Process Model Improvement for Source Code Plagiarism Detection in Student Programming Assignments

Kermek, Dragutin; Novak, Matija

Informatics in Education, 2016, Vol. 15, No. 1, 103–126

DOI: 10.15388/infedu.2016.06

Process Model Improvement for Source Code Plagiarism Detection in Student Programming Assignments

Dragutin KERMEK, Matija NOVAK
Faculty of Organization and Informatics, University of Zagreb
e-mail: {dkermek, matnovak}@foi.hr

Received: March 2015

Abstract. In programming courses there are various ways in which students attempt to cheat. The most commonly used method is copying source code from other students and making minimal changes in it, like renaming variable names. Several tools like Sherlock, JPlag and Moss have been devised to detect source code plagiarism. However, for larger student assignments and projects that involve a lot of source code files these tools are not so effective. Also, issues may occur when source code is given to students in class so they can copy it. In such cases these tools do not provide satisfying results and reports. In this study, we present an improved process model for plagiarism detection when multiple student files exist and allowed source code is present. In the research in this paper we use the Sherlock detection tool, although the presented process model can be combined with any plagiarism detection engine. The proposed model is tested on assignments in three courses in two subsequent academic years.

Keywords: plagiarism detection, source-code, process model, programming assignments.

1. Introduction

Source code plagiarism in programming classes is a big concern because in performing their assignments students often try to copy source code parts, or the entire code, from their colleagues, or find similar source code on the Internet and use it without noting its origin. A lot of literature is available on plagiarism detection in student programming assignments. Lancaster (Lancaster, 2003) gave a comprehensive overview of literature in the field of plagiarism. Mozgovoy (Mozgovoy, 2006) gave an overview of desktop tools for offline plagiarism detection in computer program. Culwin, MacLoad, and Lancaster (Culwin *et al.*, 2001), and Cosma and Joy (Cosma and Joy, 2006) revealed various plagiarism issues in higher education and concluded that, on the whole, plagiarism in the UK context is a prevalent and extensive phenomenon and that academics have different perspectives on what constitutes plagiarism. For instance, there is disagreement as to

whether the reuse of source code without acknowledgement can be considered plagiarism. In another interesting research about plagiarism from a student's perspective (Joy *et al.*, 2011) examples of source code were divided into categories depending on the type of plagiarism that was performed. In that study, students were asked to detect if a certain example of source code was plagiarism or not. For example, students correctly (with more than 90% accuracy) identified category "Copying from another student" as plagiarism. So the authors concluded that if those students do that, then they cheat deliberately.

Another paper, by Simon *et al.* (Simon *et al.*, 2014), also discussed the problem of what to consider as plagiarism in computing scenarios and in essay scenarios. They have developed 14 scenarios (cases) and asked students and academics to identify the scenarios that are considered plagiarism. Majority of the participants agreed that it was plagiarism: "to purchase the work of others and submit it as one's own, to incorporate the work of another student without their permission, and to include the work of other without referencing the source". Also, by diminishing majorities, the following scenarios where considered plagiarism/collusion: "borrowing and using another student's work, giving completed work to another students and asking them to improve it, and using work that had been written and submitted for previous assignments". They found out that there are "substantial differences between academic integrity issues in text based and computing assessments". In computing assessments it is more difficult to define plagiarism and collusion; and even if one chose to reference one's computing work, there are no standard guidelines for doing so. Another problem is that there is a gray area between plagiarism/collusion relating computer code and standard practices like using online tutorials. Authors concluded that "educating students about academic integrity policies it a necessary but insufficient strategy... Academic integrity standards should be accompanied by strategies to imbue students with an understanding and genuine commitment to these standards".

In object oriented programming the reuse of source code is encouraged and should not be seen as plagiarism. Cosma and Joy (Cosma and Joy, 2008) suggested that, if its reuse is allowed, the acknowledgement to the original author should be made. To identify the copied source code, today there are plagiarism detection engines which detect similarities in source code. Yang, Jiau, and Ssu (Yang *et al.*, 2014) cautioned that all source code similarities cannot necessarily be considered plagiarism because there are other causes that contribute to them. This article does not deal with those other causes directly, but it is easier to detect and exclude some of them as non-plagiarism manually. For example, a possible cause of false plagiarism detection is the usage of some design patterns.

The main focus of this article is detecting plagiarism that is mostly performed by copying. One of the goals of this article is to exclude reused source code whose use is allowed and get an overview report which shows only highly potential plagiarized assignments. The second goal is to optimize the detection of plagiarism in assignments with many source code files. The third goal is to create a detailed report that lists similar source code parts as potential plagiarism cases. This report is then used to identify cases in which design patterns are used and to manually decide whether plagiarism was actually performed in them. To fulfil all these goals, we propose a process model improvement for detecting plagiarism.

In Section 2 research goals are presented in more detail. Section 3 contains a short description of the Sherlock detection engine and the reasons why it was selected for our research. It is not the intention of the authors of this paper to improve any detection algorithm but rather use one of them in our process model. In Section 4 we propose a process model for plagiarism detection. Section 5 describes the validation of the model based on student assignments in three courses from two subsequent academic years. Finally, an outline of future work is given in Section 6, followed by the conclusion in Section 7.

2. Research Questions and Problem Description

From the academic point of view, the fact that some students cheat in their assignments represents a big problem. We can distinguish between two major issues in that respect: students copy parts or the whole source code from other students, or they copy the whole source code or parts of it from an unknown source and modify that source code minimally (for example, by changing variable names only). Besides, students may be given some source code parts in class that they may be encouraged to use in their assignments. Therefore every instance of copied source code does not necessarily represent cheating. The first question is: "Is it possible to identify source code whose reuse is allowed and exclude it from being detected as plagiarism?"

Đurić and Gašević (Đurić and Gašević, 2013) offered a five step approach (process model) to detect plagiarism that involves removing the reused source code. Drawing on their concept, our previous question can be extended in the following way: "Is it possible to identify source code whose reuse is allowed and exclude it from being detected as plagiarism in larger assignments with multiple source code files?"

If that is possible, then our second question is: "Is it possible to detect plagiarism among student assignments in that revised environment and create an overview report of highly potential plagiarized assignments?" To obtain this report, we propose a process model improvement for plagiarism detection.

For the purpose of model testing, we developed a Java based application that implements the proposed model and uses the Sherlock plagiarism detection system as a tool to detect similarities in two different areas. We also made a web application for viewing the reports based on different options and filters. The test cases comprise around 100 student assignments, with each assignment consisting of approximately 10 files of source code (though the number of files is not limited) and, possibly, some files that are not part of source code and should therefore not be included in detection.

3. Plagiarism Detection Engine

The proposed process model uses an existing plagiarism detection engine. A good definition of plagiarism detection engines was given by Lancaster (Lancaster and Culwin, 2005) : "Plagiarism detection engines are programs that compare documents with pos-

sible sources in order to identify similarity and so discover student submissions that might be plagiarised." In this article we are looking at plagiarism detection engines that were made to find plagiarism in source code files, which is different than finding plagiarism in text files.

