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 contextual 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, this can require a lot of time due to the sheer number of referenced symbols.

We propose a method to efficiently detect clones taking into account contextual information. To do this, we propose a tool that uses an AST-parsing library named JavaParser to detect clones a retrieve contextual information. Our method first parses the AST retrieved from JavaParser into a graph structure, which is then 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.

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

I. Introduction

Duplicate code fragments are often considered as symptoms of bad design [2]. They create implicit dependencies, thus increasing maintenance efforts or causing bugs in evolving software. Changing one occurrence of such a duplicated fragment may require other occurrences to be changed as well [4]. Also, duplicated code has been shown to add up to 25% of total system volume [1], which entails more code to be maintained.

A lot of tools have been proposed to detect such duplication issues [6], [10], [8]. These tools can find matching fragments of code, but 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 such contextual information and propose a tool to detect clones taking into account this contextual information. Run this 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 use two concepts to argue about code clones [**?**]:
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 [**?**], [**?**]. Sometimes this code is then adapted to fit the new context. To reason about these modifications, several clone types have been proposed [**?**]:
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. A higher type of clone means that it is harder to detect and refactor. Many studies adopt these clone types, analyzing them further and writing detection techniques for them [**?**], [**?**], [**?**].

III. Motivating example

Most clone detection tools [3], [7], [5], [11], [10] 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 type 1 clones less suitable for refactoring purposes, as they require additional judgment regarding the refactorability of such a clone.

Figure 1 shows a type 1 clone with two clone classes. Defining an automatic way to refactor these clone classes is nearly impossible, 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, but apart from that their implementation is completely different.

The second cloned fragment shows how equally named variables can have different types and thus perform different

```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 type 1 clone that is functionally 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 type 1 clones can easily be automatically refactored. In section we describe an alternative approach towards detecting type 1 clones, which results in only clones that can be refactored.

IV. JavaParser

An important design decision for CloneRefactor is the usage of a library named JavaParser [9]. JavaParser is a Java library which allows parsing Java source files to an abstract syntax tree (AST). JavaParser allows to modify this AST and write the result back to Java source code. This allows us to apply refactorings to the detected problems in the source code.

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”). This is useful for the detection of our refactoring-oriented clone types, as they make use of the fully qualified identifiers of symbols.

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

V. Conclusion

References

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.