ISSN No:-2456-2165

Building a Library for Automatic Duplicate Code Detection

Dufan Quraish Shihab[1], Fadilah Fahrul Hardiansyah[2], Desy Intan Permatasari[3],
Umi Sa'adah[4], Jauari Akhmad Nur Hasim[5], Andhik Ampuh Yunanto[6], Irma Wulandari[7]
Department of Informatics and Computer Engineering
Politeknik Elektronika Negeri Surabaya
Surabaya, Indonesia

Abstract:- The existence of duplicate code can be one of the factors that complicate the software maintenance process. This can be avoided by detecting and refactoring. Duplicate code detection is generally done manually, so it is quite time consuming and makes developers less productive. This research proposes the creation of a library to automatically detect duplicate code. This research goal is to avoid detecting duplicate codes manually. The proposed library uses a new approach, combine text and tree bases as a method of detecting duplicate code. Tree base as a representation of code structure. The textbase is using pretty-printing, represents the fragment code in the form of text for comparison. The threshold used is 30%. If the comparison results are above the threshold, the fragment code is said to be a duplicate code. The output of the library is a list of codes that are indicated by duplicate codes or called clone pairs. This library can detect duplicate code Type-1 and Type-2. Manual duplicate code detection requires a very long time because the comparison process is complex. While duplicate code detection with this library, only takes 5.57 seconds. With this very significant time efficiency, it will make software developers more productive.

Keywords:- *Duplicate Code; Clone Pair; Automatic Detection.*

I. INTRODUCTION

The process of maintaining software requires adding new functions or modifying existing functions [1]. As a result, the structure of software becomes more complex as time passes. Increased software complexity has the potential to cause code smell.

Code smells are a set of common signs which indicate that your code is not good enough and it needs refactoring to finally have a clean code. If left unchecked, code smell has the potential to cause bugs, errors, or gaps in security in the future [2]. Code smell can reduce aspects of program understandability and maintainability. Understandability is the quality of a system that can be understood or read. While maintainability is an aspect related to speed, accuracy, security, and economics of maintenance activities. Maintenance is an activity that starts from the time the software starts to be used until the software can't be used anymore. The low understandability makes the program code difficult to understand, consequently, the maintainability decreases and the maintenance process is more difficult to do. This can be detrimental to the developer because it requires a large enough cost. Duplicate code is one of the factors that make the software maintenance process more difficult [3].

Duplicate code is a type of code smell where there are parts of code that are very similar in software systems. Several studies [4, 5, 6] show that duplicate codes are often found on a large codebase. Duplicate code occurs as a result of reusing fragment code by copying and pasting with or without minor adaptations in software development.

Duplicate code detection is generally done manually. For that, the developer needs to read the program code in a file as a whole. Then the developer needs to determine which fragment code is indicated by duplicate code from the program code that is read. To determine this, the developer needs to make a comparison of code in one place with another place. Because there are many steps in manual detection, this process requires a considerable amount of time. The size and complexity of the software affect the time and accuracy in making the detection. As a result, developer productivity decreases. So that the specified software development schedule can be missed. Cost allocations have also increased.

II. RELATED WORKS

This section will explain the related works for building a library in this research.

Roy et. al. [11] presented clone code detection techniques and tools, provides concise explanations with comprehensive surveys and hypothetical evaluations based on editing scenarios. Jeon et. al. [7] defined rules of inference to automatically identify several candidates and refactoring strategies to change one of several candidates into the desired design pattern structure. Opdyke [13] defined several programs that specifically have restructuring operations (refactoring) to support the design, evolution, and reuse of object-based application frameworks. Aristyagama [2] provided a semi-automatic bad smell code detection design framework to deal with the problem of standardization of bad smell code in team programming. Kamiya et. al. [4] presented a clone detection tool called CCFinder with transformation rules and token-based comparisons, and optimization techniques to improve performance and efficiency. Higo et. al. [10] presented refactoring support tools called Aries to characterizes the clone code with several metrics and suggests how to remove them. In other words, Aries tells the user which clone codes can be removed and how to delete them. Bulychev and Minea [9] presented a duplicate code detection tool called Clone Digger to takes the source file name and threshold value as parameters. This tool generates HTML files with a list of clones. Each pair is reported statement after statement by displaying differences. Roy and Cordy [14] provided a new clone detection method called NICAD is based on a two-stage approach: identification and normalization of potential clones using flexible *pretty-printing* and code normalization, followed by a simple text comparison of potential clones using dynamic clusters.

