Automated Detection of Code Smells Caused by Null Checking Conditions in Java Programs

Kriangchai Sirikul and Chitsutha Soomlek
Department of Computer Science, Faculty of Science, Khon Kaen University
Naimuang, Muang, Khon Kaen, 40002 Thailand
kriangchais@kkumail.com; chisutha@kku.ac.th

***Abstract*—A null object in Java is occurred when an object is not initialized properly, but a property or a method in the object is called; resulting in the null pointer exception problem and low software quality. To solve this problem, developers usually create a number of null checking conditions in their codes. Duplicated null checking conditions affect the code quality. The duplicated code makes a computer program repeatedly and exhaustively checks for null value, therefore, it reduces the performance of the program. This research involves automatically detecting the code smells caused by null checking conditions in Java by using regular expression. The research specifically focuses on detecting (i) introduce null object, (ii) duplicated code, and (iii) null checking in a string comparison problem. Thirteen Java open-source projects were employed to verify and validate our approach. The detection results were significantly improved when using regular expression comparing to using Abstract Syntax Tree (AST) for detecting code smells caused by null checking conditions in Java programs.**

Keywords—code smell; null object detection; null detection

I. INTRODUCTION

A code smell, a.k.a. bad smell, is an indicator of a deeper problem within source code [1]. A code smell indicates that there is a problem caused by poor software design, poor programming practices, or inexperienced programmers [1, 2]. A code smell creates negative impact on the software quality and also maintainability of a software system. Code smells could lead to software failure in the future [2]. There are various types of code smells such as duplicated code, data clumping, feature envy, public field, etc. [1, 2]

Currently, there are a number of tools that could assist a programmer to detect certain types of code smells so that the problem could be easily detected, and then, could be solved automatically, manually, or both before the software is released. Checkstyle [3, 6], FindBugs [4], inCode [5], JDeodorant [6, 7], PMD [6, 8] are examples of the tools.

Java programmers frequently encounter a null pointer exception error [9]; which usually happens when an object is not properly initialized and when a return variable is null [10, 11]. To avoid this particular problem, Java programmers usually create null checking conditions in their code to handle all possible situations that an object would have null value at runtime. The resulting software system, therefore, contains a number of duplicated code checking on the same object repeatedly from different locations of the code.

978-1-5090-2033-1/16/$31.00 ©2016 IEEE

This research employed regular expressions (RegEx) to automatically detect code smells and problems caused by null checking conditions in Java programs. Currently, the research focuses on two types of code smells, i.e., duplicated code [1] and introduce null object [1], and null checking in a string comparison problem within the context of null checking conditions in Java.

This paper is organized as follows. Section II presents recent and related work. Section III describes the proposed solutions to detect code smells caused by null checking conditions in Java programs through regular expressions. Experimental results are discussed in section IV. Section V, then, outlines the conclusions of this paper and future work.

II. RELATED WORK

A. Regular Expression (RegEx)

RegEx is a standard textual syntax or a pattern used for finding a sequence of characters in strings [12]. It is a string pattern matching. RegEx is supported by a vast number of programming languages and widely used in many research areas. In 2006, H. Larkin introduced the variables and reversibility concepts to enhance the ability of RegEx so that it can extract and create text representations of data [13]. Object-oriented data can be imported and exported into other presentation formats by using a single group of RegEx statements [13].

RegEx was also adopted in code generators to reduce their complexity [14]. The existing source code was transform in order to implement the required functionalities by using regular expression substitution instead of using the traditional template-based approach [14]. The results showed steeper learning curve and enhanced maintainability [14]. Moreover, RegEx was employed to identify duplicated, non-context-free, and complex structures [15].

B. Code Smells and Problems Relative to Null Checking Conditions

Various types of code smells and poor programming problems had been described in [1, 2, 16]. Some of them are the results of null checking conditions in a software system. This research specifically focuses on the following code smells and problem:

978-1-5090-2033-1/16/$31.00 ©2016 IEEE

1) Duplicated code: The same code structure that appears more than one place in a program, the same expression is placed in two methods of the same class, or the same expression are located in two sibling subclasses [1]. Extract method [1] and substitute algorithm [1] are commonly used to solve the problem. In this research, the the same structure of null checking statement, that is repeatedly appeared, is considered as duplicated code.

