Statement-level AST-based Clone Detection in Java using Resolved Symbols

1st Simon Baars
University of Amsterdam
Amsterdam, the Netherlands
simon.mailadres@gmail.com

***Abstract*—Duplication in source code is often seen as one of the most harmful types of technical debt as it increases the size of the codebase and creates implicit dependencies between fragments of code. Detecting such problems can provide valuable insight into the quality of systems and help to improve the source code. To correctly identify cloned code, contextual information should be considered, such as the type of variables and called methods.**

Comparing code fragments including their contextual information introduces an optimization problem, as this information may be hard to retrieve. It can be ambiguous where contextual information resides and tracking it down may require to follow cross-file references. For large codebases, it could become time-consuming due to the sheer number of referenced symbols.

We propose a method to efficiently detect clones taking into account contextual information. We introduce a tool that uses an AST-parsing library named JavaParser to detect clones and retrieve contextual information. Our method parses the Abstract Syntax Tree retrieved from JavaParser into a graph structure, which is used to find clones. This graph maps the following relations for each statement in the codebase: the next statement, the previous statement, and the previous cloned statement.

We find that, when taking into account contextual information in our clone detection, 11% fewer clones are found. Manually inspecting a sample of the difference, we find that they are less relevant for refactoring.

***Index Terms*—clone detection, context, java, parsing, static code analysis**

I. Introduction

Duplicate code fragments are often considered as bad design [4]. They increasing maintenance efforts or causing bugs in evolving software [6]. Changing one occurrence of a duplicated fragment may require changes in other occurrences [9]. Furthermore, duplicated code was shown to account for up to 25% of total system volume [3], entailing more code to be maintained.

Several tools have been proposed to detect duplication issues [11], [17], [15]. These tools can find matching fragments of code, however do not take into account contextual information of code. An example of such contextual information is the name of used variables: many different methods with the same name can exist in a codebase. This can obstruct refactoring opportunities.

We describe a method to detect clones while taking into account the contextual information and introduce a tool to detect clones taking into account such contextual information. We run the tool over a corpus of 2,267 Java projects. We find that taking into account contextual information results in 11% less clones than found when not taking this information into account. Manually inspecting a sample the difference, we find that all clones with different contextual information in our sample are less relevant for refactoring.

II. Background

We first explain relevant code clone terminology and definitions. Next, we describe the JavaParser tool, used to retrieve contextual information of source code.

A. Code clones

We use two concepts to argue about code clones [12]:
Clone instance: A single cloned fragment.
Clone class: A set of similar clone instances.

Duplication in code is found in many different forms. Most often duplicated code is the result of a programmer reusing previously written code [5], [2]. Sometimes this code is then adapted to fit the new context. To reason about these modifications, several clone types have been proposed [12]:
Type I: Identical code fragments except for variations in whitespace (may be also variations in layout), and comments.
Type II: Structurally/syntactically identical fragments except variations in identifiers, literals, types, layout, and comments.
Type III: Copied fragments with further modifications. Statements can be changed, added or removed next to variations in identifiers, literals, types, layout, and comments. Many studies adopt these clone types, analyzing them further and writing detection techniques for them [13], [8], [19]. In this study, we mainly focus expanding type 1 clone detection to take into account contextual information.

B. JavaParser

JavaParser [16] is a Java library which allows parsing Java source files to an abstract syntax tree (AST). Integrated into JavaParser is a library named SymbolSolver. This library allows for the resolution of symbols using JavaParser. For instance, we can use it to trace references (methods, variables, types, etc) to their declarations (these referenced identifiers are also called "symbols"). Using this, we can find the required contextual information.

To be able to trace referenced identifiers, SymbolSolver requires access to not only the analyzed Java project but also all its dependencies. This requires us to include all dependencies with the project. Along with this, SymbolSolver solves

```java
package com.sb.game;

import java.util.List;

public class GameScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void addTen(int x) {
    x = x + 10; // add number
    Notifier.notifyChanged(x);
    return x;
  }
}
```

```java
package com.sb.fruitgame;

import java.awt.List;

public class LoseScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void concatTen(String x) {
    x = x + 10; // concat string
    Notifier.notifyChanged(x);
    return x;
  }
}
```

Fig. 1. Example of a clone that, although textually equal, is contextually different.

symbols in the JRE System Library (the standard libraries coming with every installation of Java) using the active Java Virtual Machine (JVM).