III. PROPOSED IDEA

This research proposes the creation of a library to automatically detect duplicate code. This research goal is to avoid detecting duplicate codes manually. The proposed library uses a new approach, combine text and tree bases as a method of detecting duplicate code. Tree base as a representation of code structure. The textbase is using pretty-printing, represents the fragment code in the form of text for comparison. The threshold used is 30%. If the comparison results are above the threshold, the fragment code is said to be a duplicate code. In Fig. 1 is a research system design that illustrates the steps of the duplicate code detection process.

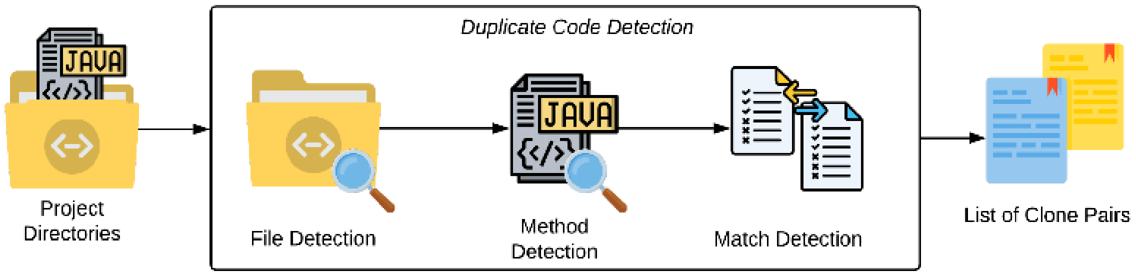

Fig 1:- System design of duplicate code detection

Based on the design in Fig. 1, to get a list of fragment codes that are indicated by duplicate code or clone pairs, the user must first enter the directories of the software.

The system is looking for the code program files that are in that directory. After getting all the program files, the files will be read for method detection to get all the methods in files. Based on the methods obtained, the system will look for the contents of the methods of clone pairs by making a comparison. The part of contents in the method or fragment code is compared with other fragment code to get the fragment code pairs indicated by duplicate code or called clone pairs.

IV. SYSTEM DESIGN

This section will explain more detail about the approach to building a library used as a solution to this research.

A. File Detection

File detection is the first step of duplicate code detection. File detection is based on software directories entered by the user. The file detection design is shown in Fig. 2. This design was obtained by reference [15].

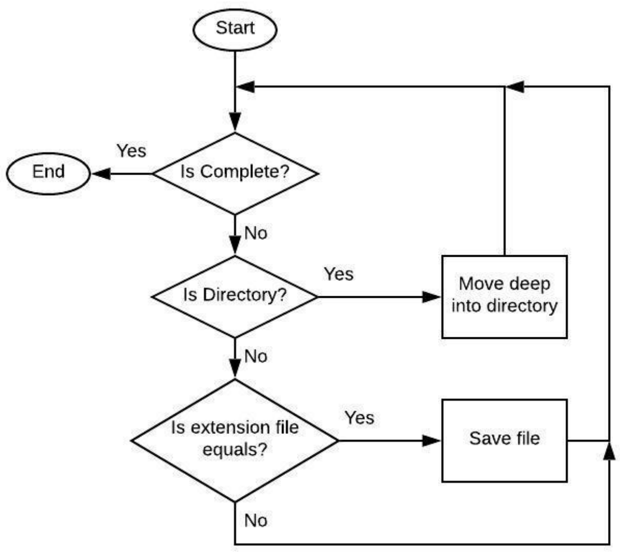

Fig 2:- System design of file detection

Based on the design in Fig. 2, the steps are below:

- ➢ The directories will be read by the system. The system checks readable directories. The system checks whether that is a directory or not.
- ➢ If the result is a directory, then it will move into that directory.
- ➢ If the result is a file, then the file will be checked, whether it has the file extension as desired.
- ➢ If the file extensions match, then the file will be saved for the next process.

B. Method Detection

The next step is reading the method for each file. Below in Fig. 3 is the system design of method detection.

