer.print( problems.size() );
6     writer.print( ( problems.size() == 1 ) ? " problem was " : " problems were " );
7     writer.print( "encountered while building the effective model" );
8     if ( modelId != null && modelId.length() > 0 )
9     {
10        writer.print( " for " );
11        writer.print( modelId );
12    }
13    writer.println();
14    for ( ModelProblem problem : problems )
15    {
16        writer.print( "[" );
17        writer.print( problem.getSeverity() );
18        writer.print( "]" );
19        writer.print( problem.getMessage() );
20        writer.print( "@" );
21        writer.println( ModelProblemUtils.formatLocation( problem, modelId ) );
22    }
23    return buffer.toString();

```

(b) Method *toMessage* of *ModelBuildingException* class.

**Figure 3.6.** Methods *ToolchainsBuildingException.toMessage* and *ModelBuildingException.toMessage* extracted from Maven.

methods at the same position. This highlight does not happen for the method *formatLocation* because it was found only at line 21 of the *ModelBuildingException.toMessage* method. In addition, the fifth column shows the total counter of coincident method calls and the sixth column shows the counter of continuous coincident method calls. We can observe that this last counter resets on every occurrence of a non-coincident method call.

**Table 3.1.** Sequences of method calls of *ToolchainsBuildingException.toMessage* and *ModelBuildingException.toMessage*.

<i>ToolchainsBuildingException.toMessage</i>		<i>ModelBuildingException.toMessage</i>			
<i>Line #</i>	<i>Method call</i>	<i>Line #</i>	<i>Method call</i>	<i>Coincident</i>	<i>Continuous</i>
3	<b>StringWriter</b>	3	<b>StringWriter</b>	1	1
4	<b>PrintWriter</b>	4	<b>PrintWriter</b>	2	2
5	<b>size</b>	5	<b>size</b>	3	3
5	<b>print</b>	5	<b>print</b>	4	4
6	<b>size</b>	6	<b>size</b>	5	5
6	<b>print</b>	6	<b>print</b>	6	6
7	<b>print</b>	7	<b>print</b>	7	7
		8	length		
		10	print		
		11	print		
8	<b>println</b>	13	<b>println</b>	8	1
11	<b>print</b>	16	<b>print</b>	9	2
12	<b>getSeverity</b>	17	<b>getSeverity</b>	10	3
12	<b>print</b>	17	<b>print</b>	11	4
13	<b>print</b>	18	<b>print</b>	12	5
14	<b>getMessage</b>	19	<b>getMessage</b>	13	6
14	<b>print</b>	19	<b>print</b>	14	7
15	getLocation				
16	isEmpty				
18	<b>print</b>	20	<b>print</b>	15	1
18	print	21	formatLocation		
21	<b>println</b>	21	<b>println</b>	16	1
23	<b>toString</b>	23	<b>toString</b>	17	2

## 3.6 The Code Clone Candidates

If two or more points in the system have the same sequence of method calls (with a minimum size), they form a code clone candidate. However, the ideal size of a sequence is not easy to be chosen. If the size is small, e.g. with three method calls, it is expected to find too many coincident sequences and, by consequence, too many false positive code clones. On the other hand, if the chosen size of a sequence is a big value, for example, 20 method calls, it is expected too many false negatives. In order to address these issues, our configuration contains sequence parameters to be changed through different experiments. The first parameter is the total number of coincident method calls. The second parameter is the maximum size of continuous coincident method calls. The code of Figure 3.6 is detected as code clone if the first parameter is at most 17 and the second parameter is at most 7. Table 3.1 shows in the latest two columns these two values respectively.

## 3.7 Final Remarks

This chapter described the proposed method for detecting code clone using sequence of method calls. First, we presented the idea behind considering a coincident sequence of method calls as being code clones. The proposed method was detailed by parts and these parts were related to each other.

The chapter objective was to provide knowledge about why, how, and which data was collected. In Chapter 4, the proposed method is applied to a set of 14 selected software systems.



# Chapter 4

## Automated Method in Action

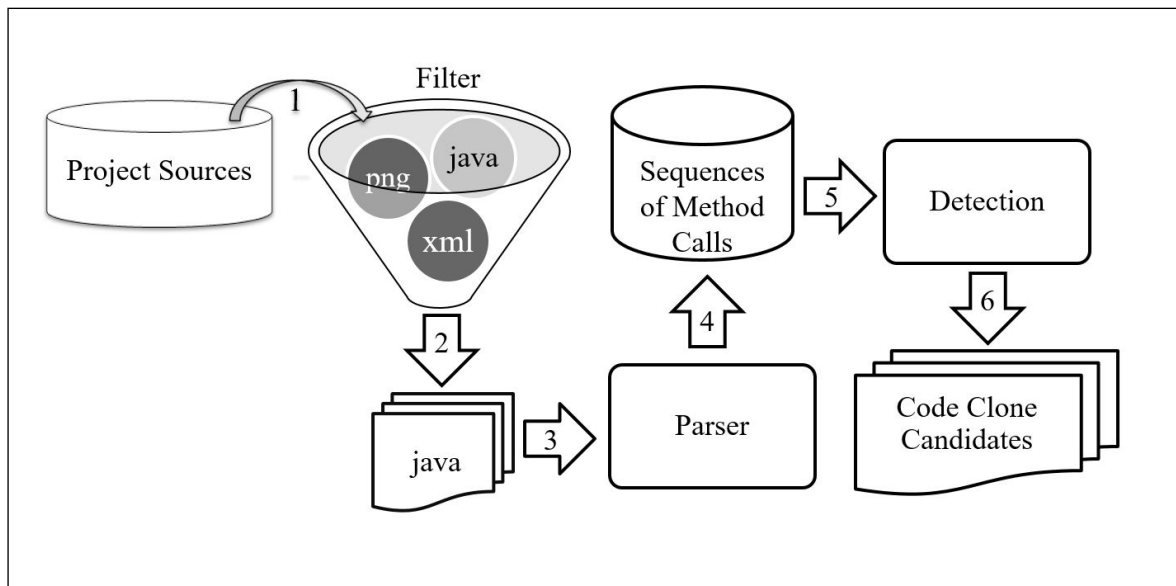
One of the contributions of this work is to provide an open source tool that implements the proposed method. This chapter provides details about this tool, developed for code clone detection following the proposed method. This chapter also presents the systems analyzed by the proposed method and the code clones found. In addition, this chapter shows examples of data gathered from real-world systems with different characteristics. Therefore, this chapter is a bridge between Chapter 3, where a new method for detecting code clones is proposed, and the next two chapters, which evaluate code clone detected by this method.

This chapter is structured as follows. Section 4.1 presents the developed tool, called McSheep, for code clone detection following the proposed method. Section 4.2 presents the criteria for selecting the systems and some metrics of the chosen projects. Section 4.3 presents the clones found by the proposed method. Finally, Section 4.4 presents the final remarks of this chapter.

### 4.1 Tool Support

We implemented a tool in order to extract code clones according to our method. The tool, called McSheep, is able to detect coincident sequences of method calls in different system points. The method workflow starts from filtering files from systems written in Java, proceeds by parsing Java files in order to select all method calls, continues by comparing coincident sequences of method calls at different locations, and finally points code clone candidates.

Figure 4.1 shows the processes executed by the McSheep in order to provide the results. These processes perform the following actions in the McSheep workflow:



**Figure 4.1.** McSheep Steps for Code Clone Detection.

Process 1) *Filter* - The input for Process 1 are the project files inside a specific folder, represented by “Project Sources”. McSheep is able to open all zip files in that folder and extract files. This process, represented by a filter, receives as input all the extracted files and ignores all the non-Java artifacts, outputting only the Java files to be parsed on the next process.

Process 2) *Parser* - This process receives as input the Java files filtered on the previous process and proceeds parsing each one. The parser used is JavaParser<sup>1</sup>. JavaParser is a Java 1.8 Parser with Abstract Syntax Tree (AST) generation and visitor support. The AST records the source code structure, Javadoc, and comments. JavaParser implements the Visitor Pattern [Gamma et al., 1994]. For each Java element visited in the source file, a specific visitor method is called. This visitor implementation allows the tool to select what is important, i.e., method calls. Of course, comments, expressions, statements, and any other elements are ignored. This feature differs from traditional methods. Therefore, only analysis of method calls takes place. For the sake of this study, the data to be collected is method calls, but some other Java elements are important and used as follows.

McSheep collects the data with help of the JavaParser engine. JavaParser receives a Java file as input and reports to the tool all classes found inside

<sup>1</sup><http://javaparser.github.io/javaparser/>



that file, along with import statements. These imports are used to identify the complete package where a class came from.

For each class, McSheep extracts and stores all methods belonging to that class (always using the JavaParser visitor pattern schema). For each method, the tool extracts and stores all method calls. Taking the Figure 3.1 as example, McSheep stores the method *countAvailablePapers* and the calls *List.size()*, *List.get()*, and *Paper.isAvailable()*. At this time, the imports collected are used to uniquely identify a class and hence a call. Therefore, McSheep stores the calls as *java.util.List.size*, *java.util.List.get*, and *com.example.Paper*. McSheep also records the exact order of occurrence for each call inside the method. The result of the parsing process is a database with the sequences of method calls.

Process 3) *Detection* - This process receives as input the database with all method calls of the system grouped by methods. The goal of this step is the search of code clones. Each unique method call is a key in a hash list which value is a list of methods where such method call was found. In addition, it is stored the execution order that this method call occupies, once the method executes a sequence of method calls in a specific order. McSheep transverses the hash list searching for keys with more than one method in their list. Once a method call is located in two or more methods of the system, McSheep compares these methods. All method calls in positions above and below the current method call in a transversed hash list are then compared.

The total number of coincident method calls and the maximum size of continuous coincident method calls are compared with McSheep configuration. If these values are equals or greater than configured threshold, McSheep considers this sequence as a code clone candidate.

Table 4.1 shows the hash list created as result of parsing the entire code of Figure 4.2, where the complete package name was omitted. Each hash element contains the method call as key and a list of method locations/position where that method call was found. For instance, the method call *List.get* was found at position (2) in method *foo1* and at position (2) in method *foo2*.

Observe that, as this data structure is a hash list, the order of keys is not under control. *List.get* is shown first in the hash list intentionally. In addition, there is a hash list of methods with each method as key and all method calls, ordered as they