III. Motivating example

Most clone detection tools [7], [14], [10], [18], [17] detect type 1 clones by textually comparing code fragments (except for whitespace and comments). Although textually equal, method, type and variable references can still refer to different declarations. In such cases, refactoring opportunities could be invalidated. This can make the detected clones less suitable for refactoring purposes, as they require additional judgment regarding the refactorability of such a clone.

Figure 1 shows two clone classes. Merging these clone classes is very hard (and likely not desirable), as both cloned fragments describe different functional behavior. The first cloned fragment is a method that adds something to a `List`. However, the `List` objects to which something is added are different. Looking at the `import` statement above the class, one fragment uses the `java.util.List` and the other uses the `java.awt.List`. Both happen to have an `add` method, however their implementation is completely different.

The second cloned fragment shows how equally named variables can have different types and thus perform different functional concepts. The cloned fragment on the left adds a specific amount to an integer. The cloned fragment on the right concatenates a number to a String.

This shows that not all textually equal clones can be easily refactored.

IV. Contextual information

To solve the issues identified in Section III, we expand clone detection by taking into account contextual information: cloned fragments have to be both textually *and* contextually equal. We check contextual equality of two fragments by validating the equality of the fully qualified identifier (FQI) for referenced types, methods and variables. If an identifier is fully qualified, it means we specify the full location of its declaration (e.g. `com.sb.fruit.Apple` for an `Apple` object).

A. Referenced Types

Many object-oriented programming languages (like Java, Python, and C#) require the programmer to import a type (or the class in which it is declared) before it can be used. Based on what is imported, the meaning of the name of a type can differ. For instance, if we import `java.util.List`, we get the interface which is implemented by all list data structures in Java. However, importing `java.awt.List`, we get a listbox GUI component for the Java Abstract Window Toolkit (AWT). These are entirely different functional concepts. To be sure we compare between equal types, we compare the FQI for all referenced types.

1) Called methods: A codebase can have several methods with the same name. The implementation of these methods might differ. When two code fragments call methods with an identical name or signature, they can still call different methods. Because of this, textually identical code fragments can differ functionally.

We compare the fully qualified method signature for all method references. A fully qualified method signature consists of the fully qualified name of the method, the fully qualified type of the method plus the fully qualified type of each of its arguments. For instance, an `eat` method could become `com.sb.Apple.eat(com.sb.Tool)`.

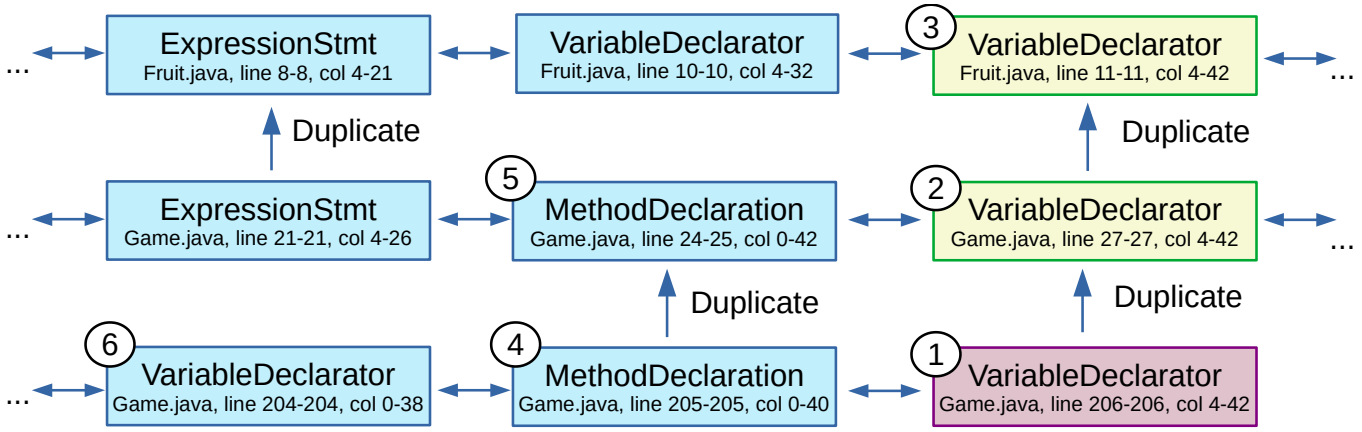

Fig. 2. Example of a clone graph built by CloneRefactor.