2) Introduce null object: Repeated null checking conditions are added into the code to prevent the null pointer exception problem [1]. By doing so, the duplications of null checking conditions could have been placed in different locations of the software system. Recently, Gaitani et al. proposed a novel method to discover and eliminate null checking conditions through refactoring to the Null Object design patterns [17].

3) Null checking in a string comparison problem: Null checking conditions are usually found in string comparison, particularly in an if statement, as shown in Fig.1-Case A. This form of defensive programming can be employed to prevent the null pointer exception error. The same null checking statement is repeatedly appeared when the same String object is compared, resulting in a marvellous number of duplicated null checking conditions. The null checking conditions could be eliminated by calling the SomeString.*equals(value)* method instead [16]. Therefore, if *value* is null, there will be no null pointer exception error since it does not point to the String object, i.e. SomeString.

Case a
input : String str = null

```
Line 01  : if str != null and str.equals("word") then
Line 02  :    operate something ...
Line 03  : }
```

Case b
input : String str = null

```
Line 01  :if "word".equals(str) then
Line 02  :    operate something ...
Line 03  : }
```

Fig.1. Using the return value of a method as a condition in an if statement to replace a null checking condition.

III. METHODOLOGY

In this research, RegEx is employed to automatically discover and classify various patterns of null checking conditions in Java programs. The process of automated detection of null checking conditions is illustrated in Fig.2. The process starts from inputting a Java project. There are three major activities in the process: detect null checking conditions and extract variables, extract data types, and classify the types of code smells and the problem.

A. Detecting Null Checking Conditions and Extracting Variables

In this stage, null checking conditions in the inputted Java project are discovered and the variables used for null checking in the relevant statements are extracted. The steps are as follows:

1) Open Java project and list all subdirectories
2) Search for Java files within the each subdirectory
3) Open each Java file and count the line of code (LOC)
4) Go through each statement to search for if statements and null checking conditions, and extract the variables within those conditions by using RegEx in Table I
5) Write the extracted information to Comma Separated Value (.csv) files

TABLE I. REGULAR EXPRESSIONS CREATED FOR DETECTING NULL CHECKING CONDITIONS AND EXTRACTING RELEVANT VARIABLES

Cases	Patterns
Detect if conditions	"^(if|(}\\s*if)|(else\\s+if)|(}\\s*(else\\s+if)))\\s*\\("
Detect comment lines and space lines	1. ^(\\s*)$\|^(//)(.*)\|^(/\\*)(.*)\|(\\*/)$|^(/*)(.*)(*/)$|^(*)(.*) 2. (.*)(//)(.*) 3. (.*)(/*)(.*)
Detect null checking conditions	(?=.*\\b(if)\\s*\\()(?=.*\\b(null)\\b)
Extract variable in this.*variable*	(.*this)(\\.)(.*)
Extract variable in this.*variable*.method()	(.*this)(\\.)(.*)(\\.)(.*)(\\()(.*)
Extract variable in this.*variable*)	(.*this)(\\.)(.*)(\\))(.*)
Extract variable in if(*variable*	(.*if\\s*)(\\(\\s*)+(\\w+)(.*)
Extract variable in if(*variable*[*]	(.*if\\s*)(\\(\\s*)+(\\w+\\[.*\\])(.*)
Extract variable in *variable*)	(.*\\w+)(\\s*)(\\))(.*)
Extract variable in *variable*[*])	(.*\\w+\\s*\\[.*\\])(\\s*)(\\))(.*)

There are cases that we did not include in this research:

1) A null checking condition containing a method that compares to null: An example of the condition is *if (getModel() != null) {*.

2) A null checking condition that assigns a value to the variable within the statement while comparing to null value: For example, *if ((frcc = fileContainers.get(alias)) == null) {*.