```

1  int foo1(List<Paper> papers){
2      int total = 0;
3      for (int i = 0; i < papers.size(); i++) {
4          Paper paper = papers.get(i);
5          if (!paper.isAvailable())
6              continue;
7          total++;
8      }
9      return total;
10 }
11
12 int foo2(List<Paper> list) {
13     int count = 1, papers = 0;
14     while (count <= list.size()) {
15         if (list.get(count-1).isAvailable()) {
16             papers = papers+1;
17         }
18         count = count+1;
19     }
20     return papers;
21 }

```

**Figure 4.2.** Code clone detected by coincident sequence of method calls.

<i>key</i>	<i>value</i>
<i>Method call</i>	<i>List of methods (position)</i>
<i>List.get</i>	<i>foo1</i> (2), <i>foo2</i> (2)
<i>List.size</i>	<i>foo1</i> (1), <i>foo2</i> (1)
<i>Paper.available</i>	<i>foo1</i> (3), <i>foo2</i> (3)

**Table 4.1.** Hash list with method call as key and where call occurs as values.

appear in respective method, as value. Table 4.2 shows this hash list. For instance, the first element of this hash list is the method *foo1* as key. The value of this element is the list of calls *List.size*, *List.get*, and *Paper.available* ordered as they are executed in *foo1* method.

<i>key</i>	<i>value</i>
<i>Method</i>	<i>List of method calls</i>
<i>foo1</i>	<i>List.size</i> , <i>List.get</i> , <i>Paper.available</i>
<i>foo2</i>	<i>List.size</i> , <i>List.get</i> , <i>Paper.available</i>

**Table 4.2.** Hash list with method as key and its method calls as values.

Figure 4.2 contains an example of a code clone that could be detected by coincident sequences of method calls. The algorithm retrieves the first key: *List.get*. There is a list of methods where *List.get* is called: *foo1* and *foo2*, both as second call in the order of calls. Then, the algorithm lookup the list of methods (which contains *foo1* and *foo2* as values). The algorithm interrupts the verification for *List.get* when detects that there is another coincidence above: *List.size*. Only when the first method call coincident is found that the algorithm will navigate through all method calls in order to retrieve the complete sequence. Moreover, this navigation happens when *List.size* is the key being checked. Starting from *List.size*, the algorithm detects the whole chain of coincidences. Now, a code clone with a three-call sequence is pointed.

## 4.2 Selected Systems

The proposed method presented in Chapter 3 starts with selection of the target projects. These software projects are supposed to contain all sort of code clones. Therefore, some criteria shall be settled in order to select projects as representative as possible within a large set of possibilities. Finally, once selected, these projects are ready to be submitted to McSheep in order to view the proposed method in action.

With this objective in mind and using a set of criteria commonly adopted in similar studies involving code analysis [Vidal et al., 2015], we selected 14 software projects. All chosen systems are written in Java, open source, and have an active community of users and contributors. Along with these features, the systems were also selected from different domains and with varying sizes. The system sizes started from the smallest with 1,116 lines of code to the biggest with 392,405 lines of code. The number of classes varies from 13 to 6,452.

Table 4.3 shows the systems selected for analysis using the proposed method. The first column contains the system name (*Systems*), followed by the column with number of code lines (*LOC*), and Number of classes (*Classes*). For instance, Mobile Media is a system with 3,025 lines of code and 51 classes.

## 4.3 Code Clone Candidates

The method proposed in Chapter 3 was applied on each selected system. Table 4.4 lists these projects in the first column. The second and third columns present, for each system, the numbers of total method calls and distinct method calls. This values show that all systems have method calls scattered across the classes. This observation is

**Table 4.3.** Systems selected for analysis.

<i>Systems</i>	<i>NLOC</i>	<i>Classes</i>
Restaurantr	1,116	13
Telestrada	3,660	213
Learn Engh	2,114	41
Mobile Media	3,015	51
Ecommerce 21	3,883	96
Health Watcher	5,990	88
Ecommerce 2	56,734	718
PetStore	17,866	308
Junit	25,916	392
Facebook Android	36,277	250
Restaurant Open	39,231	406
Maven	78,471	938
ArgoUML	195,363	1,922
JBoss	392,405	6,452

true since distinct method calls are always smaller than the total number of method calls. The ratio of distinct calls by total calls is a mean of about 35%.

**Table 4.4.** Method calls scattering.

<i>System</i>	<i>Total</i>	<i>Distinct</i>
Restaurantr	399	230
Telestrada	688	304
Learn Engh	920	381
Mobile Media	1,419	449
Ecommerce 21	1,706	561
Health Watcher	2,301	491
Ecommerce 2	5,987	1,982
PetStore	6,105	2,352
Junit	8,221	2,932
Facebook Android	12,773	4,428
Restaurant Open	19,416	5,616
Maven	31,267	9,543
ArgoUML	82,435	23,582
JBoss	160,040	51,755

Once the number of distinct method calls is smaller than total of method calls, we observe the possibility of method calls been spread all over the system. In addition, there is the possibility of finding the same sequence of method calls in different points of code. Table 4.5 confirms this possibility. This table shows each system in the first column, followed by the results for three configurations in the subsequent second, third,

**Table 4.5.** Code clones candidates per configuration.

<i>System</i>	<i>Configuration</i>		
	<i>10-5</i>	<i>15-7</i>	<i>20-10</i>
Restaurantr	0	0	0
Telestrada	1	0	0
Learn Engh	7	2	0
MobileMedia	15	8	4
Ecommerce 21	103	42	11
Health Watcher	70	21	11
Ecommerce 2	617	203	88
PetStore	56	20	8
JUnit	0	0	0
Facebook Android	7	1	1
Restaurant Open	378	168	62
Maven	58	19	9
ArgoUML	771	175	47
JBoss	888	273	102
<b>Total</b>	<b>2,971</b>	<b>932</b>	<b>343</b>

and fourth columns. The title of result columns contains the configuration values separated by dash. The values are respectively the minimum size of coincident method calls and the minimum size of continuous coincident method calls. For instance, the title 10-5 of the second column means code clones detected with at least 10 coincident method calls and at least 5 continuous coincident method calls. Therefore, Table 4.5 shows that our proposed method detected 56 code clone candidates in PetStore matching 10-5 configuration criteria.

For illustration, one of these code clone candidates found by our method is shown in code snippets of Figure 4.3. This sequence of method calls in this pair of methods is what McSheep detected as code clone candidate. The sequence of method calls is *getDataSource*, *getConnection*, *toString*, *printSQLStatement*, *buildSQLStatement*, *executeQuery*, *getString*, *getString*, *getMessage*, *CatalogDAOSysException*, and *closeAll*. These 11 method calls are in the exact same order in two different methods, satisfying the minimum number of coincident method calls, configured as 10. The longest sequence of continuous coincident method calls is: *getDataSource*, *getConnection*, *toString*, *printSQLStatement*, *buildSQLStatement*, and *executeQuery*. These six method calls are in the exact same continuous sequence in the two methods, satisfying the minimum size of continuous coincident method calls, configured as 5.

Observing the code snippet of both methods, we detect similar computation: connect to database, debug, build and execute query, handle the result, handle exception,

and close resources. The types and data handled by method calls are different, but the logic is similar. Chapters 5 and 6 evaluate whether or not these candidates are actual code clones.

```

1 public Category getCategory(String categoryID, Locale locale) throws
  ↪ CatalogDAOSysException {
2     Connection connection = null;
3     ResultSet resultSet = null;
4     PreparedStatement statement = null;
5     try {
6         connection = getDataSource().getConnection();
7         String[] parameterValues = new String[] { locale.toString(), categoryID };
8         if (TRACE) {
9             printSQLStatement(sqlStatements, XML_GET_CATEGORY, parameterValues);
10        }
11        statement = buildSQLStatement(connection, sqlStatements, XML_GET_CATEGORY,
  ↪ parameterValues);
12        resultSet = statement.executeQuery();
13        if (resultSet.first()) {
14            return new Category(categoryID, resultSet.getString(1), resultSet.get-
  ↪ String(2));
15        }
16        return null;
17    } catch (SQLException exception) {
18        throw new CatalogDAOSysException("SQLException: " + exception.getMessage());
19    } finally {
20        closeAll(connection, statement, resultSet);
21    }
22 }

```

(a) Method *getCategory* of *GenericCatalogDAO* class.

```

1 public Page getProducts(String categoryID, int start, int count, Locale locale) throws
  ↪ CatalogDAOSysException {
2     Connection connection = null;
3     PreparedStatement statement = null;
4     ResultSet resultSet = null;
5     try {
6         connection = getDataSource().getConnection();
7         String[] parameterValues = new String[] { locale.toString(), categoryID };
8         if (TRACE) {
9             printSQLStatement(sqlStatements, XML_GET_PRODUCTS, parameterValues);
10        }
11        statement = buildSQLStatement(connection, sqlStatements, XML_GET_PRODUCTS,
  ↪ parameterValues);
12        resultSet = statement.executeQuery();
13        if (start >= 0 && resultSet.absolute(start + 1)) {
14            boolean hasNext = false;
15            List products = new ArrayList();
16            do {
17                products.add(new Product(resultSet.getString(1).trim(), resultSet.get-
  ↪ String(2), resultSet.getString(3)));
18            } while ((hasNext = resultSet.next()) (--count > 0));
19            return new Page(products, start, hasNext);
20        }
21        return Page.EMPTY_PAGE;
22    } catch (SQLException exception) {
23        throw new CatalogDAOSysException("SQLException: " + exception.getMessage());
24    } finally {
25        closeAll(connection, statement, resultSet);
26    }
27 }

```

(b) Method *getProducts* of *GenericCatalogDAO* class.

**Figure 4.3.** Methods *GenericCatalogDAO.getCategory* and *GenericCatalogDAO.getProducts* extracted from PetStore.

## 4.4 Final Remarks

This chapter detailed McSheep, a tool designed to extract data according to the proposed method. As described step by step, McSheep was developed for parsing Java files of systems in order to detect code clones candidates as per strategy. The chapter also presented the selected systems for data extraction along with the criteria used for choosing the target systems. In addition, this chapter presented the proposed method in action against 14 real-world software systems. Furthermore, the chapter showed the code clones candidates detected using coincident sequence of method calls.

The chapter goal was to show the code clones candidates detected upon execution of the proposed method. In Chapter 5, these code clone candidates are compared to the results of other code clone detection tool, called PMD.



# Chapter 5

## Comparative Evaluation

Several tools implement different strategies for code clone detection. Therefore, by choosing one of these tools and running against the same systems we used with McSheep, we have a mechanism for comparison with our proposed method. This chapter evaluates the code clone candidates detected in Chapter 4 by McSheep in comparison with the code clones detected by other tool, called PMD. The goal of this chapter is to compare our method presented in Chapter 3 with a different code clone detection method, the token-based implemented by the PMD. Although both methods detected some code clones candidates in common, the agreement between them was low. Therefore, the comparison results shown that both methods are complementary. Additionally, this chapter analyzes the threats to validity of this comparative evaluation.

This chapter is structured as follows. Section 5.1 provides the study settings, detailing the criteria to classify the detected code clone candidates. In addition, the section describes the chosen tool, called PMD, along with the reasons of this choice. Section 5.3 presents and analyzes the results. Section 5.4 discuss threats to validity of the comparison. Finally, Section 5.5 presents the final remarks of this chapter.

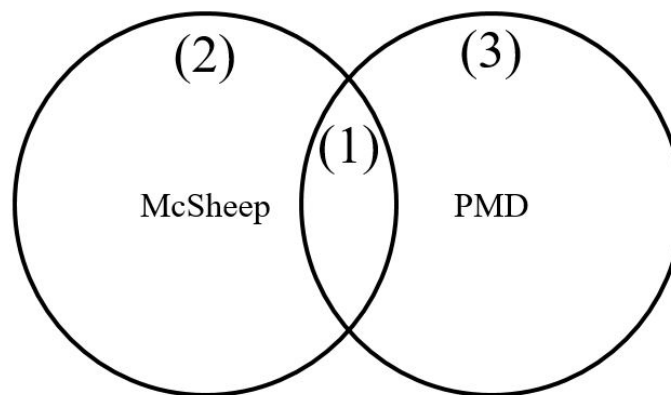
### 5.1 Study Settings

In order to validate the code clones candidates detected by the McSheep, the first strategy is to compare its results with results of a code clone detection tool commonly used by developers and established in the literature. Additionally, this tool uses a different strategy for code clone detection in order to reveal which scenarios are best covered by each method. The chosen tool is PMD [PMD, 2002], described in Section 5.2.

This comparison leads to three group of clones, as presented in Figure 5.1. This figure has a circle representing the code clone candidates found by the McSheep and

a circle representing the candidates found by other code clone detection tool, namely PMD. These two circles have an intersection set, which are the code clone candidates detected in common by both tools. Therefore, the three groups are, as shown in Figure 5.1:

- (1) Group of code clones candidates detected by both tools.
- (2) Group of code clones candidates detected exclusively by the McSheep.
- (3) Group of code clones candidates detected exclusively by the PMD.



**Figure 5.1.** Expected set of code clones candidates presented by tool.

According to Table 4.5 of Chapter 4, we ran McSheep with three different configurations: 10-5, 15-7, and 20-10. Following these configurations in this order, the results are always a set that contains the next configuration. That is, results of configuration 10-5 contains results of configuration 15-7, which by its turn contains results of configuration 20-10. As explained in Chapter 3, these dash-separated values are respectively the total number of coincident method calls and the maximum size of continuous coincident method calls. Therefore, if a total of 20 coincident method calls is reached, the total of 15 and 10 are also achieved. The formula in Figure 5.2 shows the mathematical relation between different ran configurations.

The results used for comparison are the ones gathered by executing McSheep with configuration 20-10. That is, this configuration considered code clone candidates when

$$C_{(10-5)} \supseteq C_{(15-7)} \supseteq C_{(20-10)}$$

**Figure 5.2.** Relation between different configurations.

two or more methods have at least 20 coincident method calls in the same order and at least 10 continuous coincident method calls. The reason of choice is that configuration 20-10 is the most conservative, since its results are present in all other sets. The contrary is not true.

Before discussing the results, it is important to understand that each tool has different strategy for code clone detection. While McSheep considers code clone only between methods, PMD can consider a code clone with lines spread over multiple methods. Therefore, a single PMD code clone can be matched in two or more McSheep code clones. In addition, all McSheep candidates are enclosed inside methods. The method signature is never part of the McSheep code clone, what differs from PMD analysis. These distinct approaches lead to conventions about the intersection set of code clones between both tools. This work considers that both tools found the same code clone when there is an intersection between start and end lines of each code clone pair. For example, if McSheep detects a code clone between lines 5 and 10 and PMD detects a code clone between lines 3 and 9, this work considers that both tools detected the same code clone.

## 5.2 The PMD Tool

As mentioned before, the chosen code clone detection tool was PMD<sup>1</sup> [PMD, 2002]. PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports Java, JavaScript, PLSQL, Apache Velocity, XML, and XSL. PMD has more than 6 million downloads since the project was started in 2002.

According to the documentation, PMD is a code quality tool that scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements.
- Dead code - unused local variables, parameters, and private methods.
- Suboptimal code - wasteful String/StringBuffer usage.
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops.
- Code clone detection - copied/pasted code means copied/pasted bugs.

---

<sup>1</sup><https://pmd.github.io/>

Along with this developer community acceptance, PMD is widely accepted in the literature for finding bugs and code smells [Rutar et al., 2004; Roy et al., 2009]. Specifically regarding code clone detection, PMD includes CPD, the copy-paste-detector. CPD detects code clones in several programming languages, such as Java, C, C++, and C#. CPD uses token-based for code clone detection, a strategy different from the one implemented by the McSheep, sequence of method calls based. Therefore, PMD, a state of the practice tool, was considered relevant choice for the comparison proposed in this chapter.

### 5.3 Results and Discussion

The same systems were submitted for analysis of PMD and the code clones detected were compared to the ones detected by the McSheep using the previous described configuration, 20-10. Table 5.1 shows the number of code clones candidates detected by the McSheep and the PMD for each analyzed system. Each row of this table contains the data of the respective system. The last row contains the global total. The columns with a positive sign contain the total of clones found by the tool above them and confirmed by the other tool. The columns with a negative sign contain the total of clones found exclusively by the tool above them, i.e., not found by the other tool. The last two columns contains Cohen's Kappa agreement [Gwet, 2014] of one tool in relation to another. For instance, the row for system ArgoUML shows that McSheep found 21 code clones that were also detected by the PMD. The same row also shows that PMD found 25 code clones that were also detected by the McSheep. Figure 5.3 presents one of these clones.

In addition, the McSheep found 26 code clones that are not detected by the PMD and the PMD found 253 code clones not detected by the McSheep. Therefore, the PMD agreement with McSheep results is 45% and the McSheep agreement with PMD results is 9%. The last row of this table shows that McSheep detected 343 (156+187) code clones candidates and PMD detected 853 (179+674) code clone candidates. The complete list of code clones, including the visualization, can be reached at <http://ampaiva.github.io/mcsheep>.

The first analysis of Table 5.1 shows that the overall agreement between both tools is low. Common levels of agreement for Kappa are Poor ( $< 0.20$ ), Fair (0.21 to 0.40), Moderate (0.41 to 0.60), Good (0.61 to 0.80), and Very Good (0.81 to 1.00) [Altman, 1991]. The average PMD agreement with McSheep is 45% and the McSheep agreement with PMD is 21%. The Moderate and Fair agreements indicate that McSheep and PMD

**Table 5.1.** Code Clones Found: McSheep and PMD.

<i>System</i>	<i>McSheep</i>		<i>PMD</i>		<i>Kappa</i>	
	<i>+</i>	<i>-</i>	<i>+</i>	<i>-</i>	<i>McSheep</i>	<i>PMD</i>
Restaurantr	0	0	0	0	0%	0%
Telestrada	0	0	0	9	0%	0%
Learn Engh	0	0	0	0	0%	0%
Mobile Media	1	3	1	2	25%	33%
Ecommerce 21	3	8	3	1	27%	75%
Health Watcher	7	4	6	1	64%	86%
Ecommerce 2	35	53	38	64	40%	37%
PetStore	3	5	3	28	38%	10%
JUnit	0	0	0	1	0%	0%
Facebook Android	1	0	1	3	100%	25%
Restaurant Open	22	40	27	46	35%	37%
Maven	5	4	5	26	56%	16%
ArgoUML	21	26	25	253	45%	9%
JBoss	58	44	70	240	57%	23%
<b>Total</b>	<b>156</b>	<b>187</b>	<b>179</b>	<b>674</b>	<b>45%</b>	<b>21%</b>

are complementary tools. Since both tools have a certain level of agreement, there are code clones that both tools detect in common. This is the first positive result for the strategy based on method calls used by the McSheep. Figure 5.3 presents a code clone detected by both McSheep and PMD tools. This pair of code snippets were extracted from the ArgoUML system. The left side is the snippet between lines 76 and 111 of the *ComponentInstanceNotationUml* class. The right side is the snippet between lines 76 and 111 of the *NodeInstanceNotationUml* class. The method call sequences are the same in both snippets and there are more than 20 method calls. Therefore, McSheep detected this code clone candidate. PMD also detected this code snippet as being code clone because there are more than 100 language equal tokens in the same order.

The 674 code clones detected exclusively by the PMD (Table 5.1) is an expected result. This total confirms that the method used by the McSheep cannot detect some types of code clones that the PMD token-based strategy can. Figure 5.4 presents a code clone detected only by the PMD. This pair of code snippets were extracted from the ArgoUML system. The left side is the snippet between lines 2095 and 2124 of the *CPPLexer* class. The right side is the snippet between lines 2132 and 2161 of the same class. This example shows that the number of equal language tokens between both sides is bigger than 100, favoring detection by the PMD token-based method. In addition, we can notice that there are only three method calls: *matchRange* at lines 2105 e 2142, *matchRange* again, this time at lines 2116 and 2153, and *match* at lines 2121 and 2158. The total number of method calls and the maximum sequence of method calls are three. Therefore, these code snippets do not match the McSheep configuration criteria used.

Other important result is the 187 code clone candidates detected exclusively by the McSheep (Table 5.1). This total reveals that the method used by the McSheep can

