A Review on Code Smell Techniques using Graph Theory

Chaitanya Kulkarni, S.D. Joshi

ckulkarni47@gmail.com\(^1\), sdj@live.in\(^2\)
Department of Computer Engineering\(^1,2\)
Shri Jagdishprasad Jhabarmal Tibrewala University, Rajasthan, India\(^1,2\)

Abstract—In software engineering field of code clone recognition, which essentially comprises of the location of the code clones that is acquainted in programming due to reuse approach amid improvement of programming. At the advancement stage for simplicity of programming, the developers reuse the code by slight adjustments. In the event that the product has a high level of likelihood clones, it might bring about numerous issues for the product, from the support perspective. There are numerous apparatuses accessible, however yet exactness required in clone location can't be guaranteed to amazing degree. A portion of the apparatuses accessible don't identify clones with great effectiveness, Moreover they devour a considerable measure of time. The aim of this proposition is to reveal the method by which proposed instrument can effectively recognize different sorts of clones with proficient adaptability and striking streamlining.

Keywords—Code Clone, Code Smell, Code Clone Management, Refactoring of Code Clone.

I. INTRODUCTION

Clone is catchphrase that is utilized for duplication of things. Code clone is only the duplication of code. Reuse through replicating and sticking source code is regular practice. Supposed programming clones are the outcomes. A code part CF1is an arrangement of code line is clone like another code bit CF2, having analogous properties i.e.

\[ F(CF1) = F(CF2) \]

Two sections that have comparable properties/usefulness are alluded as clone pair (CF1, CF2). F (CF1) and F (CF2) give the yields of the code parts CF1 and CF2 separately. In the event that the yield of the sections is comparable, then those parts are said to be clone. CF1 and CF2 may have literarily closeness or practical similitude. The examination says that in programming advancement, software engineers much of the time reuse code sections by duplicate glue operations. Those code parts, which are comparative or indistinguishable, are called code clones.

So it can be said that Sequences of copy code are known as code clones, and the mechanized procedure of discovering duplication code is known as clone recognition. The grouping of source codethat happens more than once either either inside system or crosswise over various projects possessed or kept up by some amount. Copy code is viewed as undesirable, for various reasons.

II. HISTORY & BACKGROUND

The history of Code clones have for quite some time been perceived as terrible stenches in delicate product frameworks and are considered to bring about upkeep issues amid development. It is comprehensively accepted that the more clones two records offer, the all the more regularly they must be changed together. This connection between clones and change couplings has been proposed however neither evil spirit-strated nor measured yet. In any case, given such a connection it would sim- plify the recognizable proof of rebuilding hopefuls and decrease change couplings. In this paper, we look at this connection and examine if a cor- connection between code clones and change couplings can be confirmed. For that, we propose a structure to examine code clones and relate them to change couplings taken from discharge history examination. We approved our structure with the open source venture Mozilla and the consequences of the acceptance demonstrate that despite the fact that the connection is measurably unverifi- capable it infers a sensible measure of situations where the connection exists. In this manner, to find clone possibility for rebuilding we also propose an arrangement of measurements and a representation system. This permits one to spot where a connection amongst's cloning and change coupling exists and, accordingly, which records ought to be rebuilt to ease further advancement architecture paradigm [6]. Design patterns are appealing. Most developers have already suffered from the consequences of a bad design, so they invest a lot of time into upfront design and usage of design patterns to avoid ending up in a similar situation again.
One problem which you might face is over-engineering. A certain design pattern is implemented in anticipation of a future need; this need might never arise though. So, you make a design more sophisticated or flexible than it needs to be which comes with an additional effort in maintaining such a system. An overly complex system, usually leads to new developers not understanding your code or developers starting to work on separate, discrete parts of the system potentially duplicating code because they do not know each others’ part of the system.

III. DESIGN ISSUES

The issue explanation uncovers a requirement for such apparatus since existing research and devices does not highlight astounding execution in clone twin examination. A few instruments use compiler and break down the outcomes in light of yield from compiler. Such devices for the most part apply Software Testing Scripts and process is unreasonable. Utilizing Testing example won’t affirm that all cases are tried. A few apparatuses think about two documents at once. Accordingly multi-code report examination is crucial. Perusing code documents for coordinated examination in reality builds postponement and I/O operation in Code mining. In this manner, the framework to create does not base on Compiler Testing Technique but rather use information mining and machine learning approach.

The above fig.1 code smell mapping approach for code smell detection indicates that code clone refactoring and detection code smell with the help of graph mechanism in program dependency graph Unification and Acess Refactoring with graph major concerned. The User might be Technical/Non-specialized then can distinguish the code clones efficiently by writing or embeddings the code into the textbox gave by the client interface. There will be a given look at catch by clicking which the client can acquire the code location results. Design patterns represent a simple and flexible way to design software applications. They are tested, and proven development paradigms which can be seen as templates which you apply to a concrete situation.