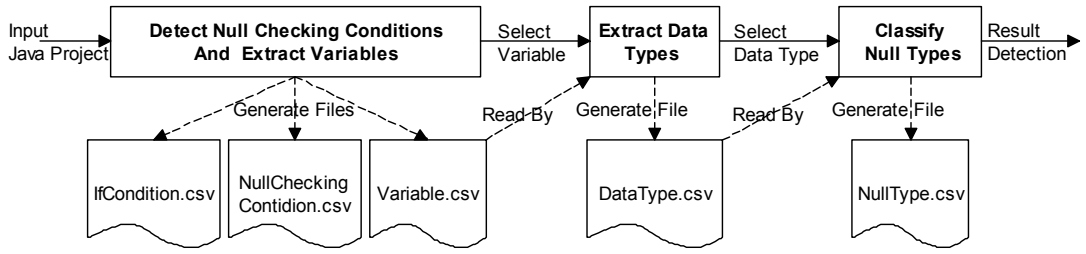

Fig.2. The overall process of automated null-checking-condition detection

```
input : java file
Line 01  : for loop line to LOC in java file do
Line 02  :    if isCommentLineAndSpaceLine(line) then
Line 03  :       continue;
Line 04  :    end if
Line 05  :    if isNullCommentAfterOperateStatement(line) then
Line 06  :       line = remove comment null content.
Line 07  :    end if
Line 08  :    if isNullChecking(line) then
Line 09  :       if ! isExceptPattern(line) then
Line 10  :            extPattern = verifyPattern(line)
Line 11  :            variable = extractVariable(line, extPattern)
Line 12  :       end if
Line 13  :    end if
Line 14  : end for loop
```

Fig.3. Psedocode created for detecting null checking conditions and extracting relevant variables

3) A null checking condition containing ternary if-else operator: An example of the statement is *if (this.period != null ? !this.period.equals(timePeriodValue.period)*.

4) A null checking condition that compares the result from a called method to null value: For example, *if (Mark.GENERATE_MARK. checkmark (page) == null) {*.

5) A null checking condition containing a cast variable: For example, *if (((Hyphen)item).noBreak != null) {*.

Those five cases are exceptions which requires further manual operations.

Fig. 3 gives more details of step 4). In each Java file, if "*null*" is presented in a comment line, the comment line is removed, in order to avoid misdetection. Then, a statement containing a null checking condition is identified. When the statement is found and it is not match to any of the excepted cases, the RegEx presented in Table I are employed to extract the variables in the statement.

B. Extracting Data Types

The data types of the extracted variables are identified in this stage. There are four groups of data types: (i) Internal class, (ii) Primitive type [10, 11], (iii) Non-primitive type [10, 11], and (iv) External class. Internal classes are originated in a Java project. External classes are imported classes and inherited classes. The steps are as follows:

1) Open Variable.csv file

2) Read a variable name and its java path file

3) Open each Java file and count the line of code (LOC)

4) Search for the variable of interest by following the algorithm shown in Fig.4: The algorithm is developed from the concept of "*this*" keyword in Java [10, 11]; which could be used to decide whether a variable is from a local class. If the variable is from a local class, then the searching process starts from the class header to the line containing the variable in null checking condition. On the other hand, the searching process starts from the line containing the variable in null checking condition to the method header instead. In case that the variable is not found, the searching process starts from the last line of the file to the class header. If the variable is not located

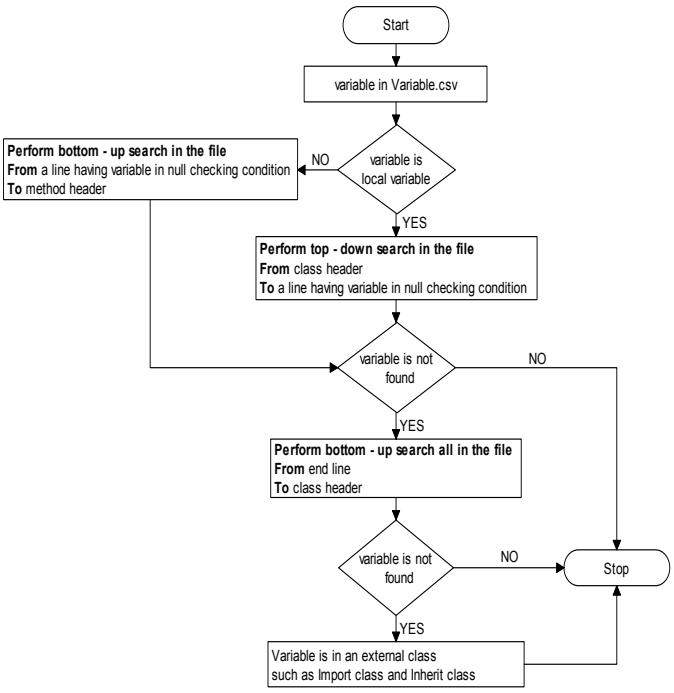

Fig.4. The algorithm used for searching a variable in a Java file

in the Java file, there is a possiblity that it is placed in an external class. The output of this step is the line of a statement that contains the variable of interest.