There are many source code plagiarism detection engines, like JPlag (Prechelt *et al.*, 2000), MOSS (Schleimer *et al.*, 2003) and Sherlock (Joy and Luck, 1999). Lancester and Culwin (Lancaster and Culwin, 2004); (Lancaster and Culwin, 2005) provided a comparative overview of available plagiarism detection engines including the metrics that each of them uses. Plagiarism detection is often performed on different types of files for finding similarity between them. Sherlock, for example, creates multiple versions of original files for finding similarities that comprise: tokenized version (files are tokenized and then checked, a very popular and analyzed technique (Joy and Luck, 1999); (Prechelt *et al.*, 2002); (Mozgovoy *et al.*, 2005)), original version (similarity check is run on the original file), no white spaces (white spaces are removed before checking), normalized (normalizing files for Java or C++), no comments (all comments are removed from the file before checking), no comments and normalized, no white spaces and no comments, and comments only (this was not used).

Since our faculty uses Moodle LMS, JPlag and Moss were considered for usage at the start of our research. Several advantages of using a plagiarism detection engine within the Moodle system have been reported (Tresnawati *et al.*, 2012). For example, it directly provides information to teachers about existing similarities in the submission system for each submission.

However, the problem with MOSS and JPlag[1] is that source files are first uploaded to their web server, after which they perform plagiarism detection and return the report. Uploading files to a server might be a security issue (concerning, for example, the access to the data after their upload) and also depends on the Internet connection. Also, there is always a possibility that for some reason the service might shut down temporarily, or even permanently. According to our process model, each assignment must be compared with source code whose reuse is allowed so that those parts of source code are deleted from assignments. This step is repeated until no similarity is found. Such a procedure is likely to result in a large amount of data traffic and much slower processing.

Due to the aforementioned reasons, we prefer a detection engine that can be used locally. Another requirement is that the tool has a public license so that it can be used free of charge. The tool should also enable some kind of tokenized detection needed for detecting copied source code containing changed variable names. Sherlock fulfils all those prerequisites and its source code is under the GPL v2 license (University of Warwick, 2012).

Sherlock also provides documentation as evidence for plagiarism. However, the obtained results were not satisfactory for our needs. In the following sections we explain reasons for some of our decisions and present modifications we made in the process model for plagiarism detection.

[1] JPlag since March 2015 enables downloading source code and use it on local server.

4. Process Model for Plagiarism Detection

This section presents a process model improvement for plagiarism detection. A similar model presented in (Đurić and Gašević, 2013) has five steps:

- Pre-processing – making the detection robust to source code transformations like: removing blocks of comments, splitting/merging variable declarations, changing variable order, adding redundant statements, etc.
- Tokenization – making the detection robust to source code transformations like: language translation, renaming of variables, etc.
- Exclusion – creating and removing a tokenized version of the allowed source code
- Similarity measurement – detecting similarities on more types of algorithms
- Final similarity calculation – calculating overall similarity between two sources based on similarities obtained from different algorithms.

Our process model has a similar global structure but as we deal with multiple files in one assignment a merge step has been added and the exclusion phase split into two steps (5 and 6) which are repeated iteratively. Tokenization and pre-processing are already implemented in Sherlock, and they are executed every time similarity checking is performed (steps 7 and 10), but with different objectives.

The process model we propose has the following steps (Fig. 1):

1. Load all student assignments – loads all files from student assignments.
2. Exclude overhead files – deletes all unused files, leaving only files that should be checked.
3. Merge all files from one assignment – all assignment files of each student are merged into one big file that represents his/her assignment.
4. If allowed source code files exist, go to step 5, otherwise go to step 10.
5. Prepare one file with all the source code whose use is allowed for each group.
6. Choose allowed source code file – if more than one file is present (e.g. if two groups of files exist), it is checked which file among all files given to groups matches best the student assignment and this file is then used further
7. Compare each student assignment with the file chosen in step 6 – the plagiarism detection engine is run to find similarities between the student assignment and the allowed source code file.
8. Filter student assignment – delete similarities found in step 7 from the student assignment.
9. Iterate steps 7 and 8 until no similarity exists between the file in step 6 and the student assignment. This should be done for every student's assignment.
10. Run the plagiarism detection engine on all student assignments – finding similarities between refined student assignments.
11. Create overview report – with information from the assignments for which highest similarity was determined:
    a. Similarity that is higher than 20% between two student assignments, based on one type of similarity checking (tokenized, original, without white spaces…).

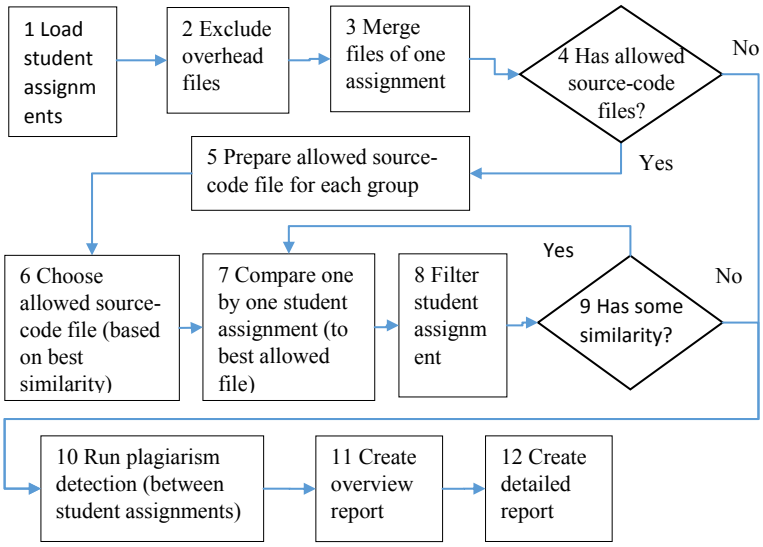


Fig. 1. Plagiarism detection process model.

b. Similarity that is between 10% and 20% based on four types of Sherlock similarity checking.

12. Create detailed report – report with concrete file names and source code blocks that are similar between two student assignments.

As we already mentioned, to test our model we developed a Java application. The rest of this section includes a detailed description of each step and explains how each of them was implemented.

4.1. *Load all Student Assignments*

The first step is simple and consists of loading all the assignments (with all the files included) that should be checked. In the implemented application all student assignments should be placed into one directory. Each student assignment is then placed under a separate subdirectory (further referred to as the student assignment directory) containing files and directories of the assignment. It is important that each student assignment directory has a unique name (e.g. student's username) to enable the differentiation of assignments in further steps.