When you overly or only use design patterns, you might loose sight of simpler solutions, and introduce indirection where it is not necessary or not necessary yet.

Just as an example, let’s assume that there are different ways of downloading something. You might be tempted to start to use the Strategy pattern and implement different download strategies while it might be enough to simply use a conditional statement and a couple of private methods.

IV. Proposed Methodology

The proposed technique includes Code lessening and Normalization. The Code lessening module is the most vital piece of both the customer and additionally the server application. The key part of our methodology is lessening the source code with the assistance of standardization. The other existing clone discovery frameworks for the most part concentrating on printed coordinating and other recognition approaches cause extensive circle space utilization. Besides the clone recognition requires delivering an immense database loaded with the code, for correlation. These variables are dangerous to productivity and working of a framework. Subsequently, the proposed framework has its highlight over the Code diminishment and Normalization forms. Code Reduction makes the code to minimize and can be put away in the base space in framework memory. In addition it likewise makes the clone identification less demanding with this methodology. For lessening the codes the standardization procedures are utilized i.e. Variable Normalization and Process Normalization. Variable Normalization joined with rearranged indexing makes the discovery of Type I and Type II conceivable. The Type III and Type IV clones are distinguished utilizing process standardization. Code Clone Types There are basically two kinds of similarities between two code fragments. Two code fragments can be similar based on the similarity of their program text or they can be similar in their functionalities without being textually similar. The first kind of clones are often the result of copying a code fragment and then pasting to another location. In this section, we consider clone types based on the kind of similarity two code fragments can have: • Textual Similarity: Based on the textual similarity we distinguish the following types of clones.
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 for variations in identifiers, literals, types, layout and comments. Type III: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

**V. Refactorability Analysis**

Input to the stage 2 is output of the stage 1 i.e. detected clones or let’s say group of code fragments detected as clones. Three steps are there to be carried out to check refactorable opportunities. In this step, we consider two code fragments can be unified and therefore refactored if they share common nesting structure. Nesting structure of the input clone fragments is analyzed. This is useful in finding maximal isomorphic sub-trees. It is assumed that the code fragments can be unified only if they are having an identical nesting structure. It cannot specify that two source code fragments can be unified simply if they consist of equal nesting arrangement [4]. Each matched sub-tree will be further investigated as a separate clone refactoring opportunity in the further steps. Clone scientific categorizations can be valuable for streamlining of location and reengineering methods. By knowing the frequencies with which distinctive classifications of clones happen in source code, we can focus our endeavors on the most conspicuous sorts or on the sorts which appear to be most applicable to the reengineering job that needs to be done. In the accompanying, we arrange the diverse clone scientific categorizations from the writing taking into account three qualities, similitudes between the clones, area of the clones. Behavior smells are smells we encounter while compiling or running tests. We don't have to be particularly observant to notice them as they will present themselves to us via compile errors or test failures. The static code analysis (Line 19) involves analyzing the AST by traversing the tree. During this step, we extract CSS style usage, objects, properties, inheritance relations, functions, and code blocks to calculate the smell metrics. If the calculated metrics violate the given criteria (τ), the smell is returned.

for example for bad smell opf source code

```java
final class TextFile {
    private final File file;
    TextFile(File src) {
        this.file = src;
    }

    public int grep(Pattern regex) throws IOException {
        Collection<String> lines = new LinkedList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(this.file))) {
            while (true) {
                String line = reader.readLine();
                if (line == null) {
                    break;
                }
            }
            int total = 0;
            for (String line : lines) {
                if (regex.matcher(line).matches()) {
                    ++total;
                }
            }
            return total;
        }
    }
}
```

In this above source code This method first loads the content of the file. Second, it counts how many lines match the regular expression provided. So why does method grepsmell.

![Flow Diagram of Statement Mapping](image1)

Fig. 2 Flow Diagram of Statement Mapping
For the simplicity of programming, it is a common sight that software engineers make some slight changes in the code and then reuse the same code. Due to these slight changes done in the code, there is a high chance of generating clones for such types of codes. This gives rise to numerous issues in the product. Although there are numerous tools for checking the code clones, the accuracy cannot be guaranteed by these tools. These tools do not identify a large part of the clones and a large amount of time is wasted in this detection. Our goal is to first detect if the cloned codes can be refactored or not and if yes, then how to refactor the cloned codes. Refactoring techniques have gained popularity because of their ability in making more agile code. These activities perform with an aim of improving the software quality, making them easier to understand, maintain, improvements on the software artifacts. Refactoring allows reorganizing code keeping in mind to protect the external behavior.