```

String s = text.trim();
if (s.length() == 0) {
    return;
}
if (s.charAt(s.length() - 1) == ';'') {
    s = s.substring(0, s.length() - 2);
}

String name = "";
String bases = "";
StringTokenizer tokenizer = null;

if (s.indexOf(":", 0) > -1) {
    name = s.substring(0, s.indexOf(":")).
        ↪ trim();
    bases = s.substring(s.indexOf(":") + 1)
        ↪ .trim();
} else {
    name = s;
}

tokenizer = new StringTokenizer(bases, ",")
    ↪ ;

List<Object> classifiers = new ArrayList<
    ↪ Object>();
Object ns = Model.getFacade().getNamespace(
    ↪ modelElement);
if (ns != null) {
    while (tokenizer.hasMoreElements()) {
        String newBase = tokenizer.
            ↪ nextToken();
        Object cls = Model.getFacade().
            ↪ lookupIn(ns, newBase.trim())
            ↪ ;
        if (cls != null) {
            classifiers.add(cls);
        }
    }
}

Model.getCommonBehaviorHelper().
    ↪ setClassifiers(modelElement,
    classifiers);
Model.getCoreHelper().setName(modelElement,
    ↪ name);

```

```

String s = text.trim();
if (s.length() == 0) {
    return;
}
if (s.charAt(s.length() - 1) == ';'') {
    s = s.substring(0, s.length() - 2);
}

String name = "";
String bases = "";
StringTokenizer tokenizer = null;

if (s.indexOf(":", 0) > -1) {
    name = s.substring(0, s.indexOf(":")).
        ↪ trim();
    bases = s.substring(s.indexOf(":") + 1)
        ↪ .trim();
} else {
    name = s;
}

tokenizer = new StringTokenizer(bases, ",")
    ↪ ;

List<Object> classifiers = new ArrayList<
    ↪ Object>();
Object ns = Model.getFacade().getNamespace(
    ↪ modelElement);
if (ns != null) {
    while (tokenizer.hasMoreElements()) {
        String newBase = tokenizer.
            ↪ nextToken();
        Object cls = Model.getFacade().
            ↪ lookupIn(ns, newBase.trim())
            ↪ ;
        if (cls != null) {
            classifiers.add(cls);
        }
    }
}

Model.getCommonBehaviorHelper().
    ↪ setClassifiers(modelElement,
    classifiers);
Model.getCoreHelper().setName(modelElement,
    ↪ name);

```

(a) Snippet code of the *ComponentInstanceNotationUml* class. (b) Snippet code of the *NodeInstanceNotationUml* class.

**Figure 5.3.** Code clone detected in the ArgoUML by both McSheep and PMD.

detect different code clones compared to the PMD token-based strategy. Figure 5.5 presents a code clone detected only by the McSheep. This pair of code snippets were extracted from the ArgoUML. The left side is the snippet between lines 80 and 105 of the *CrInvalidJoinTriggerOrGuard* class. The right side is the snippet between lines 79 and 116 of the *CrNoTriggerOrGuard* class. The sequence of right side is interrupted at line 86 and restart at line 91 (comment of line 90 in the left side is ignored by our method). There is a new interruption at line 98 and a new restart at line 105. These gaps presented at the right side are not enough to our method disconsider these

```

{
switch ( LA(1)) {
case 'a': case 'b': case 'c': case 'd':
case 'e': case 'f': case 'g': case 'h':
case 'i': case 'j': case 'k': case 'l':
case 'm': case 'n': case 'o': case 'p':
case 'q': case 'r': case 's': case 't':
case 'u': case 'v': case 'w': case 'x':
case 'y': case 'z':
{
    matchRange('a', 'z');
    break;
}
case 'A': case 'B': case 'C': case 'D':
case 'E': case 'F': case 'G': case 'H':
case 'I': case 'J': case 'K': case 'L':
case 'M': case 'N': case 'O': case 'P':
case 'Q': case 'R': case 'S': case 'T':
case 'U': case 'V': case 'W': case 'X':
case 'Y': case 'Z':
{
    matchRange('A', 'Z');
    break;
}
case '_':
{
    match('_');
    break;
}
default:

```

```

do {
switch ( LA(1)) {
case 'a': case 'b': case 'c': case 'd':
case 'e': case 'f': case 'g': case 'h':
case 'i': case 'j': case 'k': case 'l':
case 'm': case 'n': case 'o': case 'p':
case 'q': case 'r': case 's': case 't':
case 'u': case 'v': case 'w': case 'x':
case 'y': case 'z':
{
    matchRange('a', 'z');
    break;
}
case 'A': case 'B': case 'C': case 'D':
case 'E': case 'F': case 'G': case 'H':
case 'I': case 'J': case 'K': case 'L':
case 'M': case 'N': case 'O': case 'P':
case 'Q': case 'R': case 'S': case 'T':
case 'U': case 'V': case 'W': case 'X':
case 'Y': case 'Z':
{
    matchRange('A', 'Z');
    break;
}
case '_':
{
    match('_');
    break;
}
case '0': case '1': case '2': case '3':

```

(a) Snippet code (lines 2095-2124) of the *CPPLexer* class. (b) Snippet code (lines 2132-2161) of the *CPPLexer* class.

**Figure 5.4.** Code clone detected in the ArgoUML only by the PMD.

sequences as being code clone candidate. On the other hand, the token-based method used by the PMD does not consider this pair of code fragments as being code clones, since the gaps have relevant weight in its detection strategy.

## 5.4 Threats to Validity

This section discusses some threats to validity the comparative evaluation presented in this chapter. The threats list is by no means exhaustive. The goal here is to analyze the procedures taken during comparative evaluation and understand, for example, possible fails or wrong conclusions.

The comparison was performed with a single tool, PMD, which implements a specific code clone detection strategy, namely token-based. Hence, the set of tools and techniques are not completely representative. However, PMD is a state of practice tool, widely used by developers. In addition, the token-based technique was not able to detect some types of code clones detected by our method. Therefore, the subset represented by our choice for comparison is valid, since our proposal is a complementary method.

```

Object dv = Model.getFacade().getTarget(tr)
↪ ;
if (!(Model.getFacade().isAPseudostate(dv))
↪ ) {
    return NO_PROBLEM;
}

// WFR Transitions , OMG UML 1.3
Object k = Model.getFacade().getKind(dv);
if (!(Model.getFacade().
    equalsPseudostateKind(k,
        Model.getPseudostateKind().
            ↪ getJoin())) {
    return NO_PROBLEM;
}

boolean hasTrigger =
    (t != null Model.getFacade().
        ↪ getName(t) != null
        Model.getFacade().getName(t).
            ↪ length() > 0);
if (hasTrigger) {
    return PROBLEM_FOUND;
}
boolean noGuard =
    (g == null
        || Model.getFacade().getExpression(
            ↪ g) == null
        || Model.getFacade().getBody(Model.
            ↪ getFacade()
            .getExpression(g)) == null if (hasTrigger) {
        || Model.getFacade().getBody(Model.
            ↪ getFacade()
            .getExpression(g)).toString
            ↪ ().length() == 0);
}

Object target = Model.getFacade().getTarget(
    ↪ (transition));

if (!(Model.getFacade().isAPseudostate(
    ↪ target))) {
    return NO_PROBLEM;
}

Object trigger = Model.getFacade().
    ↪ getTrigger(transition);
Object guard = Model.getFacade().getGuard(
    ↪ transition);
Object source = Model.getFacade().getSource(
    ↪ (transition));