At this stage Sherlock can be used to detect plagiarism although it is hard to find in a large set of data, as can be seen in Fig. 2, where points on the circle show the assignments and the lines between two points represent the similarity between these two assignments. The line color represents the strength of similarity between two files (red – highest, yellow – medium, black – lowest). A few smaller subsets of files were identified (Fig. 3) in which some files were similar owing to the allowed reused source code

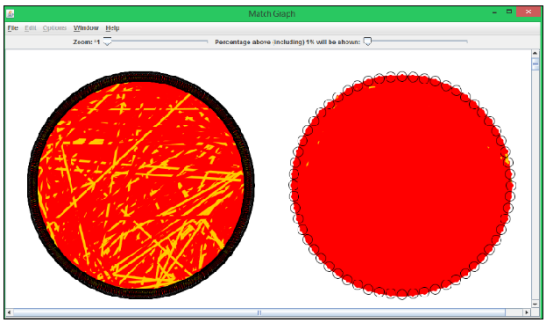


Fig. 2. Sherlock output – large amount of similarities.

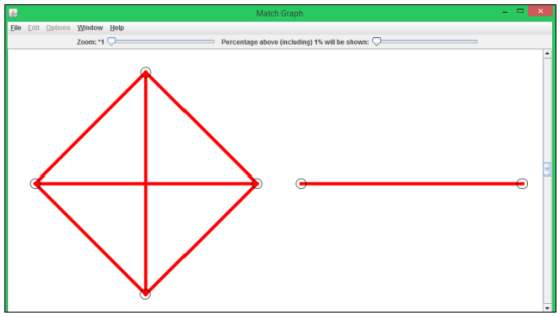


Fig. 3. Sherlock output – small amount of similarities.

or similarities between files inside one student assignment. However, it is not interesting to see the similarity between files in the same student assignment. Even when such similarity is not obtained, no conclusions can be based on a single file as we still do not know what amount of plagiarism exists in the whole assignment. These issues should be resolved in step 3.

Note that step 1 (Load all student assignments) can be preceded by another step, i.e. extraction (download) of assignments from the online submission system.

4.2. *Exclude Overhead Files*

Some similarities shown in Fig. 2 to Fig. 5 were identified owing to files that do not belong to source code, like log files. In this step all files that do not have a valid extension are deleted (e.g. jpg, xml) and all parts of source code that are not directly connected to the assignment (e.g. libraries and frameworks, like jQuery, Smarty, Twig, reCaptcha, etc.) are deleted too. The aim of this step is to exclude overhead (i.e. unimportant) files from being further used in plagiarism detection.

4.3. *Merge all Files of One Assignment*

The goal of this step is to merge all important files of one student assignment into one big file to reduce the number of files. These files are then directly checked with Sherlock which returns the information about the similarity of the whole assignment and not about each file in the assignment separately. The issue of similarity between files within the same assignment is also resolved in this way.

In this step all the files from one assignment are merged into one file. It is important to add the name of the original file (following the format: FILE_ {original file name}) before its content in the merged file in order to be able to identify which file the source code pertains to (Example 1). After this step is completed, Sherlock can be used to detect plagiarism.

```
FILE_file_name_1.java
package some_package_1;
import some_package.some_class;
…
FILE_file_name_2.java
package some_package_2;
import some_package.some_class;
…
```

Example 1. Added lines with file name.

Fig. 4 shows better results than those in Fig. 2. In Sherlock it is possible to change the minimum similarity (i.e. line between two points) that two assignments must demonstrate to be shown in the graph. Fig. 4 and Fig. 2. have a 1% similarity. If the minimum similarity is set at a higher value, for example 7%, the results are much better (Fig. 5).

A lot of similarities in this stage are still based on the allowed source code. Also, there is another issue to be considered here, for which it is important to understand different types of files for finding similarity that Sherlock works with. They were described

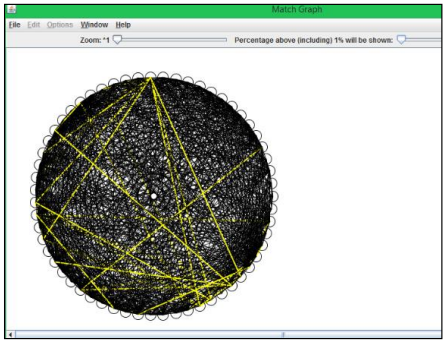


Fig. 4. Sherlock output –merged files.

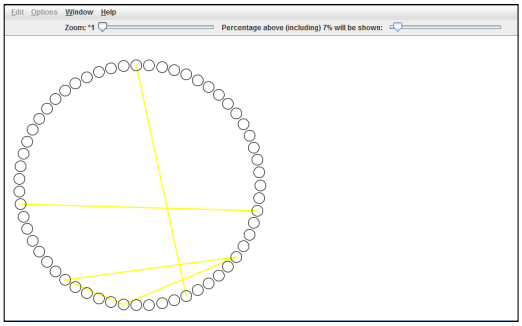
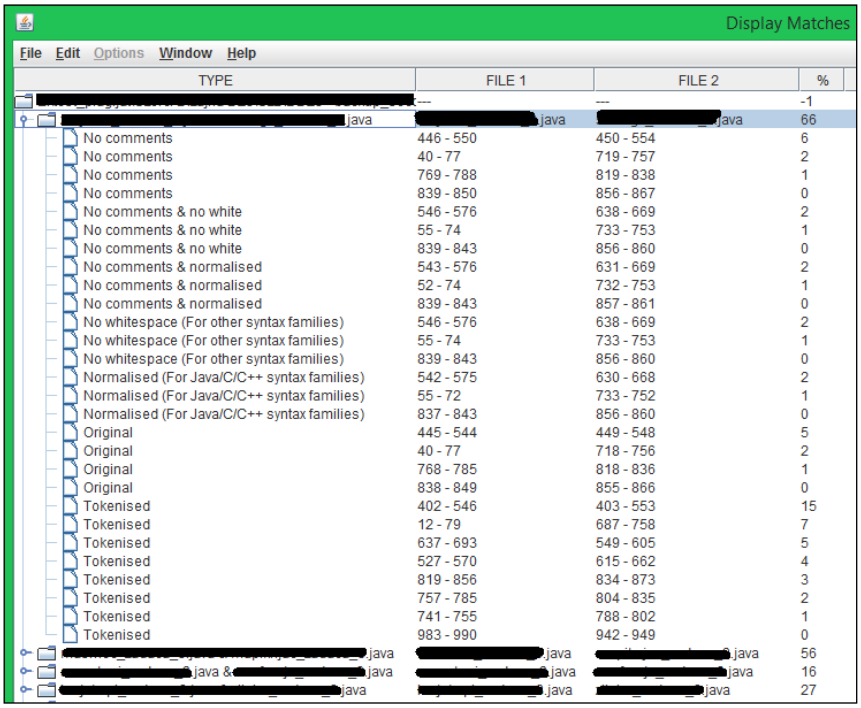


Fig. 5. Sherlock output – merged files with similarity above. 7%

at the beginning of section 3. For each of those similarity types Sherlock creates matched blocks which contain the percentage of similarity between the compared files.

The problem is that every matched block of the same type has its own similarity, but the similarity of the assignment is the sum of all the matched block similarities of the same type. Therefore it is not so easy to determine the similarity between assignments. A solution to this problem is provided in steps 10 and 11. Fig. 6 shows the "Display Matches" screen.



