PAPER • **OPEN ACCESS**

Generic Code Clone Detection Model for Java Applications

To cite this article: Al-Fahim Mubarak-Ali and Shahida Sulaiman 2020 *IOP Conf. Ser.: Mater. Sci. Eng.* **769** 012023

IOP Conf. Series: Materials Science and Engineering **769** (2020) 012023 doi:10.1088/1757-899X/769/1/012023

Generic Code Clone Detection Model for Java Applications

Al-Fahim Mubarak-Ali[1], Shahida Sulaiman[2]

Faculty of Computing, 26300 Gambang, Pahang, Malaysia.[1]
School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, 81310 UTM Johor Bahru, Johor, Malaysia.[2]

fahim@ump.edu.my[1],shahidasulaiman@utm.my[2]

Abstract. Code clone is a common term used for codes that are repeated multiple times in a program. There are Type 1, Type 2, Type 3 and Type 4 code clones. Various code clone detection approaches and models have been used to detect a code clone. However, a major challenge faced in detecting code clone using these models is the lack of generality in detecting all clone types. To address this problem, Generic Code Clone Detection (GCCD) model that consists of five processes which are Preprocessing, Transformation, Parameterization, Categorization and Match Detection process is proposed. Initially, a pre-processing process produces source units through the application of five combinatorial rules. This is followed by the transformation process to produce transformed source units based on the letter to number substitution concept. Next, a parameterization process produces parameters used in categorization and match detection process. Next, a categorization process groups the source units into pools. Finally, a match detection process uses a hybrid exact matching with Euclidean distance to detect the clones. Based on these processes, a prototype of the GCCD was developed using Netbeans 8.0. The model was compared with the Generic Pipeline Model (GPM). The comparisons showed that the GCCD was able to detect clone pairs of Type-1 until Type-4 while the GPM was able to detect clone pair for Type-1 only. Furthermore, the GCCD prototype was empirically tested with Bellons benchmark data and it was able to detect clones in Java applications with up to 203,000 line of codes. As a conclusion, the GCCD model is able to overcome the lack of generality in detecting all code clone types by detecting Type 1, Type 2, Type 3 and Type 4 clones.

1. Introduction

Java is a programming language that is used to develop software and it was released in 1995. It is an open source language that is widely used in Netbeans or Eclipse to develop software. The original intention of the Java language was intended to let application developers write once, run anywhere (WORA). This means that the compiled Java source codes can run on all platforms that support Java without the need for recompiling the code again [1]. A lot of applications are developed using Java language. This is due to it being open source [1] and has more development resource compared to other programming language. Therefore, this makes Java language is more preferable among developers in developing applications. An experiment was conducted to see the occurrence of code clones in Java applications. Several Java applications totaling up to 512 000 lines of codes used in this experiment [1]. A total of 6% of the 512 000 lines of codes or 30 720 lines of codes from the tested Java applications contains clones. The analysis of this experiment concluded that large and early year developed Java applications contains more code clones compared to the smaller and later years built Java applications. There are two reasons contributing to this situation [1]. The first is because of the intense pressure of

preparing the application for the stakeholders in limited time. The second reason is the absence of generic modules in Java in which would have prevented a lot of clones if existed. Therefore, it is essential that the code clone or also known as duplicated code is also an issue in Java applications.

Code clones can be detected through models. Three most notable code clone detection models available are Generic Clone Model [2], Generic Pipeline Model [3] and Unified Clone Model [4]. Generic Clone Model is a model that defines clone exists in a program [2]. The main use of this model is to describe clones. The model allows separation of concerns between clone detection, description and management using layers. This model has a clear separation of clone detection process definition using layers. Furthermore, the model is more focused on management of code clone that was driven by the operational aspects of code clon detection and removal. Generic Pipeline Model is a code clone detection model that has a combination of five processes to detect code clone. There are five processes involved in this generic pipeline model [3]. The first process is parsing process and it transforms source code into source units. The second process is pre-processing process and the purpose is to normalize the source units that was obtained from the previous process. The third process is pooling process and the purpose of this process is to group pre-processed source units into groups or pools based on user defined criteria. The fourth process is comparing process and the purpose of this process recursively compares source units in all pools using a divide and conquer strategy. The fifth process the filtering process and the purpose of this process is to remove irrelevant clone candidate sets from the result set. This process is utilized in removing non relevant candidate sets out of the result set. Unified Clone Model is an attempt in having a generic model that can represent all the results of all code clone tools [4]. It was designed through the different clone representations of existing tools. The outcome of the analysis has been divided into four groups which are detection for clone triage and management, integration of additional data from other sources, replication of scientific studies, and benchmarking of clone detection techniques.