// WFR Transitions , OMG UML 1.3
Object k = Model.getFacade().getKind(target
    ↪ );
if (Model.getFacade().
    equalsPseudostateKind(k,
        Model.getPseudostateKind().
            ↪ getJoin())) {
    return NO_PROBLEM;
}
if (!(Model.getFacade().isAState(source))
    ↪ ) {
    return NO_PROBLEM;
}
if (Model.getFacade().getDoActivity(source)
    ↪ != null) {
    return NO_PROBLEM;
}
boolean hasTrigger =
    (trigger != null
        Model.getFacade().getName(trigger)
            ↪ != null
        Model.getFacade().getName(trigger)
            ↪ .length() > 0);
if (hasTrigger) {
    return NO_PROBLEM;
}
boolean noGuard =
    (guard == null
        || Model.getFacade().getExpression(
            ↪ guard) == null
        || Model.getFacade().getBody(
            Model.getFacade().getExpression
                ↪ (guard)) == null
        || Model
            .getFacade().getBody(Model.
                ↪ getFacade().
                ↪ getExpression(guard))
    )
}

```

(a) Snippet code of the *CrInvalidJoinTriggerOrGuard* class.(b) Snippet code of the *CrNoTriggerOrGuard* class.**Figure 5.5.** Code clone detected in the ArgoUML only by the McSheep.



Fourteen Java open source projects were randomly selected from software systems. Hence, the set of selected systems are not completely representative. However, all 14 systems have been developed by various organizations and contributors, are technically different, belong to varied domains, and provide substantially distinct functionalities. In addition, each tool detected, commonly or exclusively, code clone candidates, achieving our comparative proposal.

We used the default configuration for the PMD analysis, which is 100 tokens. Since McSheep is a new tool, there is no default configuration yet. Therefore, for the McSheep analysis, we chosen the most conservative configuration, which requires at least a 20-10 coincident sequence of method calls for considering a code clone candidate. Although should have ideal values for comparison, this choice seems to be fair, since both methods achieved complementary results.

The recorded values cannot be reliable, because there may be counting errors due to losses, duplications, or wrong case assessments. In order to minimize such counting errors, we automated the whole process between collecting the data and outputting the results. However, the automated process is also error prone. Therefore, for reducing the possible fails of the automated process, we tried to cover all known cases using unit tests, which improves the reliability of our code and results. In addition, we performed a visual inspection, manually checking various cases for quality assurance.

The execution of both tools were done using the same machine and operating systems. We take this specific care, although it is unlikely that the execution environment has influence on the results. We argue that the results are deterministic, not being influenced by performance factors.

The tools used for comparison implement methods to detect cloned code written in Java. Although the methods may be adapted for code clone detection in systems written using other languages, the results cannot be extended for these languages. For now, this extension is reserved for future work.

## 5.5 Final Remarks

This chapter evaluates of code clone candidates detected by our method in comparison with results of a state of the practice tool. First, we explained the candidates with the results of each tool. In a sequence, we presented the chosen tool, called PMD, and the reasons of this choice. Furthermore, we presented the comparison results, analyzing the common code clones candidates found and the differences. Finally, the chapter discussed some possible threats to validate the comparative evaluation.

The chapter objective was to evaluate the method by means of a comparison with results of a widely used tool. Chapter 6 is the next in sequence and describes a user study in order to evaluate the group of code clone candidates detected by our method and not by the PMD.

# Chapter 6

## User Study

Our method was able to detect code clone candidates not detected by the other tool, PMD. These code clone candidates need manual inspection in order to evaluate the results. This chapter details a user study for evaluation of code clones detected as described in Chapter 4. The goal of this chapter is to provide an empirical validation on top of data gathered using proposal method of Chapter 3. A survey containing 12 code clones detected by proposed method was submitted to 25 subjects. The subjects were asked to answer whether each pair of code is clone or not. In addition, the subjects should explain their reason of choice. Prior to the survey, subjects filled in a characterization form with background profile. Moreover, survey results were analyzed against subject profiles. In general, more than 90% of subjects agree with extracted code as being clones. Additionally, this chapter analyzes the threats to validity this research evaluating the most common aspects upon planning and conducting an experiment. We discuss four categories of threats: conclusion validity, internal validity, construct validity, and external validity.

This chapter is structured as follows. Section 6.1 provides the complete description of applied survey in order to evaluate code clones detected by proposal method. Section 6.2 presents the survey results and uses graphical analyzes for correlating opinions trends with subject profiles. Section 6.3 discuss threats to validity the experiment divided in subsections for each threat category. Finally, Section 6.4 presents the final remarks of this chapter.

## 6.1 Study Settings

The proposed method detected code clones in different systems with different sizes. Part of these code clones were detected exclusively by the McSheep and not detected by the PMD. Therefore, it is necessary to validate these specific McSheep results.

“Is it code clone?” This is the question raised for each code clone extracted by the proposed method. In the 14 systems, the method found 187 code clones that are not detected by the PMD. In order to answer this question for these code clones, we executed a user study. Subjects of this user study gave their opinion regarding a set of clones. They had to decide whether a pair of methods contains code clone. In addition, they were required to fill in a free text explaining the reason of choice. Due the high number of clones, just a sample of them was submitted to manual inspection.

The user study was composed of 25 subjects, i.e., 25 participants. The subjects were undergraduate and graduate students. Each subject filled in a characterization form<sup>1</sup>. This form intends to collect profile data. The form questions were regarding skills on the following areas: *English*, *Object Oriented Programming*, *Java Programming*, *Bad Smells*, *Code Clone*, and *General Development Experience*. The characterization form asked the subjects to self-classify their skills on related areas. The options of expertise level are *Few*, *Moderate*, or *Expert*. The complete characterization form is available at Appendix A. The goal of collecting profile data is further correlation with main survey described below.

The main survey was submitted to the subjects via specialized website<sup>2</sup>. Each subject was invited to answer the question “Is it code clone?” for 12 code snippet pairs. The code snippet pairs were code clones candidates detected by the proposed method. These code snippet pairs were randomly chosen from the total amount of code clones detected. Figure 6.1 shows the first case to be analyzed by the subjects during survey. The set of code clones was the same for all subjects. Appendix B contains all the survey questions formed by the 12 code snippet pairs.

All subjects received a thirty minutes training session previous starting answering the survey. This training was divided in two parts. First, a quick background regarding code clones. This background included an explanation about refactoring code clones [Fowler, 1999]. Although *Pull up* and *Extract Method* are the most recommended approaches for refactoring, only *Extracted Method* was exemplified due time restriction. Since *Pull Up* is somehow similar to *Extracted Method*, this had no significant relevance in the training.

---

<sup>1</sup><https://eSurv.org?u=characterizationform>

<sup>2</sup><https://eSurv.org/?u=iscodeclone>

\* 2.

```

public Node toDOM(Document document) {
    Element root = document.createElement(XML_PURCHASEORDER);
    root.setAttribute(XML_LOCALE, locale.toString());
    XMLDocumentUtils.appendChild(document, root, XML_ORDERID, orderId);
    XMLDocumentUtils.appendChild(document, root, XML_USERID, userId);
    XMLDocumentUtils.appendChild(document, root, XML_EMAILID, emailId);
    XMLDocumentUtils.appendChild(document, root, XML_ORDERDATE, dateFormat.format(orderDate));
    Element element = (Element) document.createElement(XML_SHIPPINGINFO);
    element.appendChild(shippingInfo.toDOM(document));
    root.appendChild(element);
    element = (Element) document.createElement(XML_BILLINGINFO);
    element.appendChild(billingInfo.toDOM(document));
    root.appendChild(element);
    XMLDocumentUtils.appendChild(document, root, XML_TOTALPRICE, totalPrice);
    root.appendChild(creditCard.toDOM(document));
    for (Iterator i = lineItems.iterator(); i.hasNext(); ) {
        LineItem lineItem = (LineItem) i.next();
        root.appendChild(lineItem.toDOM(document));
    }
    return root;
}

public Node toDOM(Document document) {
    Element root = document.createElement(XML_LINEITEM);
    XMLDocumentUtils.appendChild(document, root, XML_CATEGORYID, categoryId);
    XMLDocumentUtils.appendChild(document, root, XML_PRODUCTID, productId);
    XMLDocumentUtils.appendChild(document, root, XML_ITEMID, itemId);
    XMLDocumentUtils.appendChild(document, root, XML_LINENUM, lineNumber);
    XMLDocumentUtils.appendChild(document, root, XML_QUANTITY, quantity);
    XMLDocumentUtils.appendChild(document, root, XML_UNITPRICE, unitPrice);
    return root;
}

```

**Is there code clone above (Justify)?**

Figure 6.1. Survey Case 1.

The second part of the training took care of explaining how the survey will be. The training instructor taught the subjects regarding the survey and expected answers. That is, for each code snippet, the subject should start answering “Yes” or “No” for “Is it code clone?” question. At that time, an explanation regarding reason of choice was required as well. In order to provide to the subjects the maximum information at hand, the first steps of survey were the resume of training main points. Figure 6.2 shows the survey first step with instructions recalling what should be done in sequence. Figure 6.3 shows the survey second step containing an example of expected analysis of subject. In addition, the training slides were available to the subjects all the time upon survey application.

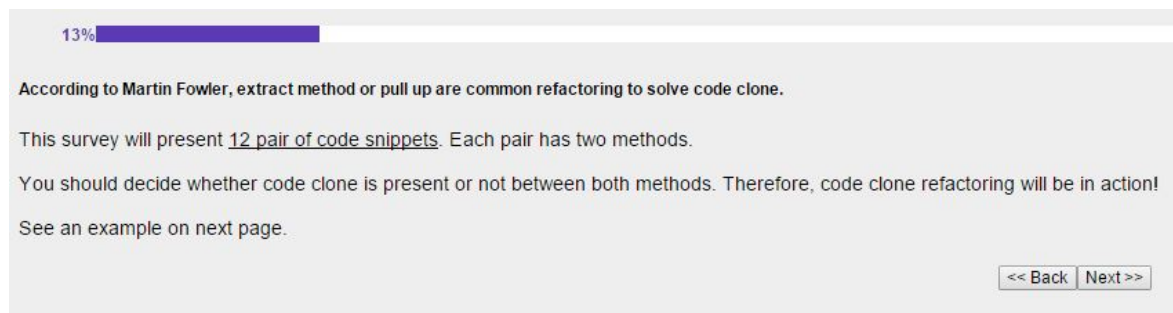


Figure 6.2. Survey First Step.

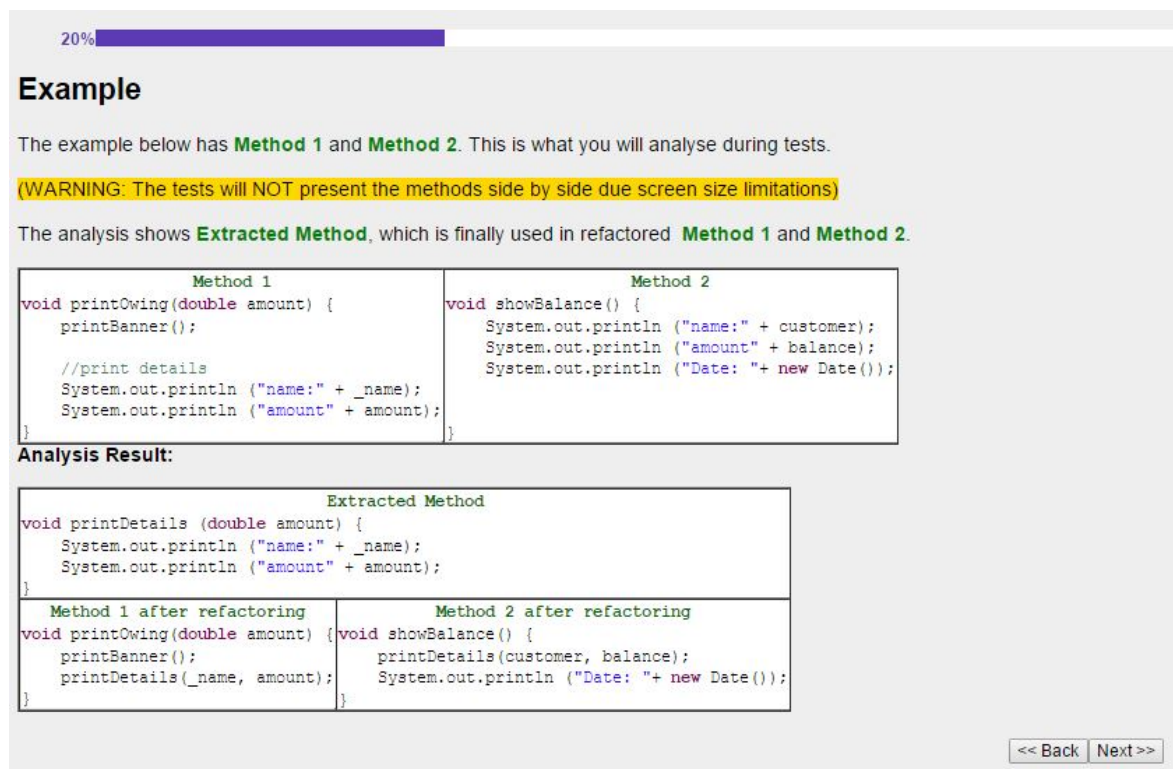
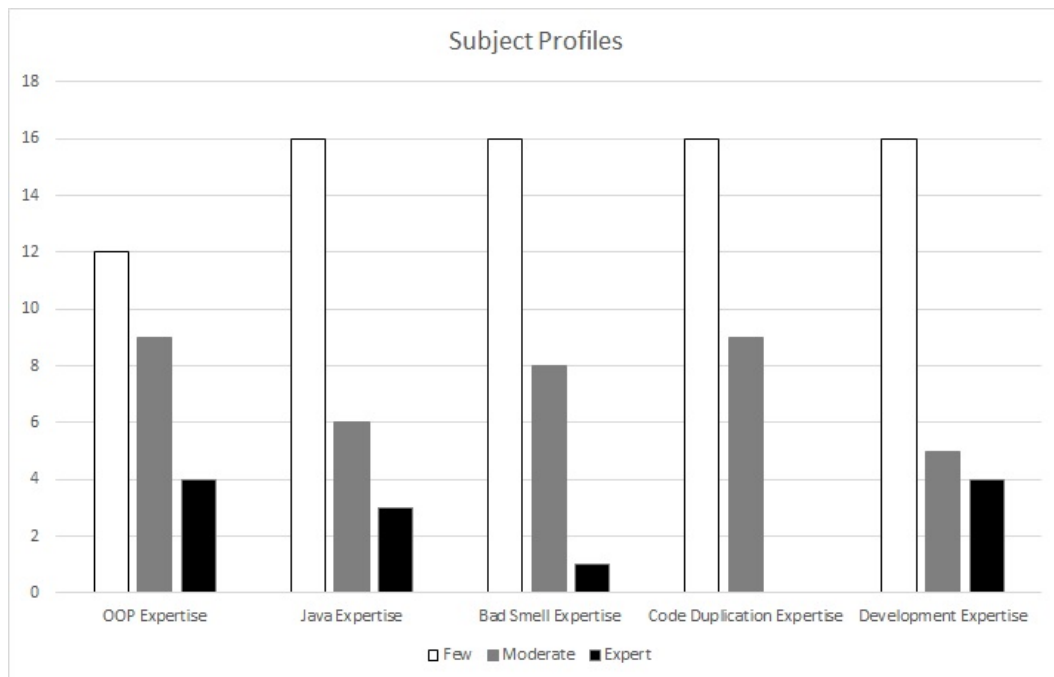


Figure 6.3. Survey Second Step.

Resuming, we applied a user study to examine the method detections. In a controlled environment, 25 developers, by means of code inspection, evaluated 12 code snippets extracted by the proposed method. The goal of this controlled experiment is to collect information regarding how developers see the code clone pointed by our proposal.

## 6.2 Results and Discussion

Figure 6.4 shows the subject profiles collected data. The *x-axis* contains columns for each knowledge level (*Few*, *Moderate*, and *Expert*) per profiles. The *y-axis* represents the absolute number of subjects per knowledge level. We observe that most subjects have just few experience level in the asked areas since the majority of them are undergraduate students.

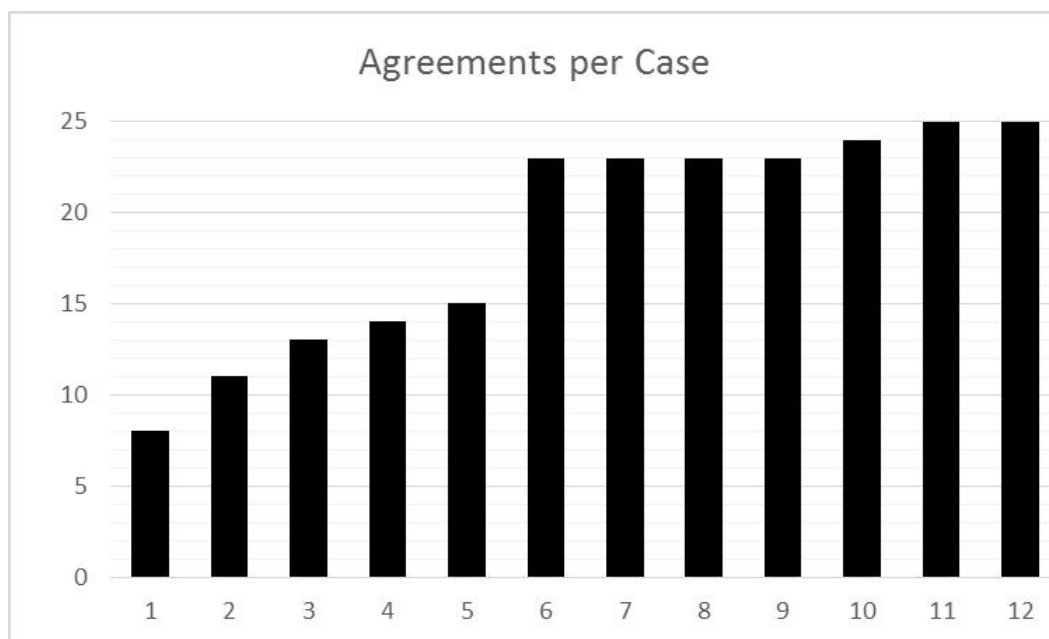


**Figure 6.4.** Subject Profiles.

Figure 6.5 shows overall agreements per case. This figure is ordered from lowest to highest agreement. That is, Case 1 of figure is not necessarily the first one in the survey. Said that, we hereafter name cases considering the order of acceptance instead of order of survey application.

Preliminary results show that, in general, more than 90% of subjects agree with extracted code as being clones. Only in 2 out of 12 cases, around 50% of subjects do not agree with the codes as being clones. Such cases are input for method algorithms improvement. Therefore, results so far indicate that a coincident sequence of method calls can be code clone.

We observe some relation between subject profile and trend to consider each case as code clone or not. These relations are not meaningful for the cases which option was total consensus (cases 11 and 12). However, for cases with certain level of disagreement it is important to analyze. That is the Case 1, represented in Figure 6.6. This figure has



**Figure 6.5.** Number of agreements per Survey Case.

the code snippet with the lowest level of acceptance. Four method calls are common in both methods: *ArrayList()*, *iterator()*, *hasNext()*, and *next()*. They are highlighted.

Although some cases got different answers, divergent judgment express how difficult is to have an agreement regarding cases like that. Below is transcript two different points of view for Case 1. Subjects with moderated knowledge in Object Oriented Programming wrote both points of view:

*“Yes. Although they are very different from each other, the codes do the same thing, which is to collect all the items to the Collection. The difference is that the first method differentiates each type that is collecting while the second method only reads. I consider this type of cloning type 4.”*

*“No. Although similar, the methods do very different calculations to be extracted to a generic method. The parameters of the methods and the objects returned are very different from the methods.”*

The subjects agree that textual code of each method is very different. However, while the first subject considers both code as doing same computation, the second subject does not. The point of view of each subject depends on seeing a way to refactor the code. Therefore, in some cases it is not possible to say precisely whether it is or not code clone only with subject judgment.

Figure 6.7 and Figure 6.8 are respectively the Code snippet 1 and Code snippet 2 that compound one of two cases with 100% of agreement. One could guess that