TYPE	FILE 1	FILE 2	%
[redacted]	---	---	-1
[redacted] java	[redacted] java	[redacted] java	66
No comments	446 - 550	450 - 554	6
No comments	40 - 77	719 - 757	2
No comments	769 - 788	819 - 838	1
No comments	839 - 850	856 - 867	0
No comments & no white	546 - 576	638 - 669	2
No comments & no white	55 - 74	733 - 753	1
No comments & no white	839 - 843	856 - 860	0
No comments & normalised	543 - 576	631 - 669	2
No comments & normalised	52 - 74	732 - 753	1
No comments & normalised	839 - 843	857 - 861	0
No whitespace (For other syntax families)	546 - 576	638 - 669	2
No whitespace (For other syntax families)	55 - 74	733 - 753	1
No whitespace (For other syntax families)	839 - 843	856 - 860	0
Normalised (For Java/C/C++ syntax families)	542 - 575	630 - 668	2
Normalised (For Java/C/C++ syntax families)	55 - 72	733 - 752	1
Normalised (For Java/C/C++ syntax families)	837 - 843	856 - 860	0
Original	445 - 544	449 - 548	5
Original	40 - 77	718 - 756	2
Original	768 - 785	818 - 836	1
Original	838 - 849	855 - 866	0
Tokenised	402 - 546	403 - 553	15
Tokenised	12 - 79	687 - 758	7
Tokenised	637 - 693	549 - 605	5
Tokenised	527 - 570	615 - 662	4
Tokenised	819 - 856	834 - 873	3
Tokenised	757 - 785	804 - 835	2
Tokenised	741 - 755	788 - 802	1
Tokenised	983 - 990	942 - 949	0
[redacted] java	[redacted] java	[redacted] java	56
[redacted] java & [redacted] java	[redacted] java	[redacted] java	16
[redacted] java	[redacted] java	[redacted] java	27

Fig. 6. Sherlock – Display matches.

4.4. *Has Allowed Source Code File?*

It is not necessary to have source code whose reuse is allowed. In such cases steps 5–9 are skipped. If allowed source code does exist, step 5 is included.

4.5. *Prepare allowed source code files for each group*

To resolve the problem of source code that can be reused, it is necessary to create one file which contains the source code that can be reused. This step can be done in parallel with the previous steps. This file must be created manually and usually represents a summarized copy of all source code that can be reused.

Allowed source code is given to students in lectures, as well as built during laboratory exercises. It is also necessary to mention that the amount of data that is put into the allowed source code file will influence the final results.

Due to the number of enrolled students in a course it is possible that laboratory exercises must be organized in two or more groups with the same or different teachers. Sometimes it is useful that each group has a different approach to some parts of the assignment since in that way students can suggest different variable names, data types, algorithms, etc. in the solution proposal debate. In such cases, a different file containing allowed source code is created for each group.

In the following subsection we explain why it is better to have a different source code file for each group than use the same source code file for all groups.

4.6. *Chose Allowed Source Code File*

In this step one allowed source code file (created in the previous step) is selected that has highest similarity to the student assignment based on the original type similarity detection. If only one group source code file is used, other groups will not have such a good match as the group from which the source code file is taken. Because of that, some allowed source code is not deleted, and in step 10 more similarities are found owing to the allowed reused source code.

It is possible to put all of the allowed source code from all groups into one file. However, in that case the file is quite large and contains a lot of source code that is very similar. Owing to that, it may not be easy to obtain such good matching results and similarity detection may take too long.

The third possibility is to use the "Exclude files" option in Sherlock. When this option is active, the files that contain allowed source code are given, and Sherlock uses them during similarity detection. If some source code from those files is found in assignment comparison, it is excluded from detection. The duration of similarity detection is a big problem. There is a possibility to modify the Sherlock's algorithm for this part, but since we were not able to find any technical documentation describing the source code, it was easier to do it outside of Sherlock. Also, we wanted to make

a model that we would be able to use with any detection engine in the future, not just Sherlock.

To avoid all the aforementioned issues, one allowed source code file was prepared for each group in the previous step. In this step, for each assignment the most suitable file (i.e. the one demonstrating best similarity to the original type) is chosen.

As a consequence, in most cases, each particular assignment will have best similarity with the allowed source code file of the group in which the student attends the course.

4.7. *Compare Student Assignments with Allowed Source Code*

After the file containing the source code whose reuse is allowed has been selected, we can proceed with comparison. There are two ways that were considered during the implementation:

- The first way is to run Sherlock on all assignments including the file with the allowed source code. However, this may take quite a while and may result in unnecessary matches.
- The second (preferred) way, is to run Sherlock on only one assignment with the allowed source code file and then go to step 8.

Note that Sherlock's configuration will influence the final results. For example, if a large number is put in the configuration field "max forward jump" (for example, 5), Sherlock will try to find similarity by jumping up to 5 lines forward. It is important to mention that lines that are skipped will also be deleted in further steps, so configuration parameters should be carefully chosen. It should be noted again that in this step another plagiarism detection tool could be used instead of Sherlock.

4.8. *Filter Student Assignment*

When matching which was described in the previous step is completed, filtering of files can be made. In the implemented application, matches (between the file with the allowed source code and the student assignment) found in this step are deleted from the student assignment. For example, if lines 2–5 in the student assignment are matched with some lines in the allowed source code file, these lines are simply deleted from the student assignment. During deletion, it is important to take care not to delete the line with the original file name (see step 2 and Example 1). Because of Sherlock's algorithm and its configuration it can happen that the title line gets inside a matched block, so we need to check if that has happened and accordingly exclude that line from deletion.

4.9. *Iterate Steps 7 and 8 until no Similarity Exists*

The aim of this step is only to check if some match exists between the file with the allowed source code and the student assignment. No similarity should exist between the

student assignment and the allowed source code. If no similarity is found, continue to step 9. Otherwise checking is performed again on the filtered files, so the procedure reverts to step 7. Steps 6 to 8 are repeated for each student assignment separately.

4.10. *Run Plagiarism Detection Engine on all Student Assignments*

This step can be applied when no similarities exist between particular student assignments and allowed source code files. In this step detection is made between all student assignments. Sherlock is used again in the implemented application. In the detection performed in this stage there is no problem with the reused source code (such as that presented in step 2) because we have removed the allowed source code in previous steps.

In the implemented application, match blocks generated by Sherlock are stored in a MySQL database to make the report creation easier. It stores id, file1 path, file2 path, match type, match type name (e.g. tokenized), start line in file 1, end line in file 1, line count in file 1 (start line in file 1 – end line in file 1), start line in file 2, end line in file 2, line count in file 2, similarity (given as percentage), username 1 (of the first student for whose assignment the match was found), username 2 and category name (e.g. homework assignment 1). Naturally, Sherlock can be used for data analysis, but the problem with similarity shown for every matched block that we explained in step 2 still remains (Fig. 6).

4.11. *Create Overview Report*

In this step, the analysis of the data stored in the database in the previous step is performed. After that, an overview report is generated (Appendix 1) and automatically sent via email to corresponding teachers.