Model is an attempt to have a unified process in detecting all code clone types. The attempt could be seen through the Unified Clone Model although this model is still in the design phase. The most advanced code clone detection model to date is the Generic Pipeline Model that detects on exact and near exact clones in Java applications. Exact clones refer to Type-1 clones while near exact clones refer to Type-2 clones. As a whole, Java is a programming language that is used to develop software. The absence of generic modules in Java causes the occurrence of code clone. In order to detect code clones that bring negative impacts to the software, code clone detection approaches and models have been proposed to detect the code clones. Models consist a combination of processes that pre-processes, transforms, detect and display code clone detection results. The models apply the existing code clone detection approaches as part of its process. Although code clone detection models are relatively new and few to have as a prototype, yet the existing code clone detection models have been a forward attempt in having a unified process that detects code clone regardless to the various code clone terminologies. Therefore, this research focuses on proposing a code clone detection model that detects all types of code clones in Java applications. Section 2 shows the proposed work while Section 3 shows the evaluation of the proposed work. Section 4 shows the findings while Section 5 concludes this paper.

2. Generic Code Clone Detection Model for Java Applications

Generic Code Clone Detection is a code clone detection model that is aimed to detecting all code clone types in Java applications. It consists of five processes which are pre-processing, transformation, parameterization, categorization and match detection process. The five combined processes form the proposed which is the GCCD. This model is aimed at detecting code clone Type 1, Type 2, Type 3 and Type 4. Figure 1 shows the high level view of Generic Code Clone Detection Model.

2.1 Pre-Processing Process

Source code refers to the codes that written in a source file of an application. There are five combined rules used to achieve the aim of this process. The first rule is PR-1: Remove package and import statements. The purpose of this rule is to remove package and imports in the source code. Package name

usually shows the name of the package or project while imports refer to the classes that are imported into the package or project. Although the imports are classes, these imports are just merely imported statements. Therefore, this rule is designed to remove the import statements and package names from the source file.

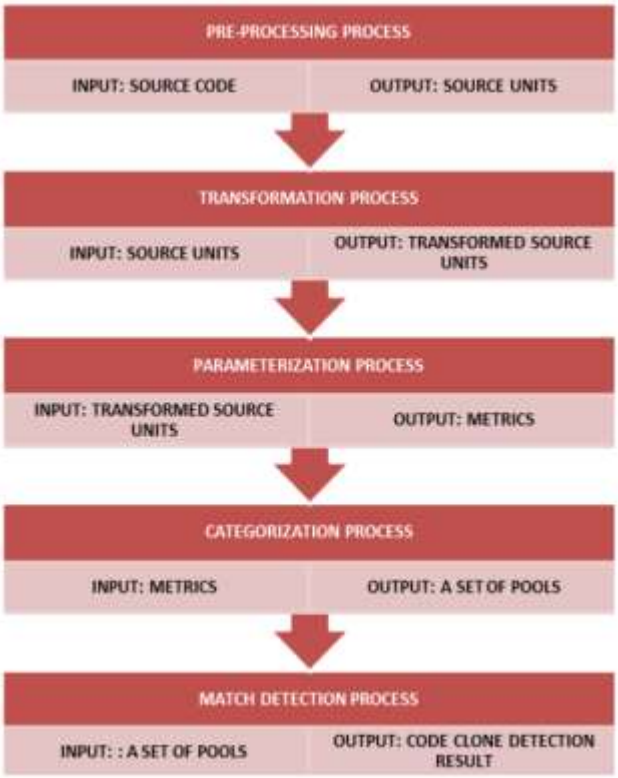

Figure 1. Generic Code Clone Detection Model

The second rule of this process is PR-2: Remove comments. Comment is an important component in programming as it serves as notes of a written source code for software developers and maintainers. Comments are usually in new lines or between source code lines and its occurrence is essential. Comment styles vary due programmers' style of programming and due to the programming language itself. Therefore, this rule attempts to remove comment lines occur in the source codes. The third rule of this process is PR-3: Remove empty lines. Empty lines refer to lines that have no content. The empty lines occur in most of the developed application and in nature happen due to programmers not cleaning up their code after developing or maintaining the code. Therefore, it is not harmful in removing these lines as it does not serve any importance in code clone detection result. The fourth rule in this process is PR-4: Regularize function access keyword to public. There are three function accesses in most of the programming languages which are public, private and protected. This rule regularizes all the function accesses into a single function access; which is public. Unlike CCFinder [5] that removes the function accessibility keywords completely, these try to preserve the code based on the function granularity by not removing these keywords but generalizing it to the same keyword. Furthermore, the keyword will serve as the constant value for ratio calculation. The fifth and final rule of this process is PR-5: Regularize source codes to lowercase. Source codes are written in different ways by programmers and therefore produce a different style of source code writing. Variables can be easily varied due to lowercase and uppercase letters; hence, causing a difference in the code clone detection results. As an example, the variable stringbuffer can be written as Stringbuffer, stringBuffer or even StringBuffer. Although it is the same variable, but code clone detection approach such as exact or string based approach will consider these variables as different variables hence affecting the code clone detection

result. Therefore, it is important to regularize the variable in the same manner. This rule regularizes these variables into lowercase form.