```

1 private Collection doWork(String xmlMessage) throws JMSEException, XMLDocumentException
    ↪ , MailContentXDE.FormatterException, FinderException, TransitionException {
2     ArrayList mailingList = new ArrayList();
3     PurchaseOrderLocal po = null;
4     OrderApproval approval = OrderApproval.fromXML(xmlMessage, entityCatalogURL,
    ↪ validateXmlOrderApproval);
5     Collection coll = approval.getOrdersList();
6     Iterator it = coll.iterator();
7     while (it != null && it.hasNext()) {
8         ChangedOrder co = (ChangedOrder) it.next();
9         String subject = MAIL_SUBJECT + co.getOrderId();
10        po = poHome.findByPrimaryKey(co.getOrderId());
11        String emailAddress = po.getPoEmailId();
12        mailContentXDE.setDocument(new DOMSource(co.toDOM()));
13        mailContentXDE.setLocale(LocaleUtil.getLocaleFromString(po.getPoLocale()));
14        String message = mailContentXDE.getDocumentAsString();
15        Mail mailMsg = new Mail(emailAddress, subject, message);
16        String xmlMail = mailMsg.toXML();
17        mailingList.add(xmlMail);
18    }
19    return mailingList;
20 }
21
22 public Collection getAllItems() {
23     Collection liColl = getLineItems();
24     if (liColl == null) return null;
25     ArrayList retVal = new ArrayList();
26     Iterator it = liColl.iterator();
27     while ((it != null) && (it.hasNext())) {
28         LineItemLocal loc = (LineItemLocal) it.next();
29         retVal.add(loc.getData());
30     }
31     return retVal;
32 }

```

**Figure 6.6.** Code snippet with the lowest level of acceptance.

this agreement should be a case of code clone with exact same text in each method. However, analyzing the case we notice that is a code clone with various code changes between methods. These changes include names of variables, order of declarations, insertions of method calls, handling of different object types, use of conditionals in distinct points, and even use of loop statements only in one side.

The following text is an interesting opinion regarding this case given by a subject that considers himself as a Java Programmer with few expertise level:

*“Yes. Many parts of the codes are exactly alike, with change of position between instructions. In addition, several variables have the same name. In both codes, you can see exactly the same instructions, connection, search parameters, creation of SQL statement, SQL query to run, and even SQL exceptions treatment.”*

The analysis above refers to generic cases, which represents the edges of agreement: lowest acceptance and highest acceptance. The subjects profile do not affect the conclusions. On the other hand, if they do, this was ignored for a while. Next analysis tries to associate subjects profile with trends of answers. Some graphs were

extracted from data in order to allow such analysis. The number of subjects for each profile varies considerably. Therefore, we treat them as percentage.

We start analyzing Figure 6.9, which shows Agreement per Object Oriented Programming expertise (OOP). The ones that have few OOP expertise tends to indicate more code snippets as code clone, especially for Cases 3, 4, 5, and 9. Subjects with few OOP expertise were not overcome in eight of 12 cases. This tendency can be justified by the fact that they avoid thinking about how to refactor the clone code. Maybe they only focus on snippets of coincidence detection.

Graph of Figure 6.10 shows how subjects with Java expertise behave when they analyses Java code searching for clones. Some cases are pairwise in terms of tendencies. Some examples are 5-9, 4-6-7-8. Although 2 is the opposite of 5-9 and 9, we can interpret the trends as somehow similar. The Java Expertise seems to be an important profile when a subject is deciding about code clones and the way they can be refactored.

Some trends deserve special attempting. That is the graph represented in Figure 6.11. That case compares the agreement per case for Bad Smell Expertise. Subjects that consider themselves as expert in Bad Smell gave unanimous opinion for all cases, regardless of these opinions being positive or negative.

No subjects self-evaluate themselves as code duplication expert. Figure 6.12 shows this phenomenal as a graph. However, there is a tendency of subjects with few expertise consider code snippets of survey as code clone. We can observe in the graph

```

1  public Category getCategory(String categoryID , Locale locale) throws
   ↪ CatalogDAOSystemException {
2      Connection connection = null;
3      ResultSet resultSet = null;
4      PreparedStatement statement = null;
5      try {
6          connection = getDataSource().getConnection();
7          String [] parameterValues = new String [] { locale.toString() , categoryID };
8          if (TRACE) {
9              printSQLStatement(sqlStatements , XML_GET_CATEGORY, parameterValues);
10         }
11         statement = buildSQLStatement(connection , sqlStatements , XML_GET_CATEGORY,
   ↪ parameterValues);
12         resultSet = statement.executeQuery();
13         if (resultSet.first()) {
14             return new Category(categoryID , resultSet.getString(1) , resultSet.
   ↪ getString(2));
15         }
16         return null;
17     } catch (SQLException exception) {
18         throw new CatalogDAOSystemException("SQLException: " + exception.getMessage());
19     } finally {
20         closeAll(connection , statement , resultSet);
21     }
22 }

```

**Figure 6.7.** Code 1: One of two cases with 100% of agreement.

