Real-time management of code clones in an IDE environment

Jakob Hansen

March 24, 2022

Contents

1 Introduction

Refactoring is the process of restructuring code in order to improve the internal behavior of the code, without changing the external behavior[3, 9]. Refactoring is often done in order to eliminate "smelly" code.

A study conducted by Diego Cedrim et al.[2] has shown that while developers tend to refactor smelly code, they are rarely successful at eliminating the smells they are targeting. A large portion of refactorings even tend to make the code smellier. Therefore, automated tools to help developers make better refactorings and perform code analysis is an important field of research.

Duplicated code is a code smell which occurs in practically every large software project. Code clone analysis has recently become a highly active field of research and many tools have been developed to detect duplicated code[4, 7]. However, few of these tools reached and see use in the industry and few have the capability of detecting more intricate types of duplicated code and managing them in a real-time IDE environment.

This thesis will present a tool for industry viable clone detection and management. It will explore the topics of finding and managing clones in real-time, refactoring-oriented clone detection and providing clone management tools in a modern IDE environment.

2 Background

2.1 Software quality

2.1.1 Software quality metrics

2.1.2 How refactoring affects software quality

2.1.3 Duplicated code

As stated, duplicated code damages software quality in practically every large software project. Duplicated code can lead to a plethora of antipatterns like Shotgun-Surgery and Divergent-Change, and will often lead to an increase in technical debt for the project[3, 99].

Statistic about percentage of duplicated code here.

2.2 Code clones

We define a code snippet or code fragment as a piece of software code in a larger software system. A code clone is then defined as a code snippet which is equal to or similar to another code snippet. The two code snippets are both code clones, and together they form a code clone pair.

2.2.1 The clone relation

The clone relation defines a relation between code snippets where snippets which are code clones are related to each other. The clone relation is reflexive and symmetric, but not always transitive. The transitive property depends on the threshold for similarity when identifying code clones. Given

$$a \xrightarrow{clone} b \xrightarrow{clone} c$$

where a, b, c are code snippets and $\xrightarrow{clone}$ gives the clone relation, a is a clone of b, but not necessarily similar enough to be a clone of c, depending on the threshold

for similarity.

2.2.2 Code clone types

Code clones are generally classified into four types[4]. These types classify code snippets as code clones with an increasing amount of leniency. Therefore, Type-1 code clones are very similar, while Type-4 clones are not necessarily similar at all. However, all code clones do still have the same functionality, it is the syntactic and structural differences which distinguish the types. The set of code clones classified by a code clone type is also a subset of the next type, meaning all type-1 clones are also type-2 clones, but not vice versa.

The code clone types are defined as follows:

Type-1 clones are syntactically identical. The only differences allowed are elements without meaning, like comments and white-space.

Type-2 clones are structurally identical. Possible differences include identifiers, literals and types.

Type-3 clones are required to be structurally similar, but not equal. Differences include statements which are added, removed or modified. For this clone type one needs to determine a threshold θ which determines how structurally different snippets can be to be considered Type-3 clones[4].

Type-4 clones have no requirement for syntactical or structural similarity. Therefore, the only requirement is having the same functionality.

Type-1 clones are often referred to as "exact" clones, while Type-2 and Type-3 clones are often referred to as "near-miss" clones[9, 1].

2.2.3 Code clone detection process and techniques

The Code clone detection process is generally split into (but is not limited to) a set of steps to identify clones[7]. This process is often a pipeline of input-processing steps before finally comparing fragments against each other and filtering. The steps are generally as follows:

1. **Pre-processing**: Removing some uninteresting parts, partitioning code into

a set of fragments, determining granularity.

2. **Transformation**: Transforming the fragments into an intermediate representation.
   (a) Extraction: Transforming source code into the input for the comparison algorithm. Can be tokens, AST, dependency graphs, etc.
   (b) Normalization: Optional step which removes superficial differences such as comments, whitespace and identifier names. Often useful for identifying type-2 clones.
3. **Match detection**: Performing the comparisons which outputs a set of candidate clone pairs.
4. **Formatting**: Convert from candidate clone pairs from the transformed code back to clone pairs in the original source code.
5. **Post-processing/Filtering**: Ranking and filtering manually or with automated heuristics
6. **Aggregation**: Optionally aggregating sets of clone pairs into clone classes

As stated, not all clone detection techniques will necessarily follow all these steps.

Code clone detection techniques are techniques which can be applied to detect clones, with various advantages and disadvantages. Some of the most popular techniques are as follows:

Text-based approaches do very little processing on the source code before comparing. Simple techniques such as fingerprinting or incremental hashing have been used in this approach. Dot plots have also been used in newer text-based approaches, placing the hashes of fragments in a dot plot for use in comparisons.

Token-based approaches transform source code into a stream of tokens, similar to lexical scanning in compilers. The token stream is then scanned for duplicated subsequences of tokens. Since token streams rarely include superficial differences such as whitespace, indentation and comments, this approach is more robust to such differences. Concrete names of identifiers and values are abstracted away when comparing the token-stream, therefore type-2 clones can easily be identified. Type-3 clones can also be identified by comparing the fragments tokens and keeping clone pairs which are lexically not more different than a given threshold. This can be solved with dynamic programming[1].