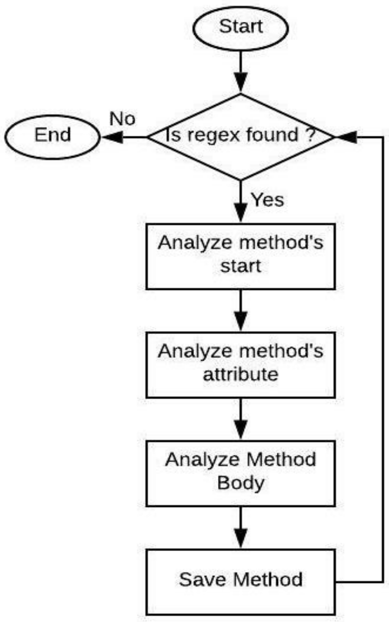

Fig 3:- System design of method detection

The design in Fig. 3 was obtained by reference [15]. Based on the design in Fig. 3, the steps are below:

- ➢ To find a method, we need global characteristics that are owned by the method. These characteristics are defined as regex (regional expression). Regex is used to search for sentences that fit the specified conditions. The system obtains all the methods based on regex.
- ➢ Then the system analyzes to determine the beginning and the end of the method. It aims to get the overall method.
- ➢ After the method has been successfully read, then the system analyzes to obtain the attributes of the method. The method has several attributes, including keywords, return types, names, parameters, and exceptions.
- ➢ The next step is to get the contents or body of the method. Based on that content, the system read each statement and the variables that are used in the method.
- ➢ If the contents of the method have been successfully obtained, then the method has been successfully read perfectly, and the method can be stored for the next process.

C. Match Detection

Match detection is looking for fragment code pairs that are indicated as duplicate code or can be called clone pairs. The output of this process is clone pairs for each file. This process is the next step after the method detection has been successful In Fig. 4 is a match detection system design. This design was obtained by reference [11].

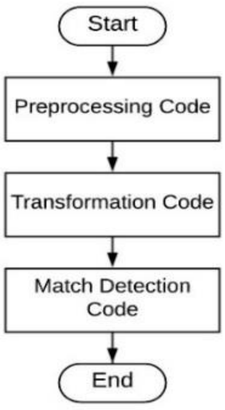

Fig 4:- System design of match detection

Based on Fig. 4, match detection has several phases that need to be passed. Below is a description of each phase.

➢ *Preprocessing Code*

This stage will build a candidate that will be used as a comparison unit in finding clone pairs. Candidates are obtained by taking statements in each method in each file. In Fig. 5 below is an example of taking candidates.

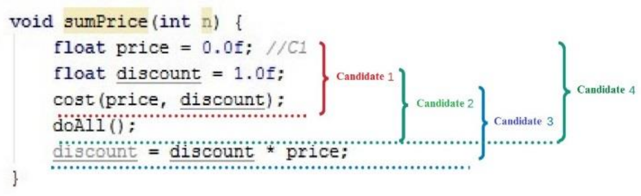
```
void sumPrice(int n) {
    float price = 0.0f; //C1
    float discount = 1.0f;
    cost(price, discount);
    doAll();
    discount = discount * price;
}
```

Fig 5:- Example of taking candidate

Based on Fig. 5, taking statements need to pay attention to the threshold and determined at least three statements to get.

➢ *Transformation Code*

After building a comparison unit or candidate, the statement on each candidate will be changed to an intermediate level representation that is suitable for the comparison process. Starting from normalization to extraction. Normalization is the step to eliminate minor differences such as differences in whitespace, comments, format, or identifier names.

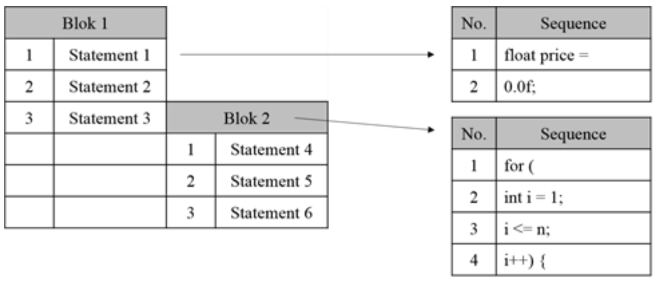

Fig 6:- Transformation code from tree approach to text approach

Extraction transform code to the form that matches with input to the comparison algorithm as shown in Fig. 6. All statements in the method that have been built in the form of a tree approach will be transformed one by one in the form of a text approach, called pretty-printing [14].

➢ *Match Detection Code*

The code that has been transformed is entered into a comparison algorithm where the comparison units are compared with each other to find a match. In Fig. 7 below is an example of comparing two statements.

No.	Sequence 1 (Original)	Sequence 2 (Duplicate)	*Similarity*
1	float price =	int p =	1
2	0.0f;	0;	1

	Unique = 0
	Total item = 2
	UPI = 0 %
		Unique = 0
		Total item = 2
		UPI = 0 %	

Fig 7:- Comparing two statements

The system will compare part of the statement for each statement that has been changed to a text approach. Each similar part will increase the value of similarity by one. The calculation of the value of similarity has been mentioned in reference [14]. The calculation is shown in equation (1) below.

$$Unique\ Percentage\ of\ Items\ (UPI) = \frac{No.\ of\ Unique\ Items * 100}{Total\ No.\ of\ Items}$$

Equation (1) is a formula for calculating the percentage of uniqueness based on the value of similarity. UPI (Unique Percentage of Items) values are obtained by dividing the number of unique values with the total items and then multiplying it with 100%. Zero value in similarity indicates that an increase in unique value by one.