```

1 public Page getProducts(String categoryID, int start, int count, Locale locale) throws
    ↪ CatalogDAOSystemException {
2     Connection connection = null;
3     PreparedStatement statement = null;
4     ResultSet resultSet = null;
5     try {
6         connection = getDataSource().getConnection();
7         String[] parameterValues = new String[] { locale.toString(), categoryID };
8         if (TRACE) {
9             printSQLStatement(sqlStatements, XML_GET_PRODUCTS, parameterValues);
10        }
11        statement = buildSQLStatement(connection, sqlStatements, XML_GET_PRODUCTS,
    ↪ parameterValues);
12        resultSet = statement.executeQuery();
13        if (start >= 0 && resultSet.absolute(start + 1)) {
14            boolean hasNext = false;
15            List products = new ArrayList();
16            do {
17                products.add(new Product(resultSet.getString(1).trim(), resultSet.
    ↪ getString(2), resultSet.getString(3)));
18            } while ((hasNext = resultSet.next()) && --count > 0);
19            return new Page(products, start, hasNext);
20        }
21        return Page.EMPTY_PAGE;
22    } catch (SQLException exception) {
23        throw new CatalogDAOSystemException("SQLException: " + exception.getMessage());
24    } finally {
25        closeAll(connection, statement, resultSet);
26    }
27 }

```

Figure 6.8. Code 2: One of two cases with 100% of agreement.

that only in two cases, 2 and 10, there was a superior acceptance by moderate expertise subjects in comparison with few expertise ones. The conclusion for Figure 6.12 is similar to of Figure 6.9, that is, more expertise, more caution.

All previous graphs reveals different point of view for Cases 5, 6, and 7. However, all four experts in development subjects completely agreed with such cases as being code clone. This can be observed in graph of Figure 6.13. Those cases have in common only the use of try/catch blocks. Probably, the subjects found the same way to refactoring this code.

That is interesting to see how background has influence in the way of thinking. Overall discussion regarding relationship between a subject profile and tendency of agreement with code clone is reserved for future work.

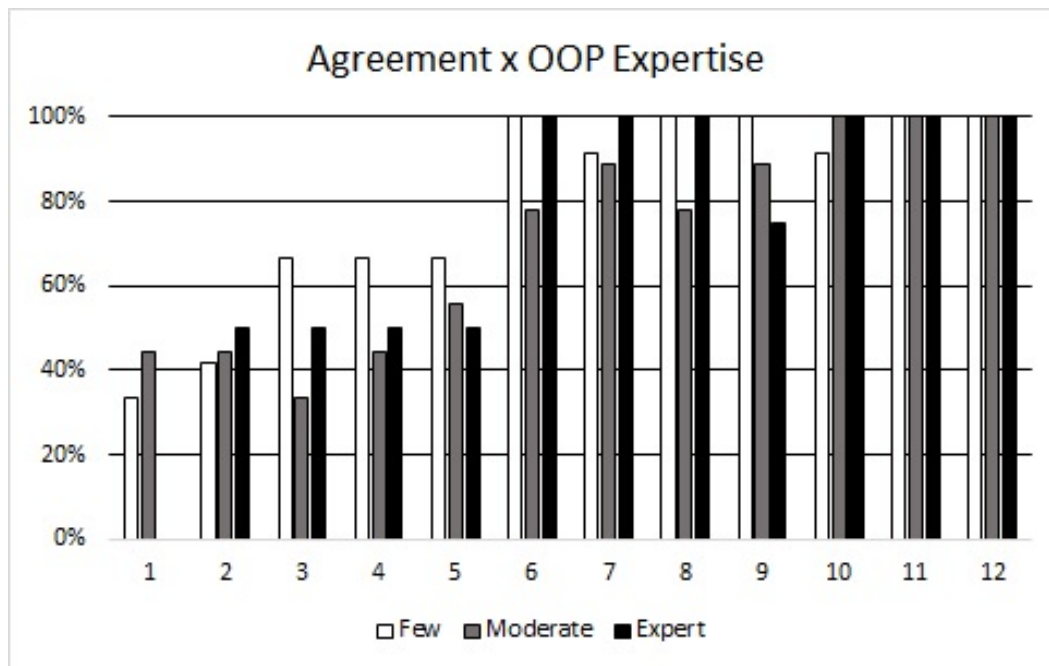


Figure 6.9. Agreement per Object Oriented Programming Expertise.

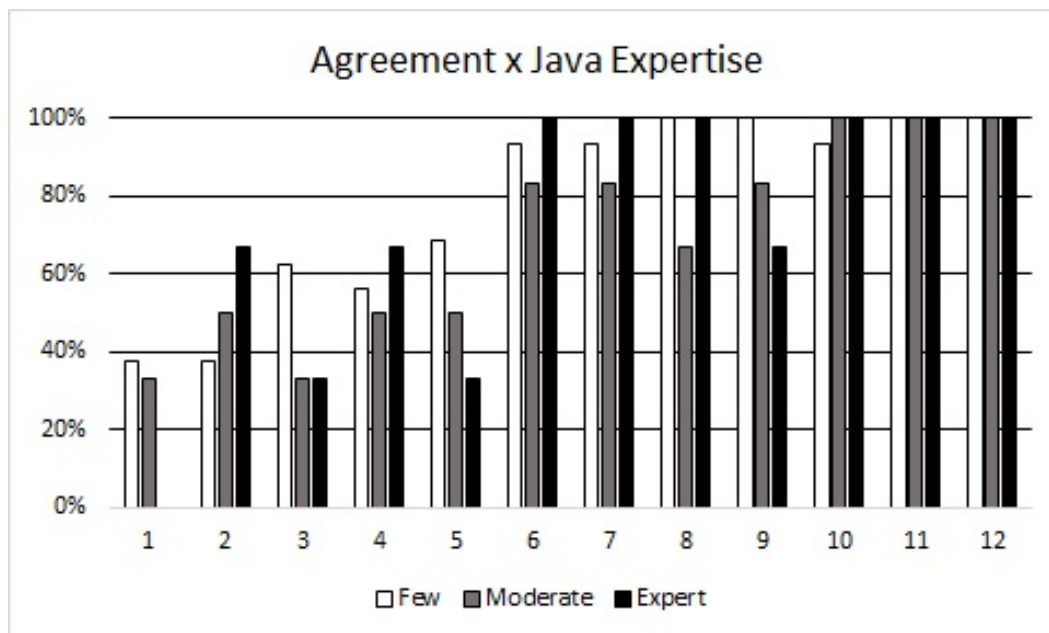


Figure 6.10. Percentage of agreement per Java Expertise.

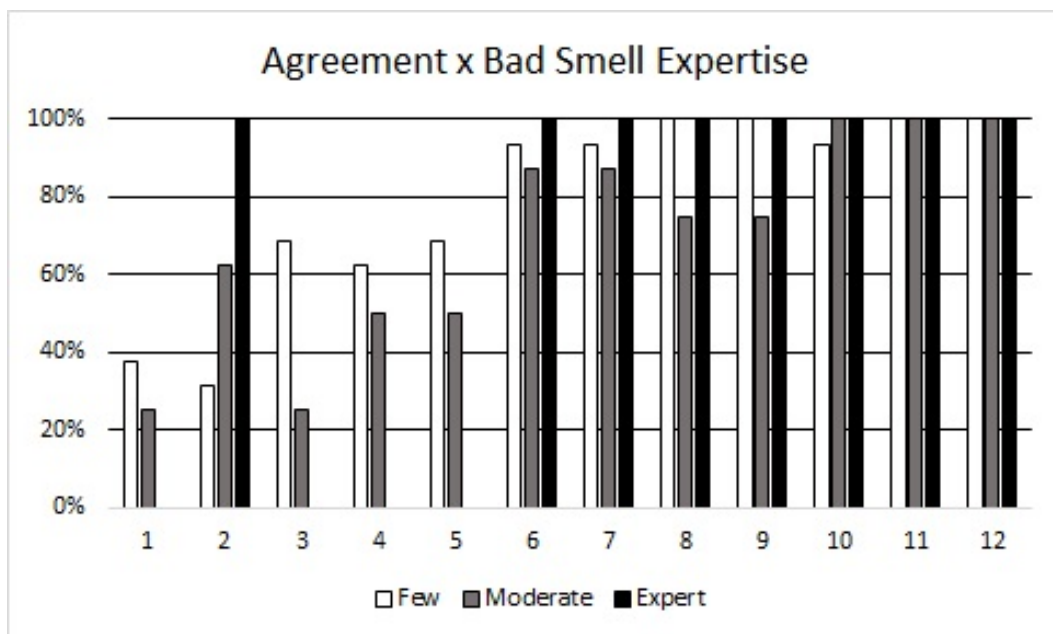


Figure 6.11. Percentage of agreement per Bad Smell Expertise.

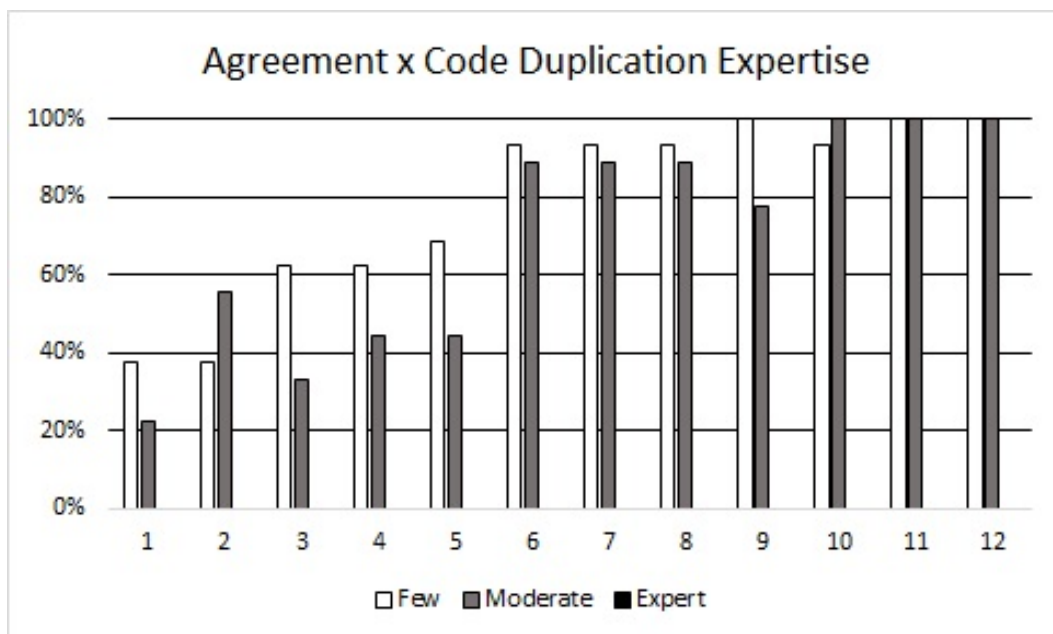
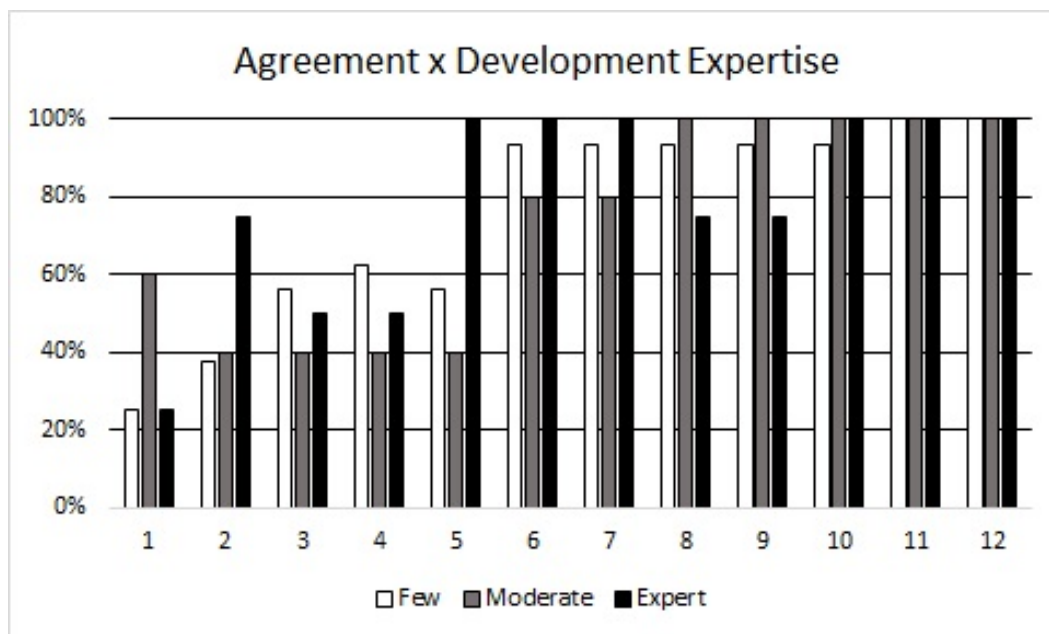


Figure 6.12. Percentage of agreement per Code Duplication Expertise.



**Figure 6.13.** Percentage of agreement per Development Expertise.

## 6.3 Threats to Validity

Since the user study involves several steps, the various threats to its validity needs analysis. This analysis allows us to validate the results for the selected code clone cases directly submitted to user study participants and expand to the entire set of code clones found by the method in action. Therefore, we analyzed whether the results can be generalized to a broader population. We discuss four categories of threats: construct validity, internal validity, external validity, and conclusion validity [Wohlin et al., 2012].

### 6.3.1 Construct Validity

Fourteen Java open source projects were randomly selected from software systems. Hence, the set of selected systems are not completely representative. However, all 14 systems have been developed by various organizations and contributors, are technically different, belong to varied domains, and provide substantially distinct functionalities.

### 6.3.2 Internal Validity

Our qualitative research method does not guarantee completeness of our results. Most of the results depend on the selected participants opinion and experience. However, we applied other code clone detection tools in order to avoid any infeasibility in our study. Furthermore, the authors know the benchmark of systems.

### 6.3.3 External Validity

We cannot claim that our findings can be held true for other software projects with larger size. Then, more studies on other systems are necessary to validate our findings. The subjects of our user study are students with different profiles. The use of students in experiments can directly influence the results, making the generality of our study be limited. However, works in the literature using students in experiments claim that the results of experiments with students are similar to the results of experienced professionals [Salman et al., 2015], so that the results are reliable with reality. The result shows that our approach is able to detect clones that are missed by other tools. Moreover, our approach misses clones detected by other tools. Therefore, we consider our method as a complementary approach to improve the state-of-art of clone detection mechanisms.

### 6.3.4 Conclusion Validity

Based on our data, we may conclude that there is a positive relationship regarding whether or not it is code clone. That is, subjects with higher knowledge in some fields tend to have a more critical view than those with lower knowledge, presenting the relationship between theory and observation. These threats to validity were mitigated because the authors have read the justifications of subjects. Thus, the conclusions are solidified in the justifications presented by the subjects in an experimental form.

## 6.4 Final Remarks

This chapter described the user study for evaluation of code clones detected which had 25 subjects with different developer profiles. First, we explained how code clones were selected for survey. In a sequence, we detailed the user study settings, which includes subjects training about code clones. Furthermore, we presented the survey application methodology, expected answers, and goals to achieve.

Continuing the chapter, it was presented the survey results with more than 90% of subjects agreeing with extracted code as being clones. The results were analyzed by means of graphs correlating subject profiles with given answers.

Finally, the chapter presented the possible threats to validate the user study.

The chapter objective was to validate the real data collected with a group of developers responsible for analyzing whether code clones detected are valid or not. Chapter 7, the next in sequence, points the conclusions of this work and includes discussions for future work.



# Chapter 7

## Conclusions and Future Work

In this research, we proposed a method to detect code clone using method calls analysis. The strategy was to find coincident sequence of method calls in different points of a system. Given a configured size, the coincident sequence of different method calls in two or more methods was a code clone detected. The proposed method was implemented by a tool, called McSheep. The proposed method was able to detect code clones using different configurations. For instance, McSheep detected 343 code clones with a sequence of at least 20 coincident method calls.

The same systems were submitted to analysis of another code clone detection tool, called PMD. The code clones detected by the PMD were compared to our method results. The code clones were categorized in three groups: the ones detected by both tools, the ones detected exclusively by the PMD, and the ones detected exclusively by the McSheep. The comparative evaluation indicated that both tools could be used in a hybrid strategy.

In order to evaluate the code clones detected exclusively by the proposed method and not detected by the PMD, a user study was conducted. In a controlled environment, 25 developers by means of code inspection evaluated code clones extracted by our method. The preliminary results show that, in general, more than 90% of subjects agree with the code clones presented. Only in 2 out of 12 cases, around 50% of subjects do not agree with the method results. Therefore, results so far indicate that method calls analysis is a valid strategy for detecting code clones.

The main contribution of this paper are:

- to propose a new technique to detect code clones using method calls analysis. Variables, operations, and control statements are ignored during analysis, which focus exclusively in method calls. This method is somehow different of other

techniques, being a complementary way of finding code clones using hybrid detection;

- to create a tool that implements the proposed method, making it available a free for downloading at <http://ampaiva.github.io/mcsheep/>;
- to compare the method results with a state of practice tool, called PMD;
- to conduct a user study with developers in order to evaluate the code clones found exclusively by our method.

Upon finishing this dissertation, we found many directions for future work. The list below is far from being exhaustive, presenting only some insights about what we or others interested in this subject can do from now on:

- provide a way to find the sequence size parameters that best fit a specific project. The approach taken here is most an empirical evaluation and the values used were the same for all analyzed projects. However, we believe that a research can determine a better choice according to the project characteristics;
- extend the research to other languages than Java;
- expand the analysis of private methods calls. Since private methods can only be called locally, this call is never identical in code clones and always interrupts an identical sequence of method calls;
- create a comparative study with other code clone detection methods, pointing the main advantages of each one and indicating possible uses as hybrid solutions;
- apply a new user study with more subjects, focusing in which type of code clone candidate is more likely and unlikely to be considered code clone.

# Bibliography

- Altman, D. G. (1991). *Practical Statistics for Medical Research*. Chapman & Hall.
- Baxter, I. D., Yahin, A., Moura, L., SantAnna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance (ICSM)*.
- Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering (TSE)*, pages 577--591.
- Burd, E. and Bailey, J. (2002). Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 36--43.
- Ducasse, S., Rieger, M., and Demeyer, S. (1999). A language independent approach for detecting duplicated code. *IEEE International Conference on Software Maintenance (ICSM)*.
- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.
- Gabel, M., Jiang, L., and Su, Z. (2008). Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 321--330.
- Gamma, E., Helm, R., Johnson, R., and J.Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Göde, N. and Koschke, R. (2011). Frequency and risks of changes to clones. *International Conference Software Engineering (ICSE)*, pages 311--320.
- Gwet, K. (2014). *Handbook of Inter-Rater Reliability: The Definite guide to Measuring the Extent of Agreement Among Raters*. Advanced Analytics, USA.

- Hummel, B., Juergens, E., Heinemann, L., and Conradt, M. (2010). Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 1--9. IEEE.
- JBoss (2016). Jboss. <http://www.jboss.org/>. Accessed: 2016-03-29.
- Johnson, J. H. (1993). Identifying redundancy in source code using fingerprints. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research: Software Engineering*, pages 171--183. IBM Press.
- Keivanloo, I., Rilling, J., and Zou, Y. (2014). Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 664--675.
- Khan, M., Roy, C., and Schneider, K. (2014). Active clones: Source code clones at runtime. *Proceedings of the Eighth International Workshop on Software Clones (IWSC)*.
- Komondoor, R. and Horwitz, S. (2001). Using slicing to identify duplication in source code. In *Eigth International Static Analysis Symposium (SAS)*, pages 40--56.
- Kontogiannis, K. (1997). Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE)*, pages 44--54.
- Koschke, R., Falke, R., and Frenzel, P. (2006). Clone detection using abstract syntax suffix trees. *Working Conference Reverse Engineering (WCRE)*, pages 253--262.
- Krinke, J. (2001). Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301--309.
- Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 350--359.
- Mondal, M., Roy, C., and Schneider, K. (2014). Late propagation in near-miss clones: An empirical study. *Proceedings of the Eighth International Workshop on Software Clones (IWSC)*.
- Oliveira, J., Fernandes, E., and Figueiredo, E. (2015). Evaluation of duplicated code detection tools in cross-project context. In *3rd Workshop on Software Visualization, Maintenance, and Evolution (VEM)*, pages 49--56.

- Paiva, A. and Figueiredo, E. (2014). Do concern metrics support code clone detection? In *11th Workshop on Software Modularity (WMOD)*, pages 130–136.
- PMD (2002). Pmd. <https://pmd.github.io/>. Accessed: 2016-03-29.
- Rattan, D., Bhatia, R., and Singh, M. (2013). Software clone detection: a systematic review. *Information and Software Technology (IST)*.
- Roy, C. K. and Cordy, J. R. (2009). A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 157–166.
- Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495. ISSN 0167-6423.
- Rutar, N., Almazan, C., and Foster, J. (2004). A comparison of bug finding tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256. ISSN 1071-9458.
- Salman, I., Misirli, A., and Juristo, N. (2015). Are students representatives of professionals in software engineering experiments? In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 666–676.
- Vidal, S. A., Bergel, A., Marcos, C., and Díaz-Pace, J. A. (2015). Understanding and addressing exhibitionism in java empirical research about method accessibility. *Empirical Software Engineering*, pages 1--34.
- Wahler, V., Seipel, D., Wolff, J., and Fischer, G. (2004). Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 128–135.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer, Norwell, MA, USA.



# Appendix A

## Characterization Form

The characterization form is available online at <https://eSurv.org?u=characterizationform>.

### Formulário de Caracterização

\* 1. Nome:

2. Curso/Semestre

\* 3. Leitura em inglês técnico  
 Básico  
 Intermediário  
 Avançado

\* 4. Programação Orientada a Objetos  
 Experiente  
 Moderado  
 Pouco  
 Nenhum

\* 5. Programação em Java  
 Experiente  
 Moderado  
 Pouco  
 Nenhum

\* 6. Bad Smell  
 Experiente  
 Moderado  
 Pouco  
 Nenhum

\* 7. Código Duplicado  
 Experiente  
 Moderado  
 Pouco  
 Nenhum

8. Você possui experiência de trabalho na área de desenvolvimento de software?  
 Não tenho experiência  
 Tenho experiência de até 1 ano  
 Tenho experiência de 1 ano a 3 anos  
 Tenho experiência de mais de 3 anos

Figure A.1. Characterization Form





# Appendix B

## Survey Questions

This appendix contains all questions of survey described in Section 6.1.