In the example report (Appendix 1) from a total of 63 assignments we found 2 pairs of students (student1-student2, student3-student4) in which the minimum similarity that was obtained was larger than 20% (Appendix 1.A) on one type of similarity checking (for pair student1-student2 similarity was found only on the Tokenized type, and for pair student3-student4 similarities were found on all types of checking). Two different pairs had similarity between 10% and 15% (Appendix 1.B) on at least 4 types of similarity checking (for pair student5-student6 similarities were found on all types, and for pair student3-student4 they were found on all types except on the No Comment type). So we obtained 4 pairs that needed to be checked. It must be mentioned that these cases were not found when the assignments were checked manually, which had been done before using this application. When we used Sherlock without the proposed model only two pairs were found, while the other two pairs were found only by using Sherlock.

After this report has been generated and the pairs have been found, we can proceed with the manual checking using Sherlock. When we have identified the pairs, we can

simply use the "Display matches" option in Sherlock and examine only the pairs that have been found.

Some conclusions can be directly drawn from the report itself. For example, when we look at the first pair in the example (Appendix 1.A), we can see that the tokenized version has a 41% similarity. Here it is notable that only the tokenized matches should be further explored and, more importantly, that students copied the source code and changed it poorly while keeping the same structure. After looking at the source code in Sherlock using the "Display matches" option, we see that students only changed variable names.

While looking at the first pair, it may be concluded that source code parts have been directly copied because of large similarities that were found in all types of plagiarism detection. Based on such a conclusion only the original files should be further examined, which was actually the case. The same conclusion applies to Pairs 3 and 4 as it was confirmed that copying actually took place, only in smaller amounts than in the first pair.

This report is not optimal because some cases of plagiarism may have not been identified. Therefore we decided to implement a solution in which the parameters used in report creation can be changed.

Parameters that can be configured are:

- Minimum required similarity (of assignments to be included in the report). If similarity is set at a smaller value, more cases are included in the report. By default, the minimum similarity is set to 20%, which means that the assignment must yield at least a 20% similarity to be included in the report.
- Minimum line count (of assignments to be included in the report). By default, it is set to zero, and can be set at, for example, 100. It means that there should be at least 100 similar lines between two assignments. So if one assignment has a 20% similarity and only 20 similar lines, that case will not be included. This parameter is useful if we have an assignment that has very large similarity with the allowed reused source code, so a lot of source code is deleted. On the other hand, we may have an assignment that did not use the allowed reused source code, has 100 similar lines and yields a 20% similarity. It is easy to determine that the second case is our target. We want only such cases, while other cases are not our concern. Naturally, the result also depends on what kind of assignments are given to students. This is likely to occur in smaller assignments where students only need to add a few lines to some previously provided source code. It is also possible that a student will make a whole new solution which is not based on the allowed source code, and another student will simply copy parts of his/her solution.

To make the process creation easier, we built a PHP application which generates overview reports from the data stored in MySQL database and enables easy changing of parameters.

Although this report is itself a great help, Sherlock still must be opened for files to be found and compared, which is time-consuming. To speed up the process, another report is created in step 12.

4.12. *Create Detailed Report*

In this step a detailed report is created on each pair found in the previous step. This report contains source code snippets that are similar as well as the files in which those source code snippets were found. By 'file' we mean the original file name in the unmerged student assignment. A small part of the report is presented in Appendix 2. This report eliminates the need to open Sherlock (which can still be optionally used). Using the original file name, the teacher can directly go to the original assignment files (uploaded by a student) and examine them in detail. Code snippets from the report are helpful in finding this source code part in the original file. The main purpose of source code snippets is that they can be examined directly to see if plagiarism was performed or not.

The idea of this report is not to replace reading the assignment for grading purposes. Likewise, it is by no means meant to accuse the student of plagiarism based on the evidence in the report. It reveals only the assignment pairs with potential plagiarism, which makes it possible to examine these two assignments one after another. It also provides the teacher with a hint about which file in the assignment should be examined.

5. Process Model Validation

Process model validation is based on the previously mentioned Java based application and 20 assignments from three courses from academic years 2012/2013 and 2013/2014. Course descriptions are as follows:

- Advanced Web Technologies and Services (AWTS) – academic year 2012/2013 – 61 students – Java programming language – 3 assignments
- Advanced Web Technologies and Services (AWTS) – academic year 2013/2014 – 65 students – Java programming language – 4 assignments
- Web Design and Programming (WDP) – academic year 2012/2013 – 91 students – PHP programming language – 6 assignments
- Web Design and Programming (WDP) – academic year 2013/2014 – 85 students – PHP programming language – 4 assignments
- Design Patterns (DP) – academic year 2013/2014 – 60 students – 3 assignments – Java and C# programming language – no reused source code is allowed in this course.

All detections and tests were performed on HP Proliant ML 350G5 LFF server with following hardware and software specifications: 2 processors Intel Xeon Quad core E5410 2,33 GHz, RAM 2*512MB + 4 GB FBD, 4 x HP HDD300GB 15K SAS 3,5", Debian GNU/Linux 8.0 (jessie), Linux kernel 2.6.32-5-amd64, OpenJDK 64-Bit Server VM (build 24.75-b04, mixed mode), Apache/2.4.10, PHP 5.6.5, MySQL 5.5.42. Development environment was NetBeans 8.0.2.

In previous sections we already explained why the deletion of the allowed source code was done and why merging of files was necessary. We also established that it is best

to have one allowed source code file for each group and do the matching based on the file that suits each assignment best.

In the case of AWTS assignment 2 we attempted to perform detection without any deletion of reused source code. The test resulted in 134 pairs, with a similarity larger than 20% and 562 similarities between 10% and 20%. This was too much to analyze. A quick look at the detailed report showed that most similarities were actually attributable to the allowed source code. Those similarities would be even larger but after 9 hours of detection the test was stopped because the tokenized detection was not finished.

To improve the results, the allowed source code was added to the process (but only the source code that was written in one laboratory group) and deleted from student assignments. The results were similar to those in the previous test and random examination of the results showed that a lot of allowed source code was still used. The entire source code from all groups was then merged into one large file. The following problems emerged in the implementation of that solution: (1) detection was very slow because of the size of the single file that was used (~65KB per group (3) = 200KB); (2) deletion of allowed source code in a single assignment took a long time (~1h); (3) matching was bad because of too much similar source code in that one file.

The final solution was to separate each group source code into one file. This approach yielded 24 pairs with similarity larger than 20% and 14 pairs with similarities between 9% and 21%.

The duration of the detection was also satisfying (~24h). The results in the report showed that 21% similarity in this case is not good, because some allowed source code was still included. To solve this, the parameter for minimum similarity was raised to 25%. The new results were 8 pairs with similarity larger than 25% and 1 pair with similarity between 15% and 26%. By checking the results manually we established that these were the actual cases of plagiarism.