VI. ANT REFACTORIZATION

According to the C.K.Roy the refactorability of software clones having the several clone detection tools and techniques in which they are identified by the clones in a code which are influenced and forced by the developer and maintenance engineering [6] The software system itself do not occur the clones by themselves but they are introduced by an accident. The literature may contain many approaches which are discussed and the duplication of code in software is refactored safely. There is a drawback in the code clones is that clones in a code will change the code fragment in a same code to overcome the automatic refactoring is developed. Therefore the refactoring doesn’t improve the software always with a code clones and software clone detection and refactoring both will overcome the problem to detect the clones from source code because in a software program there are different kinds of clones which occurs then the refactoring may applied to that source code to eliminate the clones from a program [7]. The duplicated code is seems to be reasonable or beneficial design where it is correct and useful detection of clones in a code and then refactoring will support always and they also face some problems to remove the clones from program. According to Elmar Juergens [8] the code clones are essential to remove from the program which are affected by the software that may uses the detection of inconsistent clones to identify the clones and remove the clones by tools which are provide. According to Angela Lozano [6] the effects of clones are changing due to the similar code fragments in the source code and they are assessed by using the clone detection tools without changing the behaviour of program. The clones are harmful and complex to understand and they lead to the incomplete updates to generate the bugs. The post and pre conditions are the refactoring techniques which will group the clone and the remove the invariants from them [9]. Subterranean insect Refactoring The metric qualities appeared in Table 11 allude to the discretionary bundle of Ant 1.8.2. The bundle is made out of 340 classes. The qualities for the measurements at class level rather are appeared in Table 12. A short sign (−) implies a negative effect on the metric quality and an or more sign (+) a positive effect; an equivalent sign (=) implies that the worth did not change. For instance, if the multifaceted nature diminishes, this speaks to a positive effect, and henceforth, we have an or more sign (+) for CC or WMC measurements.

We can watch that refactoring operations lessened the lines of code. Truth be told, as should be obvious in Table 11 the LOC esteem diminished by 0.38%. The quantity of strategies rather diminishes just if the refactoring operations erase a greater number of strategies than the made ones: the Extract Method refactoring procedure makes a strategy, while the Replace Method system erases a technique. For this situation, the NOM esteem diminished by 0.65%. This outcome is advocated by the way that, by and large, the refactoring comprised in separating classes and techniques to be utilized by numerous customers; this is additionally the motivation behind why likewise the CF esteem diminished. The CC metric quality diminished by 0.65%; actually, the disintegration of capacities, acknowledged by the refactoring procedure Replace Method, is a notable strategy for decreasing the cyclomatic multifaceted nature. Refactoring copied code enhances likewise the solidity of the venture and the estimation of dynamics: the DMS esteem diminished by 25%. The outcomes in Table 12 demonstrate that the metric estimations of the refactored classes enhanced extensively. Through the Extract Method and the Replace Method refactorings, the WMC esteem diminished altogether. RFC diminished with WMC, on the grounds that it is outstanding that a relationship exists between the two measurements. LCOM esteem diminished in light of the fact that, concerning the underlying condition of the framework, the use of the Replace Method refactoring brought about less techniques getting to the same traits of the refactored class. Extricate Method and Extract Class procedures decidedly affected LCOM: it is notable that these strategies enhance union amongst techniques and classes [10]. Refactoring is generally persuaded by seeing a code smell.[2] For instance the current technique might be long, or it might be a close copy of another close-by strategy. When perceived, such issues can be tended to by refactoring the source code, or changing it into another structure that carries on the same as before however that no more "smells". For a long normal, one or more littler subroutines can be extricated; or for copy schedules, the duplication can be evacuated and supplanted with one shared capacity of source code.

VII. LIMITATIONS

In code clone management we studied that number of tools are available to detect the code clone but no any tool describe about how to refactoring of code clone. In proposed system limitations is that at time only two code clone fragment take as an input. Clone pair could not be support of this methodology. We will propose to assessment of large number of code clone refactored.
VIII. CONCLUSION

This approach introduces an important and missing feature in clone management i.e. refactorability analysis which was unsupported previously. To achieve this goal, here is a technique which first matches the statements of the clones in such a way that minimizes the number of differences between them. After this, these differences are examined against a set of preconditions to determine whether they can be parameterized without changing the program behaviour. If no precondition violations are found, provided tool support for the automatic refactoring of the clones. Here we calculated number of lines, number of brackets and number of imports count by using two code fragments. Also show given code fragments are identical nesting structure or not.

REFERENCES