5) Extract the data type by using the RegEx in Table II and pseudo code presented in Fig. 5: Note that **DT** in Table II is referred to data type. Comment and empty lines are identified to eliminate non-relevant lines of code, before identifying the data type of the variable regarding the matched pattern.

6) Write the extracted data type of the variable to a CSV file

TABLE II. REGULAR EXPRESSIONS CREATED FOR EXTRACTING DATA TYPES

Extract data type in	Patterns
(final or private) **DT** *variable*	(final|private)(\\s+)(.*)(\\s+)("+ *variable* +")
*(**DT** <*> *variable*	(\\s+\\w+\\s*\\(\\s*)(\\w+\\s*\\<.*\\>)(\\s+) ("+ *variable* +")
* **DT** <*> *variable*	(.*\\s+)(\\w+\\s*\\<.*\\>)(\\s+)("+ *variable* +")
DT <*> *variable*	(\\w+\\s*\\<.*\\>)(\\s+)("+ *variable* +")
*, final **DT** *variable*	(.*)(\\,)(\\s*)(final)(\\s+)(.*)(\\s+)("+ *variable* +")
* (*, *, **DT** *variable*	(.*\\s+.*\\s*\\(\\s*.*\\s+.*\\,\\s*.*\\s+.*\\,\\s*)(.*) ("+*variable*+")
*, **DT** *variable*	(.*)(\\,)(\\s*)(.*)(\\s+)("+ *variable* +")
*(final **DT** *variable*	(.*)(\\()(\\s*)(final)(\\s+)(.*)(\\s+)("+ *variable* +")
* **DT** = *, *variable*	(.*)(\\s+)(\\w+\\s*\\=\\s*.*)(,\\s*)("+ *variable* +")
*(**DT** *variable*	(.*)(\\()(.*)(\\s+)("+ *variable* +")
DT *variable*	(\\w+)(\\s+)("+ *variable* +")
DT [*] *variable*	(\\w+\\s*\\[.*\\])(\\s+)("+ *variable* +")
* final **DT** *variable*	(.*\\s+)(final)(\\s+)(.*)(\\s+)("+ *variable* +")

```
input : java file
Line 01 : for loop line to LOC in java file do
Line 02 :   if isCommentLineAndSpaceLine(line) then
Line 03 :     continue;
Line 04 :   end if
Line 05 :   if isVerifyContentInLineForExtractDataType(line) then
Line 06 :     extPattern = verifyPattern(line)
Line 07 :     dataType = extractDataType(line, extPattern)
Line 08 :   end if
Line 09 : end for loop
```

Fig.5. Psedocode created for identifying data type

```
input : DataType.csv
Line 01 : for loop variableInfo to last of line in DataType.csv do
Line 02 :     if isExternalClass(variableInfo) then
Line 03 :         variable is Duplicate Code
Line 04 :     else if isStringComparison(variableInfo) then
Line 05 :         variable is String Comparison
Line 06 :     else if isVerifyFilePath(variableInfo) then
Line 07 :         variable is Duplicate Code
Line 08 :     else if isIntroduceNullObject(variableInfo) then
Line 09 :         variable is Introduce Null Object
Line 10 :     else
Line 11 :         variable is Duplicate Code
Line 12 :     end if
Line 13 : end for loop
```

Fig.6. Classification process

```
input : condition
Line 01 : if isNullAndEquals(condition) then
Line 02 :   if isVariableCompareNull(condition) then
Line 03 :     if isVariableDotEquals(condition) then
Line 04 :       condition is String Comparison
Line 05 :     end if
Line 06 :   end if
Line 07 : end if
```

Fig.7. Psedocode created for detecting null checking conditions in string comparison problem