```
public Node toDOM(Document document) {
    Element root = document.createElement(XML_PURCHASEORDER);
    root.setAttribute(XML_LOCALE, locale.toString());
    XMLDocumentUtils.appendChild(document, root, XML_ORDERID, orderId);
    XMLDocumentUtils.appendChild(document, root, XML_USERID, userId);
    XMLDocumentUtils.appendChild(document, root, XML_EMAILID, emailId);
    XMLDocumentUtils.appendChild(document, root, XML_ORDERDATE, dateFormat.format(
        ↪ orderDate));
    Element element = (Element) document.createElement(XML_SHIPPINGINFO);
    element.appendChild(shippingInfo.toDOM(document));
    root.appendChild(element);
    element = (Element) document.createElement(XML_BILLINGINFO);
    element.appendChild(billingInfo.toDOM(document));
    root.appendChild(element);
    XMLDocumentUtils.appendChild(document, root, XML_TOTALPRICE, totalPrice);
    root.appendChild(creditCard.toDOM(document));
    for (Iterator i = lineItems.iterator(); i.hasNext(); ) {
        LineItem lineItem = (LineItem) i.next();
        root.appendChild(lineItem.toDOM(document));
    }
    return root;
}
```

```
public Node toDOM(Document document) {
    Element root = document.createElement(XML_LINEITEM);
    XMLDocumentUtils.appendChild(document, root, XML_CATEGORYID, categoryId);
    XMLDocumentUtils.appendChild(document, root, XML_PRODUCTID, productId);
    XMLDocumentUtils.appendChild(document, root, XML_ITEMID, itemId);
    XMLDocumentUtils.appendChild(document, root, XML_LINENUM, lineNumber);
    XMLDocumentUtils.appendChild(document, root, XML_QUANTITY, quantity);
    XMLDocumentUtils.appendChild(document, root, XML_UNITPRICE, unitPrice);
    return root;
}
```

Figure B.1. Question 1.

```

public static void main(String[] args) {
    if (args.length <= 1) {
        String fileName = args.length > 0 ? args[0] : "Invoice.xml";
        try {
            TPAInvoiceXDE invoiceXDE = new TPAInvoiceXDE();
            invoiceXDE.setDocument(new StreamSource(new FileInputStream(new File(
                ↪ fileName)), fileName));
            System.err.println("fileName: " + fileName + ", orderId=" + invoiceXDE.
                ↪ getOrderId() + " lineItemIds=" + invoiceXDE.getLineItemIds());
            System.exit(0);
        } catch (IOException exception) {
            System.err.println(exception);
            System.exit(2);
        } catch (XMLDocumentException exception) {
            System.err.println(exception.getRootCause());
            System.exit(2);
        }
    }
    System.err.println("Usage: " + TPAInvoiceXDE.class.getName() + " [file -name]");
    System.exit(1);
}

```

```

public static void main(String[] args) {
    if (args.length <= 1) {
        String fileName = args.length > 0 ? args[0] : "Mail.xml";
        try {
            Mail mail = Mail.fromXML(new StreamSource(new FileInputStream(new File(
                ↪ fileName)), fileName));
            System.out.println(Mail.fromXML(mail.toXML()).getContent());
            System.exit(0);
        } catch (IOException exception) {
            System.err.println(exception);
            System.exit(2);
        } catch (XMLDocumentException exception) {
            System.err.println(exception.getRootCause());
            System.exit(2);
        }
    }
    System.err.println("Usage: " + Mail.class.getName() + " [file -name]");
    System.exit(1);
}

```

Figure B.2. Question 2.

```

private Collection doWork(String xmlMessage) throws JMSEException, XMLDocumentException
↳ , MailContentXDE.FormatterException, FinderException, TransitionException {
    ArrayList mailingList = new ArrayList();
    PurchaseOrderLocal po = null;
    OrderApproval approval = OrderApproval.fromXML(xmlMessage, entityCatalogURL,
↳ validateXmlOrderApproval);
    Collection coll = approval.getOrdersList();
    Iterator it = coll.iterator();
    while (it != null && it.hasNext()) {
        ChangedOrder co = (ChangedOrder) it.next();
        String subject = MAIL_SUBJECT + co.getOrderId();
        po = poHome.findByPrimaryKey(co.getOrderId());
        String emailAddress = po.getPoEmailId();
        mailContentXDE.setDocument(new DOMSource(co.toDOM()));
        mailContentXDE.setLocale(LocaleUtil.getLocaleFromString(po.getPoLocale()));
        String message = mailContentXDE.getDocumentAsString();
        Mail mailMsg = new Mail(emailAddress, subject, message);
        String xmlMail = mailMsg.toXML();
        mailingList.add(xmlMail);
    }
    return mailingList;
}

```

```

public Collection getAllItems() {
    Collection liColl = getLineItems();
    if (liColl == null) return null;
    ArrayList retVal = new ArrayList();
    Iterator it = liColl.iterator();
    while ((it != null) && (it.hasNext())) {
        LineItemLocal loc = (LineItemLocal) it.next();
        retVal.add(loc.getData());
    }
    return retVal;
}

```

Figure B.3. Question 3.

```

public Collection getAllItems() {
    Collection liColl = getLineItems();
    if (liColl == null) return (null);
    ArrayList retVal = new ArrayList();
    Iterator it = liColl.iterator();
    while ((it != null) (it.hasNext())) {
        LineItemLocal loc = (LineItemLocal) it.next();
        retVal.add(loc.getData());
    }
    return (retVal);
}

```

```

public Collection processPendingPO() throws FinderException {
    ArrayList invoices = new ArrayList();
    Collection coll = supplierOrderLocalHome.findOrdersByStatus(OrderStatusNames.
        ↪ PENDING);
    if (coll != null) {
        Iterator it = coll.iterator();
        while ((it != null) (it.hasNext())) {
            SupplierOrderLocal order = (SupplierOrderLocal) it.next();
            String newInvoice = null;
            try {
                newInvoice = processAnOrder(order);
            } catch (XMLDocumentException xe) {
                System.out.println("OrderFulfillmentFacade:" + xe);
            }
            if (newInvoice != null) {
                invoices.add(newInvoice);
            }
        }
    }
    return invoices;
}

```

Figure B.4. Question 4.

```

public Category getCategory(String categoryID, Locale locale) throws
↳ CatalogDAOSystemException {
    Connection connection = null;
    ResultSet resultSet = null;
    PreparedStatement statement = null;
    try {
        connection = getDataSource().getConnection();
        String[] parameterValues = new String[] { locale.toString(), categoryID };
        if (TRACE) {
            printSQLStatement(sqlStatements, XML_GET_CATEGORY, parameterValues);
        }
        statement = buildSQLStatement(connection, sqlStatements, XML_GET_CATEGORY,
↳ parameterValues);
        resultSet = statement.executeQuery();
        if (resultSet.first()) {
            return new Category(categoryID, resultSet.getString(1), resultSet.
↳ getString(2));
        }
        return null;
    } catch (SQLException exception) {
        throw new CatalogDAOSystemException("SQLException: " + exception.getMessage());
    } finally {
        closeAll(connection, statement, resultSet);
    }
}

```

```

public Page getProducts(String categoryID, int start, int count, Locale locale) throws
↳ CatalogDAOSystemException {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        connection = getDataSource().getConnection();
        String[] parameterValues = new String[] { locale.toString(), categoryID };
        if (TRACE) {
            printSQLStatement(sqlStatements, XML_GET_PRODUCTS, parameterValues);
        }
        statement = buildSQLStatement(connection, sqlStatements, XML_GET_PRODUCTS,
↳ parameterValues);
        resultSet = statement.executeQuery();
        if (start >= 0 && resultSet.absolute(start + 1)) {
            boolean hasNext = false;
            List products = new ArrayList();
            do {
                products.add(new Product(resultSet.getString(1).trim(), resultSet.
↳ getString(2), resultSet.getString(3)));
            } while ((hasNext = resultSet.next()) && (--count > 0));
            return new Page(products, start, hasNext);
        }
        return Page.EMPTY_PAGE;
    } catch (SQLException exception) {
        throw new CatalogDAOSystemException("SQLException: " + exception.getMessage());
    } finally {
        closeAll(connection, statement, resultSet);
    }
}

```

Figure B.5. Question 5.