The output refers to the output produced by this process. The output of this process is normalized source codes or better known as source units. Source unit is still in the form of source code. Each source unit represents a function of the source code.

2.2 Transformation Process

This is the second process in the GCCD. The aim of this model is to transform the source units into measurable units or numbers. The measurable units are also known as transformed source units is then are used as the medium in determining the parameters for the next process. The input of this process is normalized source codes or better known as source units that were produced from the previous process. The source units are still in the form of source code and each source unit represents a function of the source code. This process uses a letter to number substitution concept to transform the source code into measurable units. The substitution is done based on the location of the alphabet. For instance, a is the first alphabet in vocabulary therefore it is substituted to 01. The same concept applies for other alphabets in the vocabulary.

The output of this process is transformed source unit are in numerical form. The transformed source units are then divided into two groups which are *header (h)* and body (b). Header refers to the start of a transformed source unit prior the body part of a transformed source unit while body is the body of a transformed source unit. As an example to elaborate the header and body of a function, assume a function called Function A with the content of:

public void setAbbrev (String abbrev) this.abbrev.setText (abbrev) ;

After going through the first process which is the pre-processing process, the source unit of Function A appear as:

public void setabbrev string abbrev thisabbrevsettext abbrev

Therefore, the header and body of a function of Function A:

header (h) : *public void setabbrev string abbrev* and body (b) : *thisabbrevsettext abbrev*

2.3 Parameterization Process

This is the third process in the GCCD. The aim of this process is to create parameters or metrics that will be used for categorization and match detection process. The input of this process is the transformed source units that are obtained from the previous process. As mentioned before, the transformed source units are in the form of numerical. The parameter that is used for detection is header average ratio and body average ratio. In order to obtain the header average ratio and body average ratio, four metrics need to be obtained from the transformed source units. Table 1 shows the metrics that are extracted from the transformed source units.

Table 1. Metric extracted from the transformed source units

Metrics	Description
header code count	Amount of code in header
body code count	Amount of code in body
header ratio	Ratio of header
body ratio	Ratio of body
average header ratio	Average ratio of header
average body ratio	Average ratio of body

In order to obtain an average ratio, the ratio must be obtained first. As mentioned in the previous process, the access function of all the function has been transformed into the public. Therefore, the transformed source unit contains the same value of access function after going through the transformation process. By using this as the benchmark value, each source unit is divided with this value to obtain the ratio value of each code in header and body of the source unit. As an example to elaborate the detail calculation of average ratio for each transformed source unit, lets assume a transformed source unit has header, *TSUha*, and body, *TSUhb* . Therefore the ratio of the transformed source unit is:

$$R_A = \frac{(A_1 A_2 A_3 \dots A_n)}{P_1} \quad (1)$$

$$R_B = \frac{(B_1 B_2 B_3 \dots B_n)}{P_1} \quad (2)$$

where;

P_1 is value of access function that starts with public
R_A is ratio for each transformed source unit in header
R_B is ratio for each transformed source unit in body
$A_1A_2\ A_3..An$ is value in transformed source units in header
$B_1B_2\ B_3..\ Bn$ is value in transformed source units in body

Once the ratio of header and body in each function has been obtained the next step is to calculate the average ratio of header and body in each source unit. The average ratio of header and body in each source unit is:

$$AVR_A = \frac{(R_A)}{C_A} \quad (3)$$

$$AVR_B = \frac{(R_B)}{C_B} \quad (4)$$

AVR_A is average ratio for transformed source unit in header AVR_B is average ratio for transformed source unit in body CA is code count for each transformed source unit in header CB is code count for each transformed source unit in body. The output of this process is the aforementioned metrics that will be used for in the next phase which are categorization process.