In the case of WDP assignment, 4 initial results were 16 pairs with similarity larger than 21% and 4 pairs with similarity between 9% and 10%. Raising the similarity parameter to 25% resulted in 8 pairs with similarity larger than 25% and 5 pairs with similarity between 15% and 26%.

However, in the case of AWTS assignment 4, first results were 457 pairs with similarity larger than 20% and 85 pairs with similarities between 9% and 21%. Raising the parameter for similarity did not help because a very high level of similarity with the allowed source code was found. In the end, the student file contained mostly file names of the original source code files (which had not been deleted to later enable the recognition of files in which similarity would be detected). Raising the similarity in this case resulted only in obtaining more files that contained code that had mostly been deleted. The solution was to add the absolute number of lines which the match must contain in the final report. In this case the number of lines was set to 150, so it resulted in 7 pairs when similarity is larger than 20%. For cases in which similarity is between 9% and 21% the number of lines was set to 50 results, which resulted in 4 pairs.

Every test case is further described in the following tables. Table 1 shows individual test results and settings (the tuning parameters that helped us to minimize the number of

Table 1
Overview of found plagiarism results

Course – Academic Year	Groups	Delete	Language	Assignment	Assignments	Similarity	
						>20%	9–21%
DP – 2013–2014	1	No	C#, Java	2	55	2	2
DP – 2013–2014	1	No	C#, Java	3	60	4	0
DP – 2013–2014	1	No	C#, Java	4	56	0	0
AWTS– 2012–2013	4	Yes	Java, xhtml	1	61	2	0
AWTS– 2012–2013	4	Yes	Java, xhtml	2	59	24 (>25% =8)	14 (15–26%=1)
AWTS– 2012–2013	4	Yes	Java, xhtml	4	61	457 (>20% & lines>150=7)	85 (9–21% & lines>50=4)
WDP– 2012–2013	6	No	HTML	1	63	30 (> 90% = 8)	6 (80–90%=1)
WDP– 2012–2013	6	Yes	HTML, JS, CSS	2	91	4	3
WDP– 2012–2013	6	No	HTML, JS, CSS	3	83	1	2
WDP– 2012–2013	6	yes	HTML, PHP, JS	4	90	16 (>25% = 8)	4 (15–26%=4)
WDP– 2012–2013	6	Yes	HTML, PHP	5	91	151 (>28% & lines>100=13)	6 (9–29%=10)
WDP– 2012–2013	6	Yes	HTML, PHP	6	77	7	1
AWTS 2013–2014	3	Yes	Java	1	55	0	1
AWTS 2013–2014	3	Yes	Java	2	47	6	8
AWTS 2013–2014	3	Yes	Java	3	49	10 (lines>50=4)	4 (lines>50=0)
AWTS 2013–2014	3	Yes	Java	4	50	297 (>60% & lines>70=2)	10 (15–61% & lines>40=3)
WDP–2013–2014	6	Yes	HTML, CSS	1	85	1	0
WDP–2013–2014	6	Yes	JS	3	85	4	8
WDP–2013–2014	6	Yes	PHP	4	85	2	0
WDP–2013–2014	6	Yes	PHP	5	85	26 (>40%& lines>100=1)	7 (25–41% & lines>100=6)

Table 2
Overview between found and real plagiarism academic year 2013/2014

Course /Academic Year	Found potential plagiarism (total)		Real potential plagiarism
	First run	After tuning	
AWTS 2013–2014	1	NA	1
AWTS 2013–2014	14	NA	3
AWTS 2013–2014	14	4	0
AWTS 2013–2014	307	5	2
WDP 2013–2014	1	NA	1
WDP 2013–2014	12	NA	3
WDP 2013–2014	2	NA	2 + 2 found manually
WDP 2013–2014	33	7	0

results are given in brackets in the right-hand column). Table 2 gives an overview of how much potential plagiarism was found in the academic year 2013/2014, and how much of it can actually be considered plagiarism. To be able to change the minimum similarity or the number of lines, it was necessary to have an appropriate interface. Sending a report by email was not the best option. A small PHP web application was therefore built in which the user can set the similarity and the number of lines. New queries can subsequently be performed and a new report is generated right away.

In the end we discuss another special case. This is WDP assignment 1, where only one HTML file was submitted by each student. In this case very large similarity existed between the allowed source code and the student file, so after deletion only a few lines were left in the student file. Moreover, it was hard to get results by raising the similarity. Although raising the number of lines did help (in correspondence with the case mentioned in the description of minimum line count parameter in section 4.11.) in this case it was more practical to simply detect plagiarism without the allowed source code file and set the minimum similarity above 90%.

In (Đurić and Gašević, 2013) authors showed that the removal of the allowed source code is possible and necessary, which has also been confirmed in this paper. Also we showed that merging of files provides a better overview of similarity and better detection.

The following problems arose during the test:

- It was necessary to manually clean all student folders from overhead files like JavaScript libraries (e.g. jQuery) or "old" folders and similar content that was found in some students' folders.
- Sometimes students did not use their username in naming their folders and instead labelled them as "mine" (or in a similar way). This must be corrected to know which student a particular file refers to.
- The manual selection of minimum similarity and minimum number of lines depends of the teacher's judgment and therefore cannot be the best solution.
- In case in which a large amount of assignments (about 90) exists and files are not small (>70KB) and, in addition, similarity with the allowed source code file is not high, the final tokenized detection takes long (up to a few days). It is therefore necessary to change the detection setting for the tokenized version, which has an impact on the quality of the findings and results in lower similarity that will be identified.

In running these tests we noticed that the tokenized version is the slowest. To optimize the process, it was modified in a way that detection in step 10 is performed on all types of plagiarism detection except for the tokenized type. After that the process continues to steps 11 and 12, and then reverts to step 10 and performs the plagiarism detection only on the tokenized type. When this is finished, steps 11 and 12 are repeated. In this way the largest plagiarism detection is quickly found (in approximately 3 hours) so the tokenized version can take longer, after which the information about other plagiarism that was found (if any) can be additionally reported.

6. Comparison with other Systems

In Section 3 was already mentioned that there are other systems for detecting plagiarism. Several tools address also the problem of "allowed source code" like: MOSS, JPlag, Plaggie and SCSDS. In this section a comparisons with MOSS, JPlag and Plaggie is given. SCSDS was not available, so no comparison could be done. In (Hage *et al.*, 2014) MOSS, JPlag, Marble, SIM and Plaggie were compared and the authors found that: "tools are sensitive to numerous small changes, all tools do well for the majority of single refactoring, but many tools score rather badly when refactorings are combined, worse than what may be obtained from simply using diff, a striking result of the top-10 comparison is that the top-10's for JPlag, Marble and MOSS are fairly similar, whereas the top-10's of Plaggie and SIM differ quite a lot from the other three".

The comparison of the tools was performed on the same datasets that were mentioned previously. To have a controlled comparison six tests were made T1 to T6 (details in Table 3).

All six tests had one plagiarized pair. The tests were:

- T1 was a PHP assignment.
- T2 was a PHP assignment.
- T3 was a HTML and CSS assignment.
- T4 was Java assignment.
- T5 was Java assignment
- T6 was JavaScript assignment.