```

public void showMediaList(String recordName, boolean sort, boolean favorite) {
    if (recordName == null) recordName = getCurrentStoreName();
    MediaController mediaController = new MediaController(midlet, getAlbumData(), (
        ↪ AlbumListScreen) getAlbumListScreen());
    mediaController.setNextController(this);
    MediaListScreen mediaList = null;
    if (getAlbumData() instanceof ImageAlbumData) mediaList = new MediaListScreen(
        ↪ MediaListScreen.SHOWPHOTO);
    if (getAlbumData() instanceof MusicAlbumData) mediaList = new MediaListScreen(
        ↪ MediaListScreen.PLAYMUSIC);
    if (getAlbumData() instanceof VideoAlbumData) mediaList = new MediaListScreen(
        ↪ MediaListScreen.PLAYVIDEO);
    mediaList.setCommandListener(mediaController);
    mediaList.initMenu();
    MediaData[] medias = null;
    try {
        medias = getAlbumData().getMedias(recordName);
    } catch (UnavailablePhotoAlbumException e) {
        Alert alert = new Alert("Error", "The list of items can not be recovered",
            ↪ null, AlertType.ERROR);
        Display.getDisplay(midlet).setCurrent(alert, Display.getDisplay(midlet).
            ↪ getCurrent());
        return;
    }
    if (medias == null) return;
    if (sort) {
        bubbleSort(medias);
    }
    for (int i = 0; i < medias.length; i++) {
        if (medias[i] != null) {
            if (favorite) {
                if (medias[i].isFavorite()) mediaList.append(medias[i].getMediaLabel()
                    ↪ , null);
            } else mediaList.append(medias[i].getMediaLabel(), null);
        }
    }
    setCurrentScreen(mediaList);
}

```

```

private boolean playVideoMedia(String selectedMediaName) {
    InputStream storedMusic = null;
    try {
        MediaData mymedia = getAlbumData().getMediaInfo(selectedMediaName);
        incrementCountViews(selectedMediaName);
        if (mymedia instanceof MultiMediaData) {
            storedMusic = ((VideoAlbumData) getAlbumData()).getVideoFromRecordStore(
                ↪ getCurrentStoreName(), selectedMediaName);
            PlayVideoScreen playscree = new PlayVideoScreen(midlet, storedMusic, ((
                ↪ MultiMediaData) mymedia).getTypeMedia(), this);
            playscree.setVisibleVideo();
            PlayVideoController controller = new PlayVideoController(midlet,
                ↪ getAlbumData(), (AlbumListScreen) getAlbumListScreen(), playscree);
            controller.setMediaName(selectedMediaName);
            this.setNextController(controller);
        }
        return true;
    } catch (ImageNotFoundException e) {
        Alert alert = new Alert("Error", "The selected item was not found in the
            ↪ mobile device", null, AlertType.ERROR);
        Display.getDisplay(midlet).setCurrent(alert, Display.getDisplay(midlet).
            ↪ getCurrent());
        return false;
    } catch (PersistenceMechanismException e) {
        Alert alert = new Alert("Error", "The mobile database can open this item 1",
            ↪ null, AlertType.ERROR);
        Display.getDisplay(midlet).setCurrent(alert, Display.getDisplay(midlet).
            ↪ getCurrent());
        return false;
    }
}

```

Figure B.6. Question 6.

```

public IteratorDsk getSpecialityList() throws RepositoryException ,
↳ ObjectNotFoundException {
    List listaEsp = new ArrayList();
    String sql = "SELECT * FROM SCBS_especialidade";
    ResultSet rs = null;
    try {
        Statement stmt = (Statement) this.mp.getCommunicationChannel();
        rs = stmt.executeQuery(sql);
        if (!rs.next()) {
            throw new ObjectNotFoundException("");
        }
        do {
            MedicalSpeciality esp = search((new Integer(rs.getString("codigo"))).
↳ intValue());
            listaEsp.add(esp);
        } while (rs.next());
        rs.close();
        stmt.close();
    } catch (PersistenceMechanismException e) {
        throw new RepositoryException(ExceptionMessages.EXC_FALHA_PROCURA);
    } catch (SQLException e) {
        throw new RepositoryException(ExceptionMessages.EXC_FALHA_PROCURA);
    } finally {
        try {
            mp.releaseCommunicationChannel();
        } catch (PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
    }
    return new ConcreteIterator(listaEsp);
}

```

```

public IteratorDsk getDiseaseTypeList() throws RepositoryException ,
↳ ObjectNotFoundException {
    List listatd = new ArrayList();
    String sql = "SELECT * FROM SCBS_tipodoenca";
    ResultSet rs = null;
    try {
        Statement stmt = (Statement) this.mp.getCommunicationChannel();
        rs = stmt.executeQuery(sql);
        if (!rs.next()) {
            throw new ObjectNotFoundException(ExceptionMessages.EXC_FALHA_PROCURA);
        }
        do {
            DiseaseType td = partialSearch((new Integer(rs.getString("codigo"))).
↳ intValue());
            listatd.add(td);
        } while (rs.next());
        rs.close();
        stmt.close();
    } catch (PersistenceMechanismException e) {
        e.printStackTrace();
        throw new RepositoryException(ExceptionMessages.EXC_FALHA_BD);
    } catch (SQLException e) {
        System.out.println(sql);
        e.printStackTrace();
        throw new RepositoryException(ExceptionMessages.EXC_FALHA_BD);
    }
    return new ConcreteIterator(listatd);
}

```

Figure B.7. Question 7.

```

public void updateComplaint(Complaint complaint) throws TransactionException,
↪ RepositoryException, ObjectNotFoundException, ObjectNotValidException {
    try {
        getPm().beginTransaction();
        complaintRecord.update(complaint);
        getPm().commitTransaction();
    } catch (RepositoryException e) {
        getPm().rollbackTransaction();
        throw e;
    } catch (ObjectNotFoundException e) {
        getPm().rollbackTransaction();
        throw e;
    } catch (TransactionException e) {
        getPm().rollbackTransaction();
        throw e;
    } catch (Exception e) {
        getPm().rollbackTransaction();
    }
}

```

```

public void update(Employee employee) throws TransactionException, RepositoryException
↪ , ObjectNotFoundException, ObjectNotValidException {
    try {
        getPm().beginTransaction();
        employeeRecord.update(employee);
        getPm().commitTransaction();
    } catch (TransactionException e) {
        getPm().rollbackTransaction();
        throw e;
    } catch (ObjectNotValidException e) {
        getPm().rollbackTransaction();
        throw e;
    } catch (ObjectNotFoundException e) {
        getPm().rollbackTransaction();
        throw e;
    } catch (Exception e) {
        getPm().rollbackTransaction();
    }
}

```

Figure B.8. Question 8.



```

public static Schedule stringToHorario(String horarioStr, int formato) throws
↳ InvalidDateException {
    String segundoStr = null, minutoStr = null, horaStr = null;
    Schedule horario = null;
    try {
        switch(formato) {
            case (Schedule.FORMATO1):
                horaStr = horarioStr.substring(0, 2);
                minutoStr = horarioStr.substring(3, 5);
                segundoStr = horarioStr.substring(6, 8);
                break;
            case (Schedule.FORMATO2):
                segundoStr = horarioStr.substring(0, 2);
                minutoStr = horarioStr.substring(2, 4);
                horaStr = horarioStr.substring(4, 6);
                break;
            default:
                horario = null;
                break;
        }
        horario = new Schedule(segundoStr, minutoStr, horaStr);
    } catch (Exception nb) {
        throw new InvalidDateException(horarioStr);
    }
    return horario;
}

```

```

public static Date stringToData(String dataStr, int formato) throws
↳ InvalidDateException {
    String diaStr, mesStr, anoStr;
    String minutoStr, segundoStr, horaStr;
    Date data = null;
    try {
        switch(formato) {
            case (FORMATO1):
                diaStr = dataStr.substring(0, 2);
                mesStr = dataStr.substring(3, 5);
                anoStr = dataStr.substring(6, 10);
                data = new Date(diaStr, mesStr, anoStr);
                break;
            case (FORMATO2):
                diaStr = dataStr.substring(0, 2);
                mesStr = dataStr.substring(3, 5);
                anoStr = dataStr.substring(6, 10);
                horaStr = dataStr.substring(11, 13);
                minutoStr = dataStr.substring(14, 16);
                segundoStr = dataStr.substring(17, 19);
                data = new Date(segundoStr, minutoStr, horaStr, diaStr, mesStr, anoStr
↳ );
                break;
            case (FORMATO3):
                diaStr = dataStr.substring(0, 2);
                mesStr = dataStr.substring(2, 4);
                anoStr = dataStr.substring(4, 8);
                break;
            default:
                data = null;
                break;
        }
    } catch (Exception nb) {
        throw new InvalidDateException(dataStr);
    }
    return data;
}

```

Figure B.9. Question 9.

```

public Page getItem(String productID, int start, int count, Locale locale) throws
    ↪ CatalogDAOSysException {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        connection = getDataSource().getConnection();
        String [] parameterValues = new String [] { locale.toString(), productID };
        if (TRACE) {
            printSQLStatement(sqlStatements, XML_GET_ITEMS, parameterValues);
        }
        statement = buildSQLStatement(connection, sqlStatements, XML_GET_ITEMS,
            ↪ parameterValues);
        resultSet = statement.executeQuery();
        if (start >= 0 && resultSet.absolute(start + 1)) {
            boolean hasNext = false;
            List items = new ArrayList();
            do {
                int i = 1;
                items.add(new Item(productID, resultSet.getString(i++).trim(),
                    ↪ resultSet.getString(i++), resultSet.getString(i++).trim(),
                    ↪ resultSet.getString(i++).trim(), resultSet.getString(i++),
                    ↪ resultSet.getString(i++), resultSet.getString(i++), resultSet.
                    ↪ getString(i++), resultSet.getString(i++), resultSet.getString(i
                    ↪ ++), resultSet.getDouble(i++), resultSet.getDouble(i++)));
            } while ((hasNext = resultSet.next()) && --count > 0);
            return new Page(items, start, hasNext);
        }
        return Page.EMPTY_PAGE;
    } catch (SQLException exception) {
        throw new CatalogDAOSysException("SQLException: " + exception.getMessage());
    } finally {
        closeAll(connection, statement, resultSet);
    }
}

```

```

public Item getItem(String itemID, Locale l) throws CatalogDAOSysException {
    Connection c = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    Item ret = null;
    try {
        c = getDataSource().getConnection();
        ps = c.prepareStatement(GET_ITEM_STATEMENT, ResultSet.TYPE_SCROLL_INSENSITIVE,
            ↪ ResultSet.CONCUR_READ_ONLY);
        ps.setString(1, l.toString());
        ps.setString(2, itemID);
        rs = ps.executeQuery();
        if (rs.first()) {
            int i = 1;
            ret = new Item(rs.getString(i++).trim(), rs.getString(i++).trim(), rs.
                ↪ getString(i++), itemID, rs.getString(i++).trim(), rs.getString(i++)
                ↪ , rs.getString(i++), rs.getString(i++), rs.getString(i++), rs.
                ↪ getString(i++), rs.getString(i++), rs.getDouble(i++), rs.getDouble(
                ↪ i++));
        }
        rs.close();
        ps.close();
        c.close();
        return ret;
    } catch (SQLException se) {
        throw new CatalogDAOSysException("SQLException: " + se.getMessage());
    }
}

```

Figure B.10. Question 10.

```

private String getSubTagAttribute(Element root, String tagName, String subTagName,
    ↪ String attribute) {
    String returnString = "";
    NodeList list = root.getElementsByTagName(tagName);
    for (int loop = 0; loop < list.getLength(); loop++) {
        Node node = list.item(loop);
        if (node != null) {
            NodeList children = node.getChildNodes();
            for (int innerLoop = 0; innerLoop < children.getLength(); innerLoop++) {
                Node child = children.item(innerLoop);
                if ((child != null) (child.getNodeName() != null) child.getNodeName()
                    ↪ ().equals(subTagName)) {
                    if (child instanceof Element) {
                        return ((Element) child).getAttribute(attribute);
                    }
                }
            }
        }
    }
    return returnString;
}

```

```

public static String getSubTagValue(Element root, String tagName, String subTagName) {
    String returnString = "";
    NodeList list = root.getElementsByTagName(tagName);
    for (int loop = 0; loop < list.getLength(); loop++) {
        Node node = list.item(loop);
        if (node != null) {
            NodeList children = node.getChildNodes();
            for (int innerLoop = 0; innerLoop < children.getLength(); innerLoop++) {
                Node child = children.item(innerLoop);
                if ((child != null) (child.getNodeName() != null) child.getNodeName()
                    ↪ ().equals(subTagName)) {
                    Node grandChild = child.getFirstChild();
                    if (grandChild.getNodeValue() != null) return grandChild.
                        ↪ getNodeValue();
                }
            }
        }
    }
    return returnString;
}

```

Figure B.11. Question 11.

```

public int delete(String sql) throws SQLException {
    PreparedStatement prepared = null;
    int count = 0;
    try {
        prepared = connection.prepareStatement(sql);
        preparedStatementList.add(prepared);
        prepared.execute();
        count = prepared.getUpdateCount();
    } catch (SQLException e) {
        System.out.println(" - Error during execution of the following SQL Statement
        ↪ -> [" + sql + "] " + e.toString());
        throw e;
    }
    return count;
}

```

```

public void update(String sql) throws SQLException {
    try {
        PreparedStatement prepared = connection.prepareStatement(sql);
        preparedStatementList.add(prepared);
        prepared.execute();
        int rowsUpdated = prepared.getUpdateCount();
        if (rowsUpdated == 0) {
            System.out.println(" - The record to be updated does not exists in
            ↪ database");
            throw new SQLException();
        }
    } catch (SQLException e) {
        System.out.println(" - Error during execution of the following SQL Statement
        ↪ -> [" + sql + "] " + e.toString());
        throw e;
    }
}

```

Figure B.12. Question 12.