Journal of Information Technology Education: Innovations in Practice **Volume 10, 2011**

Pattern of Plagiarism in Novice Students' Generated Programs: An Experimental Approach

Marzieh Ahmadzadeh
Faculty of Computer Engineering & IT, Shiraz University of Technology, Shiraz, Iran
ahmadzadeh@sutech.ac.ir

Elham Mahmoudabadi
Faculty of Computer Engineering & IT, Amirkabir University of Technology, Tehran, Iran
e_mahmoudabadi@yahoo.com

Farzad Khodadadi
Faculty of Computer Engineering, Sharif University of Technology, Tehran, Iran
fkhodadadi@ce.sharif.edu

Executive Summary

Anecdotal evidence shows that in computer programming courses plagiarism is a widespread problem. With the growing number of students in such courses, manual plagiarism detection is impractical. This requires instructors to use one of the many available plagiarism detection tools.

Prior to choosing one of such tools, a metric that assures the chosen plagiarism detection tool is effective for novice programmer plagiarism is required. It must be able to detect the difference between accidental similarities and plagiarized code.

In this research we looked for such metrics by an experimental approach that was carried out in three phases. In the first phase we learned the possible ways to plagiarize, which was done by interviewing students. Categorizing the known methods lead us to the most popular plagiarism approach that was applied by the students. Therefore an experiment was designed to simulate the real plagiarism situation. Data that was gathered in this phase were evaluated against real plagiarism data to ensure the results.

The result showed that object-orientation concepts were of little interest for students to plagiarize; instead converting basic structures and syntax were more popular. This means that a source code which is not original is less likely to be different in terms of inheritance structure, encapsulation, or polymorphism from its original source code. This implies that an effective plagiarism detection system that is used for undergraduate teaching purposes should be able to identify similar simple structures rather than focusing on some more advanced options of languages such as inheritance.

It should be noted that the conducted experiments included both freshmen and senior students and the programming

Editor: Zlatko Kovacic

language that was used for the experiment was Java, which is an object-oriented language. Therefore the results can be generalized to all computer science undergraduates who program using an object oriented language.

Keywords: Plagiarism, Similarity, Programming, Java, Object Orientation, Novice Students.

Introduction

There are anecdotal evidences that in computer science education, specifically in programming courses, some students plagiarize in their programming exercises. Students are able to transform a source code that they perhaps obtained from their peers, deliver it as their own, and hope that the instructors/teaching assistants will not discover whether or not the code is original. If the plagiarized code is not discovered by the instructor, it may be recognized after exams are marked. An instructor might find that there are a few students who have not received enough marks in their exams to pass the course but achieved more than enough in their weekly exercises. In some cases the difference between these two marks is significantly high that one cannot attribute the low exam marks to issues such as student's stress in the exam.

We see plagiarism in programming exercises as a major problem because a student who could plagiarize without getting caught can take other courses, for which the *programming* course was prerequisite, and this creates a chain of unsuccessfully passed (i.e., with a low mark) courses, which damages his/her and the university's future reputation.

Nevertheless it is a labor intensive work; instructors are required to investigate the received source codes to assure their originality in early stage of the course. In practice, however, this is not possible. The number of enrolled students is high and this does not allow one to be vigilant for each and every one of given exercises. Using a kind of automated countermeasure, therefore, seems inevitable. Fortunately, there are some plagiarism detection tools available either to be used online for free or purchased. To choose one of such tools, however, an instructor should bear in mind that the selected tool must explore real plagiarism not innocent similarities, which are common in programming. It is interesting to note that sometimes plagiarism detection systems surprise you by reporting similarity between two codes that you are sure were produced by two different students. Therefore, the question here is what kinds of metrics are available to distinguish between plagiarized codes and accidentally produced similar codes. This motivated us to explore the kinds of syntactic or semantic changes that are more likely to be applied by novice students in their programs in order to plagiarize. We could then use our metrics to evaluate plagiarism detection tools to find the most effective one; however, this part is out of scope of this paper. The natural way of researching these issues seemed to be an experimental approach that forms the main part of this research.

The structure of this paper is as follows. The next section of this paper reviews the literature, which motivated the current research. In the later section, we introduce the design and implementation of our experiment. This is followed by the analysis of the results. We finally discuss what might be done in the future and what will be the practical implication in last section.