Fig. 7 shows that the UPI value is 0%. We assume that the UPI threshold used is 30%. This threshold obtained by reference [14]. If the UPI value is below the threshold, it can be said to be a clone.

The output of match detection is a list of matches in the transformed code that is represented or combined to form a set of prospective clone pairs.

V. EXPERIMENT AND ANALYSIS

The design of the duplicate code detection experiment is shown in Fig. 8. The experiment will compare manual detection by using a library.

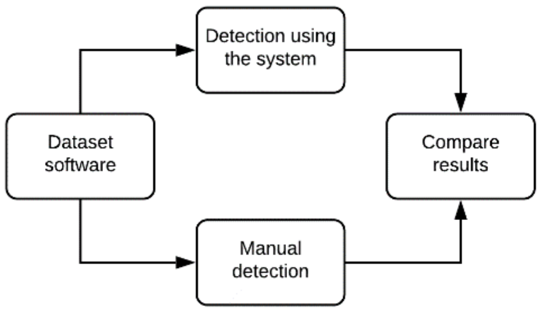

Fig 8:- Experiment design

The experiment is carried out by running the system that has been created by using the directory of the project dataset. This research limits duplicate code detection for Type-1 and Type-2. Duplicate code Type-1 is fragment codes that are identical except for variations in whitespace, layout, and comments. Duplicate code Type-2 is fragment codes that are structurally and syntactically identical except for variations in an identifier, literal, type, whitespace, layout, and comments.

The dataset specification for the experiment is shown in Table 1.

Dataset (java-ml-projects)	
Total Java file	40
Total method	201

Table 1:- Dataset Specification

Based on Table 1, the dataset used in the experiment is a project that uses Java programming language to implement various machine learning. This project has 40 Java files and has 201 total methods.

The experiment device specification is shown in Table 2. The device used for the experiment was the researcher's laptop.

Experiment Device	
Operating System	Windows 7
Processor	Intel(R) Core (TM) i7-4510U CPU @ 2.00GHz, 2.60GHz
RAM	4 GB
System Type	64-bit
HDD	1 TB

Table 2:- Experiment Device Specification

Based on Table 2, the specifications for the experiment are Windows 7 operating system, processor Intel (R) Core (TM) i7-4510U CPU @ 2.00GHz, 2.60GHz, 4GB RAM, 64-bit system type, and HDD 1 TB.

The results obtained by the system are not matched with the results obtained manually. Manual detection detected 16 clone pairs in the dataset software. Detection by the system successfully detected 20 clone pairs in the dataset software. The graph of experiment results for duplicate code detection by the system is shown in Fig. 9 below.

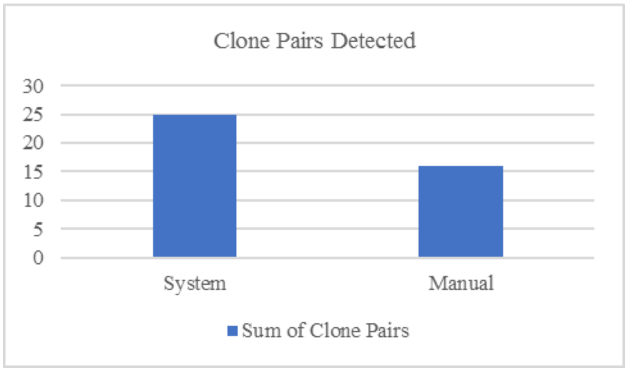

Fig 9:- Experiment results for duplicate code detection by the system

Based on the results in Fig. 9, this library can detect fragment codes that are indicated by duplicate code with an accuracy rate of 64% and an error ratio of 36%. The error ratio that is obtained is due to the existence of an anonymous class which cannot be considered as a single statement, so there is an error in taking the candidate. Anonymous class is still not handled in defining the statement in method detection. This error ratio also depends on how many errors in taking candidates in finding anonymous classes. The more anonymous ones are found, the greater the error ratio is. If anonymous class handling can be ignored, then it can be said that the detection accuracy level reaches 100%.

In addition to analyzing the accuracy of the results, the experiments also analyzed the time needed to detect duplicate code. To test the time needed for the system to detect duplicate code, the experiment is carried out by running the system five times.