C. Classification

In this stage, the null checking conditions are classified into three groups, i.e., duplicated code, introduce null object, and null checking in a string comparison. The structure of null checking conditions, the data type of an object, and RegEx are employed in the classification process. The major steps in this stage are as follows:

1) Open DataType.csv file

2) Search for the extracted information that could be used to classify the null checking conditions

3) Classify by using the conditions in Fig. 6

4) Write the classification results to a CSV file

In order to detect null checking in a string comparison, the RegEx in Table III and the algorithm in Fig. 7 are employed.

```
input : java file
Line 01 : for loop line to LOC in java file do
Line 02 :   if isCommentLineAndSpaceLine(line) then
Line 03 :     continue;
Line 04 :   end if
Line 05 :   if isPublic(line) and isBracket(line) then
Line 06 :     if ! isParameter(line) and ! isCommaSymbol(line) and ! isEqualSymbol(line) then
Line 07 :       if ! isReturnMultiValue(line) then
Line 08 :         if isReturnOneValue(line) then
Line 09 :           Class is Introduce Null Object
Line 10 :           methodName = extMethodName(line) ;
Line 11 :           Keep class information for filter process
Line 12 :         end if
Line 13 :       end if
Line 14 :     end if
Line 15 :   end if
Line 16 : end for loop
```

Fig.8. Psedocode created for detecting candidate classes

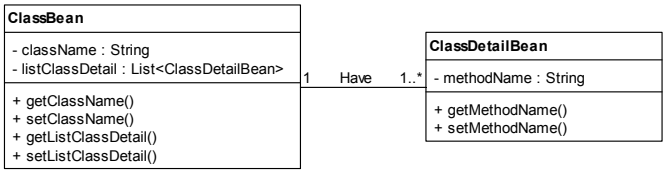

Fig.9. The structure of ClassBean and ClassDetailBean that are created for recording the extracted classes and their methods

```
input : line condition and java file of line condition
Line 01 : if ! IsVariableDotMethod(lineCondition) then // detect if variable == null
Line 02 :   for loop line to LOC in java file of line condition do
Line 03 :     if isCommentLineAndSpaceLine(line) then
Line 04 :       continue;
Line 05 :     end if
Line 06 :     if isVerifyBraces(line) then
Line 07 :       break;
Line 08 :     end if
Line 09 :     if isVerifyMethod(line) then
Line 10 :       for loop classBean to listClassPotentiallyIntroduceNullObject do
Line 11 :         if isVerifyClass(classBean.getClassName()) then
Line 12 :           for loop classDetailBean to class.getListClassDetailBean do
Line 13 :             if isVerifyMethodName(classDetailBean.getMethodName()) then
Line 14 :               lineCondition is Introduce Null Object
Line 15 :             end if
Line 16 :           end for loop
Line 17 :         end if
Line 18 :       end for loop
Line 19 :     end if
Line 20 :   end for loop
Line 21 : else // detect if variable.method == null
Line 22 :   Same as Line 10-18
Line 23 : end if
```

Fig.10. Psedocode created for detecting the introduce null object problem

If "*null*" and "*equals*" are found in the examining statement, then the relevant variable will be checked. If the variable is compared to "*null*" and "*.equals()*" is employed, then the statement is a null checking in a string comparison .

TABLE III. REGULAR EXPRESSIONS CREATED FOR DETECTING NULL CHECKING IN STRING COMPARISON PROBLEM

Detect	Patterns
Null and equals	(?=.*\\s*(null)\\s*)(?=.*\\.(equals)\\s*\\()
Variable checking null	("+variable+")(\\s*)([!=])(=)(\\s*)(null)
Variable dot equals	("+variable+")(\\s*)(\\.equals\\()

The RegEx in Table IV and the algorithm in Fig. 8 were created to identify a variable in a null checking condition located in a class that are likely to have the introduce null object problem. Comments and empty lines are removed. If the statement is a public method with no parameter and the method returns single value, then the class has a high possibility to contain the introduce null object problem. After that, the names of methods within the class are extracted. The class and their methods will be used for identifying the introduce null object problem.