Literature Review

There is a list of software that deal with the detection of the source code plagiarism (Ahtiainen, Surakka, & Rahikainen, 2006; Aiken, 2009; Daly & Horgan, 2005; Freire, Cebrian, & Rosal, 2009; Gitchell & Tran, 1999; Prechelt, Malpohl, & Phlippsen, 2002). Each one detects similarity with a different approach and strategy.

RoboProf (Daly & Horgan., 2005), a marking system, initializes the plagiarism detection process by adding a watermark to students' programs when they submit them. Later submitted programs are checked against the watermark and, if the watermark is recognized, a plagiarism report is issued. This tool therefore is language independent. Pair-wise comparison of the codes is another strategy that is used by other tools. This comparison is approached by either attribute counting or analyzing the underlying structure of the codes (Aiken, 2009; Prechelt et al., 2002).

Halstead metrics (Halstead, 1975) is the benchmark that is used mostly for attribute counting. This relies on the number of operands and operators recognized in the code. Investigating the structure of the program, however, is based on tokenizing the code (Aiken, 2009; Prechelt et al., 2002) and executing similarity measurement on tokens. Faidhi & Robinson (1987) however approached the issue differently and described seven levels of program plagiarism type, which include *original program*, *comment and indentation*, *identifier name, variable position*, *function combination*, *program statements* and *program control logic.*

A plagiarism detection tool, regardless of being developed based on attribute counting, tokenizing the code, or any other method of detection, at least should be able to cover all these levels defined above. These levels, however, exclude the possibility that object oriented attributes of a program can be a subject for plagiarism.

On the other hand, it should be noticed that finding a similarity in codes should not necessarily be accounted as a plagiarism because students use the same structure that instructors use when they teach a new subject. For example, you cannot expect students to use a completely different approach when they are reading data into an array as they have learned that this is better done by using the *for-loop* structure. One cannot accuse students of plagiarism since this similarity is completely innocent. Therefore, the question that arises here is what kinds of similarities are most likely to be plagiarism. What kinds of transformation students are more likely to make when they plagiarize? Is there a pattern of transformation, similar to what discovered by Faidhi and Robinson (1987), that covers object orientation plagiarism too? These are the kinds of questions that we will be answering in this research and that distinguish the current work from others.

There is little research, if any at all, regarding the comparison between the plagiarism detection tools we have discussed above; however, the evaluation of a plagiarism detection tool in isolation can be found (Stevens & Jamieson, 2002). Stevens and colleagues investigated the usefulness, accuracy, feedback, and quality of marking of EVE, which is a plagiarism detection tool for text homework, from students and staff point of view. This kind of research seems essential for plagiarism detection tools used for a programming course, which is out of scope of this research.

Research Design and Implementation

Our target population, to whom the results of this research apply, is the first year students who pass *principles of programming* and *advanced programming*. If syntactic and semantic changes that one applies in an original code to produce a second version of the code (i.e., plagiarized code) can be found, it establishes the metrics that we are looking for in this research. A natural way of finding these metrics is to compare an original and plagiarized source code. For this, we first needed to explore what type of plagiarism students are more interested in and where the original codes usually come from. Then we would be able to compare the two versions of the codes and find the similarities. Therefore three phases for this research were considered, which will be described in the following subsections. This research has been carried out entirely at the Faculty of Computing Engineering & Information Technology, Shiraz University of Technology, Shiraz, Iran.

Phase One

Our first aim was to find various ways that students might use to plagiarize a code. For this we arranged an interview with five senior students (i.e., fourth year), two males and three females, who were not in any way a stakeholder for the kinds of questions that we wanted to ask because they were no longer programming students and no one could blame them for what they may or may not have done. We explained the purpose of the research for them clearly and justified the questions that we asked. We made sure that every one of them understood the purpose of the study and that no one was going to place blame on them. Therefore, we relied on their answers. After conducting an informal interview, we were able to categorize dishonesties in submitting a programming assignment into four types.