2.4 Categorization Process

This is the fourth process in the GCCD. The aim of this process is to categorize and group these source units into a group of clone based on the same average ratio value of header and body between functions. The input of this process is the set of metrics obtained from the previous process. The categorization is done based on the average ratio value of header and body between transformed source units. The transformed source units are grouped into three pools based on the same average ratio value of header and the same average ratio value of body between functions. As an example, if transformed source unit A has the same average ratio value of header with transformed source unit B, therefore these two transformed source units is grouped into the same group or better known as the first pool. This process will be continued until all the transformed source units that have the same average value of header is grouped together in the same pool. As for the remaining transformed source units, these transformed source units is then grouped in the same group based on the same average value of body or better known as the second pool. It is done until all the remaining functions have been grouped together. If there are

remaining functions after the categorization based these two groups, the remaining functions will be grouped into another group or better known as the third pool.

The output of this process is a set of pools that has been grouped based on the same average ratio value of header and the same average ratio value of body between transformed source units. The output of this process serves as the input of the final process which is the match detection process.

2.5 Match Detection Process

This is the fifth and final process in the GCCD. The aim of this process is to detect the code clone. The input of this process is the pairs and groups of source units based on three categories. The match detection process uses a hybrid detection technique of exact matching with Euclidean distance. As mentioned before, there are three pools obtained from the previous process. The match detection starts by finding the exact clone or better known as Type-1 clone; and near exact clones or better known as Type-2 clone. Therefore, there is two stage of exact matching being used to detect Type-1 and Type-2 clones. The first stage is the exact matching technique is used in detecting the same average ratio value of header and body between source units in the first pool. The compared functions that have the same average ratio value of header and body are detected as Type-1. The second stage is the exact matching technique is used in detecting the same average ratio header value but different body value and the same average ratio body value but different header value between functions in the first and second pool. The clones that are detected through this stage are known as Type-2. The remaining average ratio header and body value from the first and second pool is combined into the third pool for the next step of this process. As for the remaining average ratio header and body value, Euclidean distance is applied. Assume there are two source units which are A and B. Therefore, the Euclidean distance, ED, between A and B calculated as:

$$ED_{AB} = \sqrt{(headerA - headerB)^2 + (bodyA - bodyB)^2} \quad (5)$$

where;

ED_{AB} is Euclidean distance of Function A and Function B
headerA is the average ratio header of A
bodyA is the average ratio body of A
headerB is the average ratio header of B
bodyB is the average ratio body of B

The calculation of the Euclidean distance is applied to the remaining average ratio header and body values in the third pool. Once the calculation is done, it is the function is then grouped to Type-3 and Type-4 based on the distance obtained. Type-3 clone are taken from the range of 85% - 100% [7] while the remaining is defined as Type-4.

3. Evaluation

Table 2 shows the overall result of the detected clone pairs using the GCCD for Java applications in Bellons benchmark data (Bellon et al., 2007). There are total 7281 clone pairs of Type-1, Type-2, Type-3 and Type-4 detected in J2sdk1.4.0-javax-swing using the GCCD. There are 877 clone pairs or 12% of the total clone pairs detected for Type-1 while 3697 clone pairs or 50.8% of the total clone pairs detected for Type-2. As for Type-3, there are 1710 clone pairs or 23.5% of the total clone pairs detected and there are 997 clone pairs or 13.7% of the total clone pairs detected for Type-4. As for Eclipse-jdtcore, there are total 11268 clone pairs of Type-1, Type-2, Type-3 and Type-4 detected using the GCCD. There are 626 clone pairs or 5.6% of the total clone pairs detected for Type-1 while 2886 clone pairs or 25.6% of the total clone pairs detected for Type-2. As for Type-3, there are 4265 clone pairs or 37.9% of the total clone pairs detected and there are 3491 clone pairs or 30.9% of the total clone pairs detected for Type-4. There are total 2685 clone pairs of Type-1, Type-2, Type-3 and Type-4 detected using the GCCD in

Eclipse-ant. There are 185 clone pairs or 6.9% of the total clone pairs detected for Type-1 while 552 clone pairs or 20.5% of the total clone pairs detected for Type-2. As for Type-3, there are 581 clone pairs or 21.6% of the total clone pairs detected and there are 1370 clone pairs or 51% of the total clone pairs detected for Type-4. As for Netbeans-javadoc, there are total 595 clone pairs of Type-1, Type-2, Type-3 and Type-4 detected using the GCCD. There are 99 clone pairs or 16.6% of the total clone pairs detected for Type-1 while 341 clone pairs or 57.3% of the total clone pairs detected for Type-2. As for Type-3, there are 102 clone pairs or 17.1% of the total clone pairs detected and there are 53 clone pairs or 9% of the total clone pairs detected for Type-4.