The result of the time experiment by the system is shown in Table 3.

The time needed for each experiment (s)					**Average Time (s)**
1	**2**	**3**	**4**	**5**	
5.51	5.64	5.55	5.38	5.75	5.57

Table 3:- Time Experiment by System

Testing the time for manual detection is occur only once. Manual duplicate code detection takes hours or even more than one day due to the complexity of the comparison process. Based on Table 3, duplicate code detection with the system takes only 5.57 seconds. It can be proven that the time needed by the system to detect duplicate code is faster than manual detection.

In Table 3, each experiment generates a fairly stable detection time so that the system has a good level of stability and can avoid human errors.

VI. CONCLUSION

Duplicate code is one type of code smell that makes the software maintenance process more difficult. Duplicate code detection is generally done manually. The number of steps for manual detection can be a very long time even at a lower level. A lot of time consumed causes developers to not be able to add new features. Without the addition of new features, the software development schedule that is set can be missed. At last, this problem can increase the costs.

This research proposes the creation of a library to automatically detect duplicate code. This research goal is to avoid detecting duplicate codes manually. The proposed library uses a new approach, combine text and tree bases as a method of detecting duplicate code. Tree base as a representation of code structure. The textbase is using pretty-printing, represents the fragment code in the form of text for comparison. The threshold used is 30%. If the comparison results are above the threshold, the fragment code is said to be a duplicate code.

Based on the experiment result, it can be concluded that the library can detect most of the duplicate code in the software. This is because the library has not overcome anonymous class detection in the method detection. So the level of accuracy varies. Besides, this library can detect duplicate code Type-1 and Type-2.

Because it is automatic, the library can reduce the time to detect duplicate code. Manual duplicate code detection takes hours or even more than one day due to the complexity of the comparison process. Duplicate code detection with the system only takes 5.57 seconds. This proves that the time needed by the system to detect duplicate code is faster than manual detection.

REFERENCES

[1]. Y. Higo, T. Kamiya, S. Kusumoto dan K. Inoue, “Refactoring Support Based on Code Clone Analysis,” 2004.

[2]. Y. H. Aristyagama, “Framework Deteksi Bad Smell Code Semi Otomatis untuk Pemrograman Tim,” 2016.

[3]. M. Fowler, K. Beck, J. Brant, W. Opdyke dan D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[4]. T. Kamiya, S. Kusumoto dan K. Inoue, “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code,” 2002.
[5]. M. Kim, V. Sazawal dan D. Notkin, “An Empirical Study of Code Clone Genealogies,” 2005.
[6]. Z. Li, S. Lu, S. Myagmar dan Y. Zhou, “CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code,” 2004.
[7]. S.-U. Jeon, J.-S. Lee dan D.-H. Bae, “An Automated Refactoring Approach To Design Pattern-Based Program Transformations In Java Programs,” 2002.
[8]. W. Evans, C. Fraser dan M. Fei, “Clone Detection via Structural Abstraction,” 2007.
[9]. P. Bulychey dan M. Minea, “Duplicate Code Detection Using Anti-Unification,” 2008.
[10]. Y. Higo, T. Kamiya, S. Kusumoto dan K. Inoue, “ARIES: Refactoring Support Tool for Code Clone,” 2005.
[11]. C. K. Roy, J. R. Cordy dan R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” 2009.
[12]. Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto dan K. Inoue, “On Software Maintenance Process Improvement Based on Code Clone Analysis,” 2002.
[13]. W. F. P. Opdyke, “Refactoring Object-Oriented Frameworks,” 1992.
[14]. C. K. Roy dan J. R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” 2008.
[15]. F. Z. P. Putra, D. I. Permatasari, U. Sa'adah, F. F. Hardiansyah dan J. A. N. Hasim, “Rancang Bangun Pustaka untuk Deteksi Otomatis Long Method Code Smell,” dalam The 11th National Conference on Information Technology and Electrical Engineering, Yogyakarta, 2019.
[16]. D. Silva, R. Terra dan M. T. Valente, “Recommending Automated Extract Method Refactorings,” ICPC, 2014.
[17]. S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis dan P. Avgeriou, “Identifying Extract Method Refactoring Opportunities Based on Functional Relevance,” vol. 43, no. 10, 2017.
[18]. C. K. Roy dan J. R. Cordy, “An Empirical Study of Function Clones in Open Source Software Systems,” 2008.
[19]. Sun Microsystems, Inc, Java Code Conventions, Mountain View: Sun Microsystems, Inc, 1997.
[20]. B. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems,” 1995.