2) Variables: In typed programming languages, each variable declaration should declare a name and a type. When we reference a variable, we only use its name. If we use variables with the same name but different types in different code fragments, the code can be functionally unequal but still textually equal.

The body of both methods in Figure 1 is equal. However, their functionality is not. The first method adds two numbers together and the other concatenates an integer and a String. Because of this, we compare cloned variable references by both their name and the FQI of their type.

V. Clone Detection

We develop a tool named CloneRefactor[1] that detects clones that can (relatively) easily be refactored. This tool uses JavaParser [16] to build an AST and to find contextual information (e.g. resolve symbols). We then propose a novel clone detection technique to detect clones using JavaParser.

CloneRefactor uses JavaParser to read a project from disk and build an AST. Each AST is then converted to a directed graph that maps relations between statements. Based on this graph, CloneRefactor detects clone classes and verifies them using the configured thresholds. This process is explained in further detail over the following sections.

A. Generating the clone graph

CloneRefactor parses the AST obtained from JavaParser into a directed graph structure. We have chosen to base our clone detection around statements as the smallest unit of comparison. This means that a single statement cloned with another single statement is the smallest clone we can find. The rationale for this lies in both simplicity and performance efficiency. This means we won't be able to find when a single expression matches another expression, or even a single token matching another token. This is in most cases not a problem, as expressions are often small and do not span the minimal size to be considered a clone in the first place.

[1]CloneRefactor is available on GitHub: https://github.com/SimonBaars/CloneRefactor

B. Filtering the AST

As a first step towards building the clone graph, we preprocess the AST to decide which AST nodes should become part of the clone graph: we exclude package declarations and import statements. These are omitted by most clone detection tools, as package declarations and import statements are most often generated by the IDE and not relevant for refactoring purposes.

C. Building the clone graph

Building the clone graph consists of walking the AST in-order for each declaration and statement. For each declaration/statement found, we map the following relations:

- The declaration/statement preceding it.
- The declaration/statement following it.
- The last **preceding** declaration/statement with which it is cloned.

We do not create a separate graph for each class file, so the statement/declaration preceding or following could be in a different file. While mapping these relations, we maintain a hashed map containing the last occurrence of each unique statement. This map is used to efficiently find out whether a statement is cloned with another. An example of such a graph is displayed in Figure 2.

The relations *next* and *previous* in this graph are represented as a bidirectional arrow. The relations representing duplication are directed.

D. Comparing Statements/Declarations

In the previous section, we described a "duplicate" relation between nodes in the clone graph built by CloneRefactor. Whether this duplicate relation exists between two nodes is determined by taking into account the contextual information. For *method calls* we determine their fully qualified method signature for comparison with other nodes. For all *referenced*