TABLE IV. REGULAR EXPRESSIONS CREATED FOR DETECTING INTRO DUCE NULL OBJECT

Cases	Patterns
Detect Public	\\b?(public)\\s++
Detect bracket	(?=.*\\()"(?=.*\\))
Detect parameter	\\(\\w+.*\\)
Detect comma	","
Detect equal	"="
Detect single return value	(\\bString\\b)|(\\bBoolean\\b)|(\\bboolean\\b)|(\\bInteger\\b)|(\\bint\\b)|(\\bDate\\b)|(\\bNumber\\b)|(\\bDouble\\b)|(\\bdouble\\b)|(\\bCharacter\\b)|(\\bchar\\b)|(\\bBigInteger\\b)|(\\bBigDecimal\\b)|(\\bFloat\\b)|(\\bfloat\\b)|(\\bbyte\\b)|(\\bshort\\b)|(\\bLong\\b)|(\\blong\\b)
Detect multiple return values	(\\bList\\b)|(\\bArrayList\\b)|(\\bLinkedList\\b)|(\\bMap\\b)|(\\bHashMap\\b)|(\\bHashTable\\b)|(\\bTreeMap\\b)|(\\bConcurrentHashMap\\b)|(\\bSet\\b)|(\\bEnumSet\\b)|(\\bHashSet\\b)|(\\bLinkedHashSet\\b)|(\\bTreeSet\\b)|(\\bCollection\\b)|(\\bIterator\\b)|(\\bEnumeration\\b)| (\\benum\\b)|\\[\\]
Extract method name	(.*)(\\s)(\\w+)(\\s*)(\\(\\))(.*)

TABLE V. REGULAR EXPRESSIONS CREATED FOR FILTERING INTRODUCE NULL OBJECT

Detect	Patterns
Variable dot	("+variable+")(\\s*)(\\.)
Open braces	\\{
Close braces	\\}
Bracklet	\\(\\)
Method name	("+methodName+")

Fig.10 explains the algorithm used for identifying the introduce null object problem by using the RegEx in Table V. The algorithm starts with identifying an if condition containing "*Variable==null*" (i.e., Line01-20 in Fig.9). After that, an if statement containing "*Variable.Method==null*" is detected (i.e., Line21-23 in Fig.9). An if statement that matched to any of the two cases indicates that there is an introduce null object problem in the code.

At the end the classification process, the source code containing duplicated null checking conditions, introduce null objects, null checking in a string comparison problems in a Java project are discovered and recorded in NullType.csv. Developers can use the results of the automated code smells detection to refactor their code and therefore improve the quality and maintainability of the software.

IV. EXPERIMENTAL RESULTS

Thirteen open-source Java projects were employed to verify and validate our approach. Table VI presents the results obtained from the first and second stages of automated null-checking-condition detection. Java files, source lines of code (SLOC), if conditions, null checking conditions, and variables within null checking conditions were extracted from each Java project. The results in the Table VI shows the number of each parameter extracted from the Java projects and average time spent on extracting the data. Note that the average time is calculated from 10 rounds of testing.

Since null checking conditions $\subset$ if conditions, the number of null checking conditions $\leq$ if conditions. One null checking condition may contain more than one variable, therefore, the number of extracted variables $\geq$ the number of null checking conditions. The average time spent on detecting null checking condition and extracting variables is increased when SLOC is increased.

TABLE VI. RESULTS OBTAINED FROM DETECTING NULL CHECKING CONDITIONS AND EXTRACTING VARIABLES

Java Project	Java files	SLOC	If condition	No. of null checking condition	Variables	Average time (s.)
Xml commons ext. 1.4.01	478	12908	563	284	290	0.32700
Violet 2.1.0	385	33446	1227	410	424	0.48700
Nutch 1.1	447	45414	3030	839	856	0.76800
Myfaces 2.2.10	1847	211807	12185	5683	5824	2.53500
Jackrabbit 2.12.1	3124	340782	19483	5216	5363	4.12500
Jmeter 2.9	1013	106604	5450	1472	1505	1.38000
Apache ant 1.9.7	1217	137703	9251	3487	3635	1.75600
Jade 4.4.0	1007	134829	7952	2242	2258	1.43300
Xerces 2.11.0	757	126837	12372	3867	3940	1.44000
Jfreechart 1.0.14	1005	146854	9442	2703	2753	1.65600
Xalan 2.7.2	962	175623	10136	3439	3513	2.26400
Batik 1.7	1664	215176	12550	4072	4162	2.86400
Fop 1.1	1724	179862	11867	3540	3624	2.79100