Since all three MOSS, JPlag and Plaggie are token based, for all detections minimal token match was set to nine. JPlag and MOSS support text detections that are not tokenized. All tools did a good job on finding plagiarism in Java based assignments. JPlag and Plaggie could not be made plagiarism detection on assignments that included HTML, CSS, PHP or JavaScript. The reason is that tokenized algorithm for those languages does not exist. Text detection was tried with JPlag on such assignments but it was also not

Table 3

Similarity detection of compared systems for done tests

Test	Real plagiarism (manually estimation)	JPlag	Plaggie	Moss	Proposed system based on Sherlock
T1	~25%	N/A	N/A	T: 23%	T: 21%
T2	~25%	N/A	N/A	T: 23%	T: 21%
T3	~20%	N/A	N/A	T: 8% Tx: 5%	N: 31% T: 7%
T4	~50%	T: 71.9%	T: 76,5%	T: 51%	T: 45%
T5	~35%	T: 60.3%	T: 66,8%	T: 34%	NC: 32% T: 16%
T6	~25%	N/A	N/A	T: 25%	NCNW: 23% T: 10%

Legend: T – Tokenized, N – Normalised, NC – NoComment, Tx – text version,
NCNW – NoCommentNo White Spaces, N/A – not avalible

possible. The nice advantage of the proposed system is that it can compare on several versions of source code files including tokenized. This way if tokenized version fails the other text based comparison can be useful.

In tests T4 and T5 JPlag and Plaggie gave 20% greater similarity. This happens because of the inclusion of similarity with allowed source-code. In our opinion this is not good because allowed code is given by the teacher and students can/should have those parts similar. And this can't be considered as plagiarism and should not increase the similarity. Their reports show allowed source-code parts with the rest of code without any visual marking. So teacher must spent more time finding those parts to exclude them from other parts. JPlag performs detection only on files in root folder even the project has files in subfolders which could be a significant problem if the project has for instance Java package structure.

Problem with MOSS is that it is online service as already explained in section 3. MOSS also has not so good detection in T3 when HTML and CSS are used. This is a problem in cases when PHP or Java web applications are built where HTML and CSS are normal parts of the project.

The benefits of the presented system is that it supports all languages. Findings in tokenized version are best for Java since tokens are made for Java. But regardless there are no tokens for PHP the tokenized version works just fine. In cases where other language than Java is used a text based comparison is often more useful but this is not always the case. The system gives at once multiple detection results including the tokenized version. This way if tokenized detection fails other detection can show that the plagiarism is present like in T3. The system is offline which resolves the security issues that exist with online tools. Another benefit of the presented system is that it stores all results into database. This makes easier to review the results and create different reports. The plus of the system that it enables multiple groups of allowed source-code as explained in section 4.5. This is not possible in any of other tools. This is good when distinctions between lab groups exists and the system will find the best matching of allowed source code.

7. Future Work

To be able to have a completely automated solution, an extra step should be added to the process: connecting to the online submission system (like Moodle LMS) and starting the whole plagiarism detection process directly from such a system. Since we found it was more important to have a reliable tool for reporting plagiarism and test the process model, this step was left out although it had been originally planned. Instead we decided to perform this first step manually and not integrate it right away with the online submission system. Our plan is to add this step in future versions and to combine more plagiarism detection engines to additionally verify the results.

In our future work we could extend the model to detect which programming language was used and to detect plagiarism with an algorithm specially designed for some programming language as well as to see whether better results will be obtained in that way.

Another problem that we have not dealt with so far would be to find out if some students have taken parts of source code from other students. Our task would be to determine how many different assignments a certain assignment is similar to, and what those similarities are.

8. Conclusion

Plagiarism in student assignments remains a big problem, but the presented process model makes its detection better and faster. The process model has twelve steps. It can be used regardless of whether the allowed source code is present or not. It can also be combined with any programming language as well as with different plagiarisms detection tools. Currently we are using Sherlock but other tools like JPlag or Moss can be used instead. When tools like Moss are applied, it is important to take into account that an Internet connection is necessary and that files need to be uploaded to a server, which can present a problem in some cases.

To implement the proposed process model, we built an application that we tested with assignments from three courses from two subsequent academic years. The results show that the proposed process model is valid. With the implemented application, plagiarism was detected in assignments that had previously not been found by manual detection.

In some cases plagiarism can be detected even if there is none, so the results obtained by means of an application should not be taken for granted and should be additionally confirmed manually. The tool only provides hints to identify most suspicious cases. Parameters like minimum similarity play an important role in how many pairs we will be matched in the end. Because of that our second tool (the PHP web application) is particularly helpful for adjusting different parameters in order to reduce or extend the number of pairs. It can also happen that some plagiarism is not found. The tool is therefore not perfect, but is of great help in finding plagiarism.

Finally, all tools compared in section 6 are good and each has its benefits and downsides. It is proven that the proposed system works and it has some new benefits over existing tools. As mentioned in the previous section, there are some areas that can be improved in the proposed system, like detecting the programming language used in the assignment and using different plagiarism detection tools that are specialized for such a programming language.

References

Cosma, G., Joy, M. (2006). Source-code plagiarism: a UK academic perspective. In: *The 7th Annual Conference of the HEA Network for Information and Computer Sciences*. HEA Network for Information and Computer Sciences.

Cosma, G., Joy, M. (2008). Towards a definition of source-code plagiarism. *IEEE Transactions on Education,* 51(2), 195–200.

Culwin, F., MacLeod, A., Lancaster , T. (2001). *Source Code Plagiarism in UK HE Computing Schools, Issues, Attitudes and Tools*. London: South Bank University.

Đurić, Z., and Gašević, D. (2013). A source code similarity system for plagiarism detection. *The Computer Journal,* 56(1), 70–86.

Hage, J., Rademaker, P., and Vugt, N. (2014). *A Comparison of Plagiarism Detection Tools*. Utrecht, The Netherlands: Department of Information and Computing Sciences, Utrecht University. Retrieved 07 04, 2015, from `http://www.cs.uu.nl/research/techreps/repo/CS-2010/2010-015.pdf`

Joy, M., Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on Education,* 42(2), 129–133.

Joy, M., Cosma, G., Yau, J. Y.-K., and Sinclair, J. (2011). Source code plagiarism – a student perspective. *IEEE Transactions on Education,* 54(1), 125–132.

Lancaster, T. (2003). *Effective and Efficient Plagiarism Detection, PhD Thesis*. London, United Kingdom: South Bank University – School of Computing, Information Systems and Mathematics. Retrieved April 3, 2014, from `http://www.academia.edu/168972/Effective_and_Efficient_Plagiarism_Detection`

Lancaster, T., Culwin, F. (2004). Using freely available tools to produce a partially automated plagiarism. In: *Proc. of the 21st ASCILITE Conference*. Perth, Australia, 520–529.