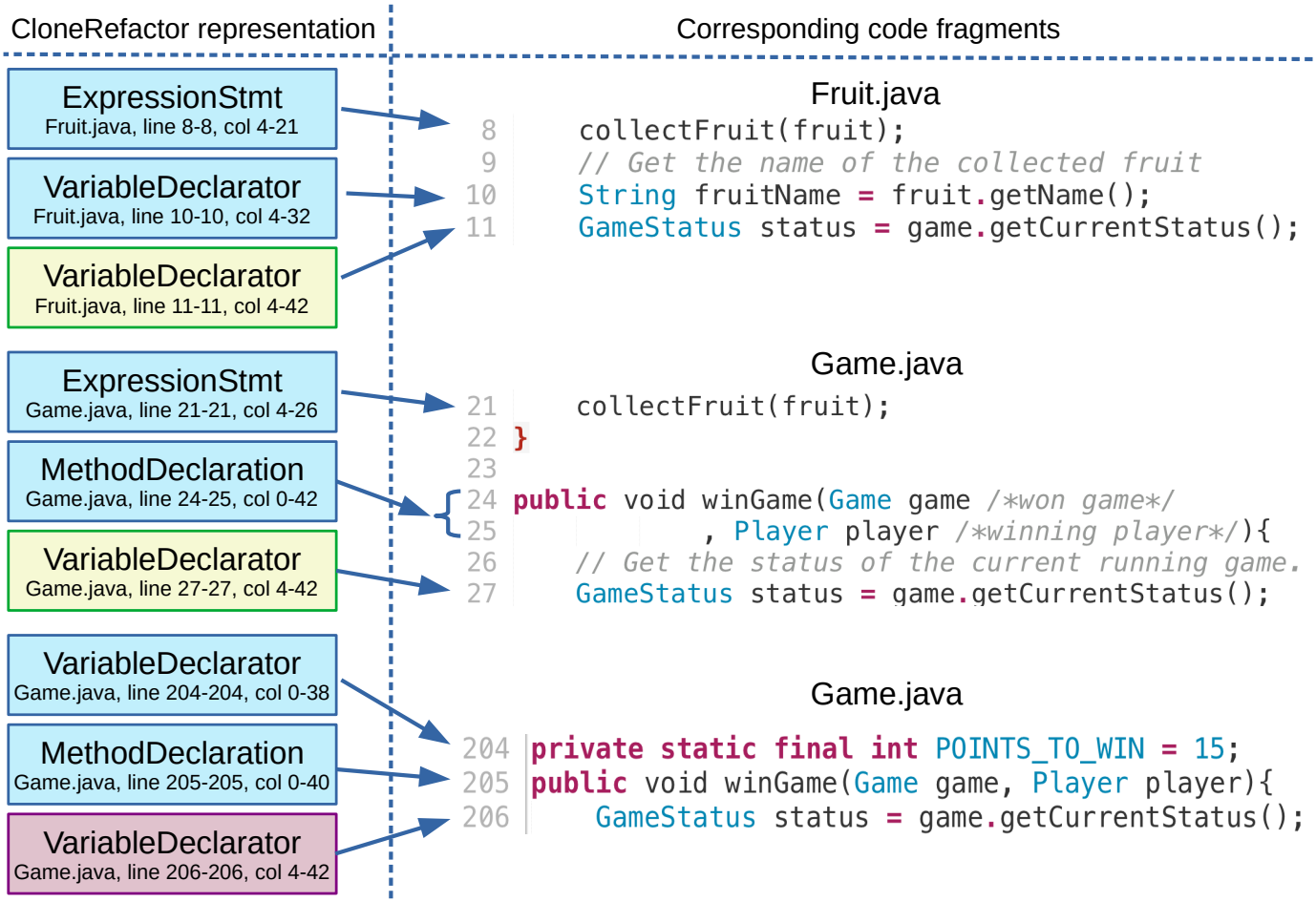

Fig. 3. CloneRefactor extracts statements and declarations from source code.

types we use their fully qualified identifier (FQI) for comparison with other nodes. For *variables* we compare their fully qualified type in addition to their name.

E. Mapping graph nodes to code

The clone graph, as explained in Section V-C, contains all declarations and statements of a software project. However, declarations and statements may themselves have child declarations and statements. To avoid redundant duplication checks, we exclude the body of each node.

Figure 3 shows an example of how source code maps to AST nodes. On line 24-25 of the code fragment is a `MethodDeclaration`. The node corresponding with this MethodDeclaration denotes all tokens found on these two lines, line 24 and 25. Although the statements following this method declaration (those that are part of its body) officially belong to the method declaration, they are not included in its graph node. Because of that, in this example, the `MethodDeclaration` on line 24-25 will be considered a clone of the `MethodDeclaration` on line 205 even though their bodies might differ. Even the range (the line and column that this node spans) does not include its child statements and declarations.

F. Detecting Clones

After building the clone graph, we use it to detect clone classes. We start our clone detection process at the final location encountered while building the graph. As an example, we convert the code example shown in Figure 3 to a clone graph as displayed in Figure 2.