The first type refers to a situation where a student receives a code from his/her peer and transforms the code to make it different from the original code. The second type concerns situations in which part(s) of the code (i.e., one specific method) is (are) received from peers. In this case that (those) received part(s) might be included in the code with or without any transformations. The third type, which was described by interviewees as the most popular one, relates to codes that were not written by the students themselves. Instead they ask senior students to do the exercise for them completely, perhaps for the sake of friendship or money. This happens frequently and the senior students might produce several versions of a code for several students. Type four applies to programs that are written by a professional programmer.

We excluded type four from our experiment for two reasons. First, it is less likely that students do that for every one of their weekly assignment because it will cost them a lot of money. Second, instructors or teaching assistants can identify this kind of code as an unusual program. We had observed cases in which students submitted a code that used some features of the language that had not been taught. Questioning the students for that professional part almost always showed that the code was not written by the students. In other words it was obvious that the code was written professionally and could not be done by a novice or an intermediate student.

At this point we decided to conduct our experiment considering type three only since we thought it can cover type one and two. To ensure that this conclusion is correct we evaluated our results with the codes that were available to us and had marked as a plagiarism in phase three of the experiment.

Phase Two

In this phase we asked two of our senior students (one girl and one boy) that had a history of producing code for several students to participate in our experiment. The experiment was conducted in a normal situation like when students ask our participants to write the code for them. Our participants were given a program specification (see Appendix) and asked to write codes as much as they normally do. The chosen assignment belonged to *advanced programming* in which object orientation concepts in Java was taught. To simulate what is happening in the process of plagiarism, one week time is given to them to submit the codes.

Since they previously had passed the same course, they were well aware that submitted codes will be checked for plagiarism. For this, we thought that they had become used to producing codes that are as varied as possible in order not to be caught by teaching staff, nevertheless we asked them to produce codes as they did it in a normal situation when students asked for it.

Participation in this experiment was entirely voluntary. Our volunteers were assured that we conduct this study for scientific purposes, no consequences exist for them, and no one is going to blame them for what they have done.

Twenty versions of the code for the same specification were gathered after the deadline. Therefore we needed to compare these 20 codes in order to find out if there are any patterns in the way that plagiarized code is generated.

The analysis of these codes was done qualitatively. For this, we looked at the patterns that emerged from the data. This gave us an idea of the kinds of pattern that we should look for. For example when inheritance hierarchy was changed, it gave us the idea of categorizing the codes based on inheritance structure.

The results gathered in this part are discussed in the next section.

Phase Three

After analyzing data that were gathered in phase two, it was time to evaluate our findings to real plagiarized code.

To run this phase, there was an ethical issue to consider. There were several programs that we had marked as plagiarism previously. On the other hand we did not want to place blame on a student for plagiarism whose code was innocently similar to others. Therefore we defined a strategy to select the codes. Our target codes must belong to students who have not received pass mark in their formal exams and received above 70 in their weekly exercise. We did not choose *pass mark* for weekly exercises, instead we chose 70, since we are aware that some students achieve less than what they deserve in their exam due to the exam stress. With this selection strategy we reduce the possibility of choosing an innocent similar code. At this point 10 codes from 10 different students were chosen.

The rest of the similar codes that were excluded belonged to students who proved to have programming ability in their exams; therefore, we did not regard them as a plagiarizer. With the assumption that plagiarizers are normally students who are not able to program and need exercises mark to pass the module, we could not regard these students as a plagiarizer and therefore their code was not investigated.

Analysis of Results

To investigate the codes, we bore in mind that the given program was written in Java and the concepts of encapsulation, inheritance, and polymorphism had been included in the program. Therefore, we investigated not only syntactic changes to the program but also were interested to see if object oriented concepts are of interest for plagiarism purpose.

Analyzing the codes that were generated by our participants revealed that both students had followed almost similar rules to transform a code to produce another version of the program. We were able to categorize the changes that were made by them in eleven taxonomies. As can be seen from Table 1, these patterns consist of both syntactic and semantic changes. This table shows the types of changes that were observed in the codes and the number of the codes that contained those changes.

The most common changes that were seen in all 20 source codes were variable rename, changing the content of comments, and transferring a method or variable definition to another point in the code which is harmless to the functionality of the code. These three are not surprising as it seems they are the first steps in generating plagiarized code. The first two are very common and easily recognizable. The third however needs more attention to be explored considering that the name of the method and its variables had been changed.