Table VII presents the results obtained from the process of data type extraction. At this stage, four data types, i.e., internal

class, primitive type, non-primitive type, and external class were extracted in order to be used for classifying the null checking conditions. The results indicated that most of the variables used in all extracted null checking conditions are from the internal classes. Primitive-type variables are rarely used in null checking conditions. The average time spent on extracting data types is increased when the number of variables in null checking conditions is increased. Note that the average time is calculated from 10 rounds of testing.

TABLE VII. RESULTS OBTAINED FROM EXTRACTING DATA TYPES

Java Project	Internal class	Primitive type	Non-primitive type	External class	Average time (s.)
Xml commons ext. 1.4.01	86	0	191	13	0.26100
Violet 2.1.0	182	0	242	0	0.33300
Nutch 1.1	279	2	568	7	0.63700
Myfaces 2.2.10	2013	1	3766	44	5.14700
Jackrabbit 2.12.1	2755	0	2576	32	5.73000
Jmeter 2.9	496	0	1004	5	1.19600
Apache ant 1.9.7	1446	0	2121	68	2.62100
Jade 4.4.0	1255	0	975	28	1.92900
Xerces 2.11.0	2287	6	1433	214	5.19200
Jfreechart 1.0.14	1488	2	1258	5	2.15900
Xalan 2.7.2	2834	17	440	222	4.78000
Batik 1.7	2608	4	1476	74	3.62700
Fop 1.1	2724	7	813	80	3.39800

Table VIII shows the classification results. A candidate variable is resided in a null checking condition within a class that are likely to contain the introduce null object problem (See Fig.8). At the end of the classification process, the results indicated that 96% of the problems found in the thirteen Java projects are duplicated code, 3.38% are introduce null object, and 0.62% are null checking in a string comparison.

Moreover, the algorithm created for detecting introduce null object is more complex than the algorithm created for detecting the other two problems. As a result, the time spent on detecting introduce null object is longer. A Java project containing a larger number of introduce null object will take longer time to process. This can be confirmed by the results presented in Table VIII. Jackrabbit 2.12.1 has the highest number of introduce null object and the classification process spent 1.02800 seconds to finish. In case of Xml commons ext. 1.4.01, the project has to lowest number of introduce null object, therefore, it took only 0.06100 seconds in the classification process.

TABLE VIII. CLASSIFICATION RESULTS

Java Project	Candidate variables (Introduce Null Object)	Introduce Null Object	Duplicated Code	Null checking in a string comparison	Average time (s.)
Xml commons ext. 1.4.01	19	5	285	0	0.06100
Violet 2.1.0	61	4	419	1	0.09600
Nutch 1.1	201	50	792	14	0.11500
Myfaces 2.2.10	1558	82	5708	34	0.80000
Jackrabbit 2.12.1	1708	182	5155	26	1.02800
Jmeter 2.9	348	69	1431	5	0.23500
Apache ant 1.9.7	802	105	3498	32	0.45300
Jade 4.4.0	735	182	2061	15	0.50500
Xerces 2.11.0	1091	157	3748	35	0.88400
Jfreechart 1.0.14	939	97	2653	3	0.66500
Xalan 2.7.2	1227	85	3392	36	0.82100
Batik 1.7	839	115	4026	21	0.71900
Fop 1.1	1161	155	3453	16	0.77100

TABLE IX. COMPARISON RESULTS

Java Project	Candidate variables detected by using AST [17]	Candidate variables detected by using RegEx	Introduce null object detected by using RegEx
Nutch 1.1	10	201	50
Jmeter 2.9	42	348	69
Xerces 2.11.0	172	1091	157
Jfreechart 1.0.14	64	939	97
Fop 1.1	207	1161	155