Using the example shown in Figure 2 and 3 we can explain how we detect clones on the basis of this graph. Suppose we are finding clones for two files and the final node of the second file is a variable declarator. This node is represented in the example figure by the purple box (1). We then follow all “duplicate” relations until we have found all clones of this node (2 and 3). We now have a clone class of three clone instances each with a single node (1, 2 and 3).

Next, we move to the *previous* line (4). Here again, we collect all duplicates of this node (4 and 5). For each of these duplicates, we check whether the node following it is already in the clone class we collected in the previous iteration. In this case, (2) follows (5) and (1) follows (4). This means that

node (3) does not form a 'chain' with other cloned statements. Because of this, the clone class of (1, 2 and 3) comes to an end. It will be checked against the thresholds, and if adhering to the thresholds, considered a clone.

We then go further to the previous node (6). In this case, this node does not have any clones. This means we check the (2 and 5, 1 and 4) clone class against the thresholds, and, if it adheres, consider it a clone. Dependent on the thresholds, this example can result in a total of two clone classes.

Eventually, following only the "previous node" relations, we can get from (6) to (2). When we are at that point, we will find only one cloned node for (2), namely (3). However, after we check this clone against the thresholds, we check whether it is a subset of any existing clone. If this is the case (which it is for this example), we discard the clone.

VI. Experiments and Results

To compare the difference in detected clones when contextual information is considered, we compared the number of clones found when considering contextual information with when it is not considered. For this, we use a corpus of 2,267 Java projects including their dependencies [1].

We find that 167,913 clones are found when contextual information is not considered, whereas 149,569 clones are found when it is considered. We manually analyse a sample of 50 clones that are not found when considering contextual information. We find that these clones are indeed hard to refactor because they describe different functional operations and can thus not be extracted to a new method. Also, most often they were not relevant because, based on our judgement, refactoring would not improve the code design. Often, different methods were called or variables of different types were used.

We also look into the difference in performance when taking into account contextual information. Detecting the clones in all 2,267 projects took 20.83 minutes when considering contextual information. When contextual information was not considered, it took 1.58 minutes. This is mainly because contextual information may be hard to retrieve, because the location of the contextual information may not be explicit. To find contextual information it may also be required to follow cross-file references.

VII. Conclusion

We propose a method to detect clones taking into account contextual information of source code. Contextual information is important because different functional concepts may turn up as clones because they are textually identical.

We define three aspects of source code as the contextual information: a) the type of variable references; b) the location of method references; and c) the location of type references. When these references have the same name but point to different locations, clones may not be easily refactorable. Our results show that most such clones are not relevant for refactoring. This accounts for about 11% of clones. This comes however with a performance trade-off: detecting clones with contextual information took 13 times longer than when not taking contextual information into account (1.6min vs 20.8min).

References

[1] Simon Baars and Ana Oprescu. Towards automated refactoring of code clones in object-oriented programming languages. Technical report, EasyChair, 2019.

[2] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[3] Magiel Bruntink, Arie Van Deursen, Remco Van Engelen, and Tom Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[4] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, second edition, 2018.

[5] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management science*, 54(1):180–193, 2008.

[6] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[8] E Kodhai, S Kanmani, A Kamatchi, R Radhika, and B Vijaya Saranya. Detection of type-1 and type-2 code clones using textual analysis and metrics. In *2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, pages 241–243. IEEE, 2010.

[9] J. Ostberg and S. Wagner. On automatically collectable metrics for software maintainability evaluation. In *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, pages 32–37, 10 2014.

[10] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*, pages 172–181. IEEE, 2008.

[11] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[12] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.

[13] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. IEEE, 2016.

[14] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Ccfindersw: Clone detection tool with flexible multilingual tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 654–659. IEEE, 2017.

[15] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[16] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. Javaparser, 05 2018.

[17] Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330. IEEE, 2014.

[18] Jeffrey Svajlenko and Chanchal K Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600. IEEE, 2016.

[19] Brent van Bladel and Serge Demeyer. A novel approach for detecting type-iv clones in test code. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 8–12. IEEE, 2019.