In second place with 90% of occurrence is a change in variable's scope. Some of local variables have became an instance variable and vice versa. The first change is not as troublesome as the second and is seen more. Changing the scope of variable from being an instance variable to being a local needed some logical changes and could not be done in isolation. Therefore changing the scope of local variable to instance variable was more popular and an insignificant number of codes contained the reverse transformation. Further, a few scope changes inside a method were observed. Variables, which had been defined in structures such as loops, conditional, and try-catch statements, were subject to scope alteration.

Table 1: Taxonomy of code similarity

Type	Number out of 20	Percentage
Renaming Variables	20	100
Changing Comments	20	100
Displacing Variables and Functions Definition	20	100
Variable Scope Change	18	90
Changing Method Signature	11	55
Adding Abstract Class or Interface	10	50
Structural Transformation	10	50
Substituting Classes with Alternative API Classes	10	50
Changing Instance Variables' Access Modifier	9	45
Separating Superclass and Interface in Separate Files	8	40
Changing Indentation	7	35

Changing the signature of the methods was a subject for more than half of the codes. Interestingly only two transformations formed this category. The first was to add/delete *access modifiers* and the second was adding a parameter to the methods where it was not really needed. Both of the above transformations resulted in minor changes of the code. For example, where a *static* method transformed to non-static, invoking the method required making an instance of the class containing the method. The use of *final* keyword for methods was seen excessively.

Half of the codes were included an *interface* to the hierarchy of the class or defined the root class as an *abstract*. All the codes in which this features was observed never took advantage of it later in the code.

Structural transformation was seen in half of the codes. The observed recoding was not remarkable and limited to re-coding *if-else* statements to *switch-case, for* loop to *while* loop, etc. Unlike what we anticipated, the number of the methods, their functionality, and the order of their calling was the same as the original code. Two codes out of 10 did not used polymorphism; instead an extra *vector* structure was defined to separate the storage of different objects. Therefore, all the related computation was done separately for each object.

In half of the codes alternative Java classes were used, where it was possible. For example *BufferedWriter, Vector, BufferedReader* and *String* replaced by *PrintWriter, ArrayList, Scanner* and *StringBuilder* respectively.

Less than half of the codes contained changing the instance variables' access modifier. For instance, the use of *protected* where *private* was used originally did not dysfunction the program and therefore was used in the code. Further to changing a variable's access modifier, use of *final* keyword for instance variable was prevalent.

Since this program consisted of several classes, 40% of the codes were separated in different .java file. In other words each class had been placed in different file. Only 35% of the codes had indentation changing.

As explained before, these were the results achieved from type three of the plagiarism. We believed that this experiment will cover the issues that would be seen in type one and two of plagiarism. Therefore, in phase three of the experiment we compared our result with the actual plagiarized code.

As it was explained before, we did not know what type of plagiarism (type one, two, or three) the selected code contained. Therefore, it seemed reasonable to compare the results achieved from phase two with the actual plagiarized code to estimate the accuracy of the results. As said before, we had ten real plagiarized codes in hand. We took every single finding from Table 1 and checked the codes against that. For example, one of the findings was *variable rename,* for which we looked at all the plagiarized codes and counted the number of variables in those ten codes. The total number of variables was 150. However not all of the 150 variables that originated from the original code had been renamed. Ten out of 150 of them had the same name, which mostly belonged to structures such as *for-loop* in which a single letter variable (i.e., i, j, k) is used. Therefore 140 of them were counted. All of these 140 variables had different names from the original codes' variables. Therefore the result was 93.33% (140/150). Of course there were several other variables that were not taken into account when total number was computed. These variables were the ones that had no equivalent in the original code as the structure of the fake codes were different from the original code. All other percentages in Table 2 were computed the same way.

It can be seen from Table 2 that more than half of the code contained renaming variables, changing comments, transforming the structure, substituting classes with alternative API class, and placing each class to separate file.