Lancaster, T., Culwin, F. (2005). Classifications of plagiarism detection engines. *Innovation in Teaching and Learning in Information and Computer Sciences,* 4(2). Retrieved April 3, 2014, from `http://journals.heacademy.ac.uk/doi/pdf/10.11120/ital.2005.04020006`

Mozgovoy, M. (2006). Desktop tools for offline plagiarism detection in computer programs. *Informatics in Education,* 5(1), 97–112.

Mozgovoy, M., Fredriksson, K., White, D., Joy, M., Sutien, E. (2005). Fast plagiarism detection system. In: *SPIRE'05*. Buenos Aires, Argentina, 267–270.

Prechelt, L., Malpohl, G., Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science,* 8(11), 1016–1038.

Prechelt, L., Malpohl, G., Phlippsen, M. (2000). *Finding Plagiarisms Among a Set of Programs*. Karlsruhe, Germany: Universität Karlsruhe, Fakultültät für Informatik. Retrieved Dec 15, 2013, from http://page.mi.fu-berlin.de/~prechelt/Biblio/jplagTR.pdf

Schleimer, S., Wilkerson, D. S., Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, US: ACM, 76–85.

Simon, Cook, B., Sheard, J., Carbone, A., Johnson, C. (2014). Academic integrity perceptions regarding computing assessments and essays. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. New York, NY, USA: ACM, 107–114.

Tresnawati, D., Syaichu, A.R., Kuspriyanto. (2012). Plagiarism detection system design for programming assignment in virtual classroom based on moodle. *Procedia – Social and Behavioral Sciences,* 67, 114–122.

University of Warwick, D. o. (2012, August 25). *Warwick – Sherlock*. Retrieved April 3, 2014, from `http://www2.warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock`

Yang, F.P., Jiau, H.C., Ssu, K.F. (2014). Beyond plagiarism: an active learning method to analyze causes behind code-similarity. *Computers & Education,* 70, 161–172.

Appendix 1 – Example of Overview Report

Appendix A

Assignments with similarity larger than 20% at least on one type of similarity checking Table below has similarities for all types of similarity checking for found assignments.

Username1	Username2	File Type	File Type Name	Line Count	Similarity (%)
student1	student2	0	Original	159.0	11.0
student1	student2	1	Normalized	39.0	3.0
student1	student2	2	No Whitespaces	36.0	3.0
student1	student2	3	No Comment	123.0	8.0
student1	student2	4	No Comment Normalized	38.0	3.0
student1	student2	5	No Comment No Whitespaces	16.0	1.0
student1	student2	8	Tokenized	417.0	41.0
student3	student4	0	Original	306.0	34.0
student3	student4	1	Normalized	301.0	38.0
student3	student4	2	No Whitespaces	316.0	41.0
student3	student4	3	No Comment	198.0	25.0
student3	student4	4	No Comment Normalized	211.0	33.0
student3	student4	5	No Comment No Whitespaces	240.0	39.0
student3	student4	8	Tokenized	266.0	32.0

Appendix B

Assignments with similarity between 10% and 20% at least on 4 type of similarity checking Table below has similarities for all types of similarity checking for found assignments.

Username 1	Username 2	File Type	File Type Name	Line Count	Similarity
student5	student6	0	Original	185.0	10.0
student5	student6	1	Normalized	131.0	10.0
student5	student6	2	No Whitespaces	134.0	10.0
student5	student6	3	No Comment	178.0	10.0
student5	student6	4	No Comment Normalized	190.0	15.0
student5	student6	5	No Comment No Whitespaces	178.0	15.0
student5	student6	8	Tokenized	166.0	10.0
student7	student8	0	Original	98.0	10.0
student7	student8	1	Normalized	107.0	11.0
student7	student8	2	No Whitespaces	109.0	11.0
student7	student8	3	No Comment	77.0	8.0
student7	student8	4	No Comment Normalized	71.0	11.0
student7	student8	5	No Comment No Whitespaces	71.0	11.0
student7	student8	8	Tokenized	139.0	12.0

Appendix 2 – Detailed Report – Partly

Detail report – part

Pair	Org File1	Org File2	Code File1	Code File2	Line File1 Count	Line File2 Count	Similarity	File Type Name
…	…	…	…	…	…	…	…	…
student1-student2	File5.java	File6.java	…. @Override public void registerObserver (Observer observer) { this. observersEfficient.add (observer); } @Override public void removeObserverEfficient (Observer observer) { this. observersEfficient .remove (observer); } { …	… @Override public void registerObserver (ObserverEfficient observer) { this.observersEfficient.add (observer); } @Override public void removeObserverEfficient (ObserverEfficient observer) { this.observeriEfficient .remove (observer); } …	60	59	4	Original
…	…	…	…	…	…	…	…	…
student1-student2	File1.java	File2.java	catch (IOException e) { System.out.println(e); } return clubs; }	catch (IOException e) { System.out.println(e); } return read; }	4	4	0	No Comment Normalized
…	…	…	…	…	…	…	…	…

student1-student2	File3.java	File4.java	<IDT>.<IDT>(<IDT>)
			}
			}
			}<CATCH>(<IDT><IDT>){
			<IDT>.<IDT>.<IDT>(<IDT>)
			}
			<RETURN><IDT>
			}
			<VALUE>
			
			<MODIFIER><MODIFIER>
			<TYPE>
			<IDT>(){
				<IDT>.<IDT>(<IDT>)
				}
				}
				}<CATCH>(<IDT><IDT>){
				<IDT>.<IDT>.<IDT>(<IDT>)
				}
				<RETURN><IDT>
				}
				<VALUE>
				
				<MODIFIER><MODIFIER>
				<TYPE>
				<IDT>(){
					10	10	0	Tokenized

…	…	…	….	…	…	…	…	…

D. Kermek received 1999. his Ph.D. in Information Sciences from University of Zagreb, Croatia. From 1986. to 1993. he worked as a programmer, software architect, and project leader at the Center for informatics at the University of Zagreb Faculty of organization and informatics in Varaždin. He served at University of Zagreb Faculty of Organization and Informatics as Vice dean for Academic affairs in 3 consecutive terms from academic year 2005/2006 to 2010/2011. Currently, he is a Full Professor at the Department of Theoretical and Applied Foundations of Information Sciences at the University of Zagreb Faculty of Organization and Informatics. He teaches following courses: Web design and programming, Advanced Web technologies and services, Design patterns, E-learning systems.

M. Novak studied at the Faculty of Organization and Informatics in Varaždin at University of Zagreb. He gained Master degree "Master of Informatics" in 2010. After finishing master's degree he worked for two years in NTH Group in Varaždin as Product manager for voice platform and Business Consultant for mobile applications for Swiss and German territory where he gained experience on international projects. After that, he worked for one year at MCS d.o.o as system architect for mobile and web platforms. From November 2013 he is a PhD student at Faculty of Organization and Informatics and is working as a teaching assistant at the same faculty where he is teaching courses Web design and programming, Building a Web application and Advanced Web technologies and services. His fields of interest are plagiarism detection, software engineering, data warehousing and data extraction, transformation and loading (short ETL).