In order to validate our approach, the detection results were compared to the results obtained from the detection phase in [17]. Gaitani et al. employed Abstract Syntax Tree (AST) to detect introduce null object in null checking conditions before performing code elimination process [17]. Table IX confirms that the detection results were significantly improved when using RegEx for detecting the introduce null object problem. By using RegEx, we detected more candidate variables (i.e., variables that could lead us to classes that are likely to contain the introduce null object problem) than using AST by 90.52% in Nutch 1.1, 78.46 % in Jmeter 2.9, 72.76 % in Xerces 2.11.0, 87.23 % in Jfreechart 1.0.14, and 69.73 % in Fop 1.1. When comparing our final results to the candidate variables detected by [17], RegEx still produced better results by 18.96 % in Nutch 1.1, 6.92 % in Jmeter 2.9, and 3.30% in Jfreechart 1.0.14. In case of Xerces 2.11.0 and Fop 1.1, our detection results were lower by 1.18% and 3.30%, respectively.

V. Conclusions and Future Work

This paper presents an automated approach to detect two types code smells and a problem caused by null checking conditions in a Java program. Duplicated code, introduce null object, and null checking in a string comparison problem can be effectively discovered by using regular expressions together with the proposed algorithms.

Thirteen open-source Java projects were employed in our experiments to verify and validate our approach. The number of the revealed problems in the Java projects were different due to the fact that the projects were developed by different programmers who have different coding styles. The experimental results also confirmed that our approach can detect more classes containing the introduce null object problem than using AST in most cases.

In conclusion, the proposed approach can help Java developers to identify deeper problems in Java programs. It can be used in conjunction with code refactoring to eliminate the detected problems. The results obtained from the automated code smells detection can be utilized in both automated and manual code refactoring in order to enhance the code quality and maintainability of software. The code refactoring process is left for future work.

Acknowledgment

This research was partially supported by the Department of Computer Science, Faculty of Science, Khon Kaen University, Thailand.

References

[1] M. Fowler and K. Beck, Refactoring. Reading, MA: Addison-Wesley, 1999.

[2] R. C. Martin, Clean code, Prentice Hall, 2009.

[3] L. Ködderitzsch, “Eclipse Checkstyle Plugin,” eclipse-cs.sourceforge.net, 2016. [Online]. Available: http://eclipse-cs.sourceforge.net. [Accessed: 29-Apr-2016].

[4] "FindBugs™ - Find Bugs in Java Programs", Findbugs.sourceforge.net, 2016. [Online]. Available: http://findbugs.sourceforge.net. [Accessed: 29-Apr-2016].

[5] G. Ganea, I. Verebi and R. Marinescu, "Continuous quality assessment with inCode", Science of Computer Programming, 2015.

[6] F. Arcelli Fontana, P. Braione and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.", JOT, vol. 11, no. 2, pp. 5:1, 2012.

[7] “JDeodorant,” The Eclipse Foundation, 2016. [Online]. Available: https://marketplace.eclipse.org/content/jdeodorant. [Accessed: 29-Apr-2016].

[8] “PMD,” 2015. [Online]. Available: https://pmd.github.io/. [Accessed: 29-Apr-2016].

[9] A. Van Hoff, "The case for Java as a programming language", IEEE Internet Computing, vol. 1, no. 1, pp. 51-56, 1997.

[10] B. Eckel, Thinking in Java. Upper Saddle River, NJ: Prentice Hall, 2006.

[11] H. Schildt, Java The Complete Reference, Seventh Edition. McGraw-Hill Publishing, 2006.

[12] J. Goyvaerts and S. Levithan, Regular expressions cookbook. Beijing: Oreilly, 2009.

[13] H. Larkin, "Variables and Reversibility in Object Oriented Regular Expressions," The Sixth IEEE International Conference on Computer and Information Technology (CIT'06), Seoul, 2006, pp. 39-39.

[14] M. C. Franky and J. A. Pavlich-Mariscal, "Improving implementation of code generators: A regular-expression approach," Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En, Medellin, 2012, pp. 1-10.

[15] M. Schmid, "Characterising REGEX languages by regular languages equipped with factor-referencing", Information and Computation, 2016.

[16] Tutorials.jenkov.com, 2016. [Online]. Available: http://tutorials.jenkov.com/java/if.html. [Accessed: 29-Apr-2016].

[17] M. Gaitani, V. Zafeiris, N. Diamantidis and E. Giakoumakis, "Automated refactoring to the Null Object design pattern", Information and Software Technology, vol. 59, pp. 33-52, 2015.