Table 2: The percentage of compatibility between results and real plagiarized code

Type	Percentage
Renaming Variables	93.33
Changing Comments	66.67
Displacing Variables and Functions Definition	46.67
Variable Scope Change	40.00
Changing Method Signature	40.00
Adding Abstract Class or Interface	20.00
Structural Transformation	53.33
Substituting Classes with Alternative API Classes	53.33
Changing Instance Variables' Access Modifier	26.67
Separating Superclass and Interface in Separate Files	66.67
Changing Indentation	33.33

Codes comprising indentation, method signature, variable scope change and variable displace and functions definition formed 33% to 40% of the codes.

Two unpopular modifications were adding abstract class or interface and changing instance variable's access modifier. Since any type of plagiarism might happen, this percentage is acceptable. We could carefully think that this percentage might belong to codes that were categorized in type three because the programmer was not a novice anymore.

Discussion, Future Work, and Practical Implication

The main aim of this research was to find the metrics with which an effective plagiarism detection tool can be evaluated. In this research, we were concerned about two different issues. First were the codes that were received from others, transformed, and submitted (plagiarized code), and the second issue was about similar codes that were not plagiarized. We then came across four types of plagiarism, excluded number four, and started with number three since we believed that this covers numbers one and two. We conducted an experiment and achieved results for type three and compared the results with real plagiarized code. We came to conclusion that the pattern of transformation remained almost the same. It should not be expected that the results in phase two and three match 100% since we are not aware of the type of plagiarism that has happened. Considering that intermediate and novice students are different in terms of program ability could justify these differences in the percentages.

Careful scrutinize of the data revealed that plagiarizing a code does not involve complex transformation and covers the six levels introduced by Faidhi and Robinson (1987). However it should be noticed that, in this experiment, participants were students themselves not professionals. In other words when current research result is compared with Faidhi and Robinson (1987) a direct relationship can be found between programming ability level and the complexity of generated plagiarized code. For instance, we observed that the number of codes that involved renaming variables were more than codes that contained variable scope transformation. While the first needs no skill, the second requires some programming skills. The fact that we chose programs that were complete and belonged to students who did not receive pass mark in exams (i.e., did not have program ability) made us trust the patterns that we found.

We observed that even students, who possess enough skills to program, are not able to produce a second working code that is significantly different from their original code in terms of programming logic. Of course we are talking about intermediate students who are able to program. We cannot generalize that for professional programmers. That is why the 20 codes that were generated by two of students were mostly different in terms of simple structure and concepts of programming.

We chose some plagiarized code while we were not aware what type of plagiarism group (type one, two, or three) these codes belong to. This enables us to generalize our results that the pattern that was found in this study can be used for any type of plagiarism (excluding number four).

The patterns explained in last section plus the comparison that was done in phase three, speak for themselves and guide us where to look when examining the possibility of plagiarism. What is not obvious in explanation of our taxonomy of similar code is that in all the programs the number of methods in each class remained the same as the original code. Ignoring the fact that some codes included an *interface* or *abstract class* in the program, which was not needed, shows that the number of classes remained the same as original code too. The total number of local variables and instance variables was the same since only the scope of the variable had changed.

Regarding object oriented programming, there was no transformation in inheritance hierarchy; however, some were interested in adding one level and defined an abstract class. This implies that a plagiarism detection tool that was not designed for object-oriented languages still can be effective for these types of codes. Since the logic remained the same and the inheritance tree did not change, there were not enough options for our participants to apply polymorphism more than what was in the original code; nonetheless, there were a few codes that looked less professional by not take advantage of polymorphism.

Some transformation in instance variables access modifiers was observed. Students were more interested in providing wider access for instance variables. For example *protected* access modifier was changed to *public.* This is not surprising because the reverse (i.e., putting *public* in place of *protected*) needed more coding that was not expected from those students that preferred plagiarism rather than writing their own code.

It should be noted that there were some limitations regarding this research that should be considered. First is the type of questions that we gave our students for the second phase of this study. If the specification of the program changes other patterns might be introduced, which we believe will contain our found pattern. Second is language features that are different from one language to another. Different languages may add some more items to our list.

What might be interesting for future work is how this pattern changes when programming ability improves. We are not sure if we will find the same pattern when we conduct the same experiment with the same participants in two years time when they are more experienced in programming. Further, to ensure whether or not the pattern that was found in phase two is independent of programmers' preferences, there is a potential for this phase to be replicated.