Table 2. Detection result based Bellon benchmark data

Clone Type	Bellons Benchmark Data (Java)			
	J2sdk1.4.0-javax-swing	**Eclipse-jdtcore**	**Eclipse-ant**	**Netbeans-javadoc**
Type-1	877	626	185	99
Type-2	3697	2886	552	341
Type-3	1710	4265	581	102
Type-4	997	3491	1370	53
TOTAL	7281	11268	2688	595

4. Evaluation

The aim of producing a generic source unit representation is essential as it helps in processing and transforming the source codes in a more unified form that can be used in detecting code clone. The first three rules in the pre-processing process are essentially needed as it is the noise or uninformative section of the source code; thus does not harm the information or structure of the source codes. The next two rules are to regularize the source code in a form that is acceptable at the function level and reduce the dissimilarity due the lowercase and uppercase of the source codes. The second process in the Generic Code Clone Detection which is the transformation process is aimed to transform the source units into measurable units or numbers. The measurable units are also known as transformed source units act as an intermediate representation of the source code that will be used for clone detection purposes. By adopting the substitution cipher concept, the code to numerical transformation allows the application of the heuristic method to be applied to the source codes. Regarding on the runtime performance of these two processes, the combination of these two processes produces the highest runtime performance for large datasets compared to the small sized data sets. The third process which is the parameterization process is the process that produces the metrics that is used for categorization and match detection process. The aim of the categorization process is to categorize and group these source units into a group of clone based on the same average ratio value of header and body between functions. Apart from the source unit representation, the match detection technique plays an important role in detecting code clones accurately. The applied match detection approach in the fifth process which is the match detection process is a heuristic approach that uses elements from the source units itself in determining the clones. The combination ratio and distance is a combination of basic metrics but yet produces results in detecting for all the code clone types.

As seen in the empirical evaluation between the GCCD and Generic Pipeline Model, the different definition between the similarity pair and clone pair highly influences the code clone detection result. Similarity definition that is used in the Generic Pipeline Model; is a loose definition for clones as such that the computed pairs would not fit any common notion of a clone but could still be useful for a certain application of pair definition. This means a certain level of the allowance is given to near miss clones or non-clones to be claimed as clones. As for the GCCD that adopts clone pair definition, it is more rigid in having exact or near exact match pair of clones. Therefore, the final output which the code clone

detection differs due the different definition of the clone or similarity pairs definition. A different definition of clone pairs yields different code clone detection result. Therefore, a more loosened clone pair definition might influence the code clone detection result.

5. Conclusion

This work shows that having a generic model for code clone detection is possible with the combination of essential processes of detecting code clone. Possible future work that can improve the code clone detection process and performance can be done through improvement of the pre-processing process in supporting code clone detection in other structural and procedural programming language. There are various programming languages exist in the current programming world; therefore it is essential the existence of clones on the applications of these programming languages. With the enhancement of the rules in the pre-processing process, the GCCD can support code clone detection in other programming languages tool. For future enhancement, this research can be extended through application of Artificial Neural Network [8].

Acknowledgement

The authors are grateful to Universiti Malaysia Pahang (UMP) in partially supporting this work through UMP Internal Grant (Grant ID: RDU180308).

References

[1] Dagenais, M., Patenaude, J.-F. F., Merlo, E. and Lagu¨e, B. (2002). Advances in Software Engineering. pp. 95–110.
[2] Giesecke, S. (2006). Generic modelling of code clones. In Koschke, R., Merlo, E. and Walenstein, A. (Eds.) Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings, vol. 06301. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.
[3] Biegel, B. and Diehl, S. (2010). Highly Configurable and Extensible Code Clone Detection. In 2010 17th Working Conference on Reverse Engineering. 237–241.
[4] Kapser, C. J., Harder, J. and Baxter, I. (2012). A Common Conceptual Model for Clone Detection Results. In Proceedings of the 6th International Workshop on Software Clones. IWSC ’12. Piscataway, NJ, USA: IEEE Press, 72–73.
[5] Biegel, B. and Diehl, S. (2010). Highly Configurable and Extensible Code Clone Detection. In 2010 17th Working Conference on Reverse Engineering. 237–241.
[6] Kamiya, T., Kusumoto, S. and Inoue, K. (2002). CCFinder: a multilinguistic tokenbased code clone detection system for large scale source code. IEEE Transactions on Software Engineering. 28(7), 654–670.
[7] Kodhai, E. and Kanmani, S. (2014). Method-level code clone detection through LWH (Light Weight Hybrid) approach. *Journal of Software Engineering Research and Development*, 2(1), 1-29.
[8] Fakharudin A. S., Mohamad, M. A., Johan, M. U. (2009). Newspaper vendor sales prediction using artificial neural networks. *International Conference on Education Technology and Computer 2019*. 339-343.