Syntactic approaches transform source code into either parse trees or abstract syntax trees and find clones using either tree matching algorithms or structural metrics. For tree matching, the common approach is to find similar subtrees in the parse tree / AST, which are then deemed as clones. One way of finding similar subtrees is to hash subtrees into buckets and compare them with a tolerant tree matching algorithm. Variable names, literal values and other source may be abstracted to find type-2 clones more easily. Metrics-based techniques gather metrics for code fragments in the parse tree / AST and uses the metrics to determine if the fragments are clones or not. One way is to use fingerprinting functions where the fingerprint include certain metrics, and compare the fingerprints of all fragments to find clones.

Chunk-based approaches decompose chunks of source code into signatures which are compared. Chunk-size is based on selected granularity, which can be functions, blocks, etc. Signatures can for example be based on some software metrics. Machine learning has been used in this approach ...

Hybrid approaches combine multiple approaches in order to improve detection. For example Zibran et al.[9] developed a hybrid algorithm combining both token-based suffix trees for type-1 and type-2 clone detection, with a k-difference dynamic programming algorithm for type-3 clone detection.

- Text-based
- Lexical/Token-based
- Syntax-based (AST)
- Metrics-based (Comparing gathered metrics)
- Semantics-based (static program analysis)

2.2.4 Clone aware development

Developers are often not aware of the creation of clones in their code. Clone aware development involves having clone management as a part of the software development process. Since code clones can be hard to keep track of and manage, tools which help developers deal with clones are useful. However, Mathias Rieger et al. claims that a problem with many detection tools is that the tools "report large amounts data that must be treated with little tool support."[6, 1]. Existing tools which partly solves this problem are presented below.

2.2.5 IDE-based clone management

There are many existing clone management tools, however the most useful tools for clone aware development are the tools which are integrated into an IDE and offer services to the programmer while developing in real-time.

The IDE-based tools which exist can be categorized as follows[8, 8]:

- *Copy-paste-clones:* This category of tools deals only with code snippets which are copy-pasted from another location in code. These tools therefore only track clones which are created when copy-pasting, and does not use any other detection techniques. Therefore, this type of tool is not suitable for detecting clones which are made accidentally, since developers are aware that they are creating clones when pasting already existing code snippets.
- *Clone detection and visualization tools:* This category of tools has more sophisticated clone detection capabilities and will detect code clones which occur accidentally.
- *Versatile clone management:* This category of tools covers tools which provide more services than the above. Services like refactoring and simultaneous editing of clones fall under this category.

There are a few existing IDE-tools which have seen success in real-time detection of clones:

- Minhaz et al. introduced a technique for performing real-time focused searches, i.e. searching only for code clones of a given code snippet. This technique can also detect Type-3 clones[9]. This technique was later used in the tool *SimEclipse*[8]. Since this tool can only detect clones of a code snippet which the developer actively selects, this tool is not well suited for finding accidental clones.
- Another tool, SHINOBI can detect code clones in real-time without the need of the developer to select a code snippet, however it can only detect type-1 and type-2 code clones[5].
- The modern IDE IntelliJ has a built-in duplication detection and refactoring, it's able to detect type-1 and type-2 code clones at a method granularity and refactors by replacing one of the clones with a method call. This tool also

requires the user to actively select the method which will be checked for clones.

No tools which we are aware of have the capability of both reporting code clones in real-time without fragment-selection and reporting type-3 clones.

2.3 The Language Server Protocol

3 The way forward

This thesis will present and evaluate a modern tool which provides clone management capabilities in a real-time IDE environment. The main goal will be to create a tool which fits well into the development cycle and works in a real-time IDE environment. Areas of focus will therefore be:

- Real-time detection and management of code clones
- Code clone refactoring and detection of refactoring-oriented clones
- IDE tooling and IDE agnostic tooling like LSP.
- Clone ranking, which clones are allowed to stay?

3.1 LSP for IDE-based clone management

References

[1] Brenda S. Baker and Raffaele Giancarlo. Sparse dynamic programming for longest common subsequence from fragments. *Journal of Algorithms*, 42(2):231–254, 2002.

[2] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo da Silva Sousa, Rafael Maiani de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint*

Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pages 465–475. ACM, 2017.

[3] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

[4] Katsuro Inoue. *Introduction to Code Clone Analysis*, pages 3–27. Springer Singapore, Singapore, 2021.

[5] Shinji Kawaguchi, Takanobu Yamashina, Hidetake Uwano, Kyohei Fushida, Yasutaka Kamei, Masataka Nagura, and Hajimu Iida. Shinobi: A tool for automatic code clone detection in the ide. pages 313–314, 01 2009.

[6] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *11th Working Conference on Reverse Engineering*, pages 100–109, 2004.

[7] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[8] Md Sharif Uddin, Chanchal K. Roy, and Kevin A. Schneider. Towards convenient management of software clone codes in practice: An integrated approach. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, page 211–220, USA, 2015. IBM Corp.

[9] Minhaz F. Zibran and Chanchal K. Roy. Ide-based real-time focused search for near-miss clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1235–1242, New York, NY, USA, 2012. Association for Computing Machinery.