Also we believe that creativity may play an import role in programming. Therefore if different participants with different creativity are chosen, we might get slightly different results. In this study we gave our students a code to transform in the second phase. It might be interesting to change the design of the second phase of this study so that students themselves write the code from scratch and produce several versions out of that. The pattern can be found in this way and compared with our findings.

As a result, what we should expect from a plagiarism detection tool for this level of programming skills is the ability to distinguish the codes that are similar in terms of logic, whatever different in syntax. However the possibility of producing two perfectly similar codes by two innocent students remained unanswered. The findings of this research can be used to evaluate the plagiarism detection systems that are widely used or being developed.

References

Ahtiainen, A., Surakka, S., & Rahikainen , M. (2006). Plaggie: GNU-Licensed source code plagiarism detection engine for Java exercises. *Proceedings of the 6th Baltic Sea conference on Computing Education Research: Koli Calling*.

Aiken, A. (2009). *Moss (measure of software similarity) plagiarism detection system*. Retrieved January 10, 2010 from http://theory.stanford.edu/~aiken/moss/

Daly, C., & Horgan, J. (2005). Patterns of plagiarism. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA.

Faidhi, J. A. W., & Robinson, S. K. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education, 11*(1), 11-19.

Freire, M., Cebrian, M., & Rosal, E. D. (2009). *AC: An integrated source code plagiarism detection environment*. Retrieved January 15, 2010 from http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0703136

Gitchell, D., & Tran, N. (1999). Sim: A utility for detecting similarity in computer programs. *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*.

Halstead, M. H. (1975). Toward a theoretical basis for estimating programming effort. *Proceedings of the 1975 ACM Annual Conference.*

Prechelt, L., Malpohl, M., & Phlippsen, M. (2002). JPlag: Finding plagiarism among a set of programs with JPlag. *Journal of Universal Computer Science*, *8*, 1016-1038.

Stevens, K., & Jamieson, R. (2002). The introduction and assessment of three teaching tools (WebCT, Mindtrail, EVE) into a postgraduate course. *Journal of Information Technology Education*, *1*(4), 233-252. Retrieved from http://www.jite.org/documents/Vol1/v1n4p233-252.pdf

Appendix

The given program administers the process of admission in a hospital.

We have three groups of people in hospital including staff, doctors and patients. In this program you are dealing with doctors and patients only.

A doctor has a name, identification code and specialization. Doctor's identification code is a String of 5.

Information that you keep for a patient is name, file number, and his/her doctor identification code.

This hospital has 3 wards; Internal, Heart and Children with an assigned code of 10, 20 and 30 respectively.

Patient file number is a string of digit with length 9-11. The first two digits are the ward code that the patient has been admitted to. The next digit is a serial number. It means that you cannot admit more than 9 patients in each day. The rest of the digits show the date that the patient has arrived the hospital (4 digits for year, 1-2 digit for month and 1-2 digit for day). For example file number 108200823 shows that this patient is the 8th patient that was admitted in internal ward on third of February 2008.

When a doctor is added to the system, his/her name, ID and specialty should be entered by operator. If a new patient is admitted, his/her name and ward ID should be provided by operator.

This program keeps all the information in a file named person.txt. What this program does is to make a report from all patients and doctors of this hospital and writes them separately in two files. (e.g. patients.txt and doctors.txt)

Your job is to change the code in your style of writing. Make sure that your program functions correctly as it does now.

Biographies

Marzieh Ahmadzadeh is an Assistant Professor of Computer Science at Shiraz University of Technology. She received her PhD in Computer Science and MSc in Information Technology from the University of Nottingham, UK and her BSc in Software Engineering from Isfahan University, Iran. She teaches a variety of graduate and undergraduate courses and her research interest includes Computer Science Education in general and Computer Supported Learning, specifically Data Mining and Human Computer Interaction.

Elham Mahmoudabadi is a MSc student at Amirkabir University of Technology majoring in Computer Networks. She received her BSc in Information Technology Engineering from Shiraz University of Technology. Her research interest includes Programming, Data Mining and Computer Network.

Farzad Khodadadi is a MSc student at Sharif University of Technology majoring in Network Security. He received his BSc in Information Technology Engineering from Shiraz University of Technology. His research interest includes Grids Computation, Computer Networks, Machine Learning, Data Security and Programming.