Journal of Information Technology Education: Innovations in Practice **Volume 12, 2013**

Presenting an Alternative Source Code Plagiarism Detection Framework for Improving the Teaching and Learning of Programming

Frederik Hattingh, Albertus A. K. Buitendag, and Jacobus S. van der Walt
Tshwane University of Technology
Pretoria, Tshwane, South Africa

hattinghFG@tut.ac.za buitendagAAK@tut.ac.za
vanderwaltJS@tut.ac.za

Abstract

The transfer and teaching of programming and programming related skills has become, increasingly difficult on an undergraduate level over the past years. This is partially due to the number of programming languages available as well as access to readily available source code over the Web. Source code plagiarism is common practice amongst many undergraduate students. This practice has a detrimental effect on the presentation of specific content relating to introduction to programming courses. One of the problems identified in the research conducted is that turnaround time with relation to assessment and feedback, which are presented to the students, is a critical factor in the subsequent success rates of the subject.

This paper investigates, utilizing a literature review, how plagiarism detection metrics and a framework for providing effective feedback to students and educators could be implemented to enhance the teaching and learning processes.

The predominant technique used for detecting plagiarism is to evaluate how a piece of source code was constructed over time. By analyzing the students' programming patterns, lectures can be adapted to address problem areas and react accordingly. The paper also provides an overview of current metrics used for plagiarism detection and suggests ways of improving the process by including enhanced techniques for the gathering of metrics over time as well as suggesting ways to use the metrics to aid learning on all cognitive levels.

Some of the key considerations presented as part of this research include effective feedback mechanisms and real-time responses to plagiarism as well as contributing towards learning on different cognitive levels.

Keywords: Blooms digital taxonomy, Plagiarism detection, Teaching Programming, Teaching methodologies.

Editor: Peter Blakey

Introduction, Background Problem and Prior Research

The positive effects of the digital revolution are offset by the negative impact it is having on academic institutions. As the digital age is evolving, thereby increasing student access to information, it is becoming more difficult for academic institutions to maintain academic integrity across instructional programs.

This paper utilize a combination of a literature review and the creation of a framework that aims at addressing some of the core issues identified as part of the literature survey, as well as reflections and insights provided by academia and colleges.

Zobel (2004, p. 147) emphasizes the fact that a literature study should contribute text in providing the reader with a better understanding of the elements of the study as well as the topic researched. He explained that in an ideal research document the literature study should be as interesting and thorough as the description of the paper's contribution.

Some of the benefits of a literature study, as listed by Leedy and Ormond (2004, p. 70), will greatly contribute to the understanding and implementation of the research objectives, the relevant benefits are:

- The capability of such a study to reveal approaches followed by other researchers in the same area or field.
- Show and introduce some relevant measurement tools developed in previous studies of a similar nature.
- Revealing methods of dealing with problem situations that may be similar in nature to the current research study.

As can be derived from the benefits, a literature study will play a key role in this study to be done, both from an informative and active learning point of view.

Plagiarism, as defined by Merriam Webster ("Plagiarism," 2012), is "the act of using another person's words or ideas without giving credit to that person." The threat of plagiarism is not only limited to academic writing, but also includes source code that is written as part of the learning process.

In addition to verbatim copying of assignments between students, a programming assignment may also be considered plagiarized if the code was converted directly from another programming language, if code is reused between assignments (self plagiarizing), if students collaborate extensively when writing code, or when other people are paid to write code (Joy, Cosma, Yau, & Sinclair, 2011).

There are numerous examples of websites and services that host searchable code which is accessible by the public, e.g., question and answer sites that provide ready-made solutions to programming problems and websites where, for a relatively small fee, a programmer can be hired to complete a task. This provides enough resources to tempt a student into plagiarizing part of, or an entire, assignment. The Internet should not carry all the blame for the prevalence of student plagiarism (Cosma & Joy, 2008).

A study done by Lim and See (2001) involving data collected from three educational institutions in Singapore showed that about 94% of students admitted allowing their own work to be copied by other students. A similar study in Australia involving first year students found that those participating in the study thought that it was acceptable to collaborate on assignments that were meant to be completed individually (Sheard, Dick, Markham, Macdonald, & Walsh, 2002).

Current Source Code Plagiarism Detection and Prevention Techniques

Methods for dealing with the problem of plagiarism can be classified based on approach followed, i.e., 'proactive' methods for preventing plagiarism from taking place and 'reactive' methods of detecting plagiarism after work has been completed and submitted (Lukashenko, Graudina, & Grundspenkis, 2007).

Recent proactive methods used by academic institutions are educating students on plagiarism, creating clear anti-plagiarism policies across different academic programs, and adopting honor codes (Devlin, 2006; Olt, 2002; Park, 2003).

Reactive methods used for plagiarism detection in source code are widely considered to be a pattern-matching problem which produces a number of metrics. The metrics can then be analyzed to determine how much of the source code was copied between different documents in the corpus that is being evaluated (Jones, 2001). Lancaster and Culwin (2005, p. 4) define a 'metric' as a rule that can convert a document into a numeric value for representing similarity.

Traditional Detection Approaches

sA common approach to plagiarism detection in source code relies on parsing the source code contained in the document and then generating token strings. Using an algorithm, the token strings generated by this approach are, then, compared to other token strings. The Sim utility is an example of the approach. It uses string alignment techniques and algorithms originally developed to detect similarity between DNA strings (Gitchell & Tran, 1999). An alternative method also relies on tokenization, but determines similarity by analyzing the structure of the source code.

Whale (1990) has argued that better results can be achieved by analyzing source code structure, and has supported these arguments by developing a utility called 'Plague'. Wise (1992) identified some problems with Plague, but supports the notion of analyzing structure similarities as opposed to text similarities.

The deterrent posed by effective plagiarism detection engines provides the only link between proactive and reactive methods. Implementing proactive methods to prevent plagiarism may require more time to implement, but it may produce a positive effect in the longer term (Lukashenko et al., 2007). Howard (2002) notes how the amount of effort used in detecting academic plagiarism may result in students seeing the educator as the enemy instead of the mentor. Blindly using detection tools gives no insight into the student's reasons for plagiarizing. The researchers believe that the notion of Howard (2002) opens the door to try and find ways in which the detection practice could also improve the teaching and learning process, and not just act as a deterrent.

It would seem that proactive and reactive methods for dealing with plagiarism are not well aligned. Reactive methods provide instant results that show whether a piece of source code was plagiarized or not. The researchers, however, stress that, in the long term, this method may not add much value to the academic process leading to students focusing on ways to defeat the engine. In contrast, the proactive approach may provide positive results in the long term. It should be noted, that the proactive methods still do not produce immediate results that can be used to verify originality of code.

Figure 1 indicates the traditional workflow of how a student may complete and submit a programming assignment. In this traditional workflow, only the final document, or documents, (containing the code) are submitted to the detection engine. The engine then analyses the documents in the corpus and feedback is provided to the educator for further analysis. The student is notified

after the educator has interpreted the results on whether the submission is considered plagiarized on or not.

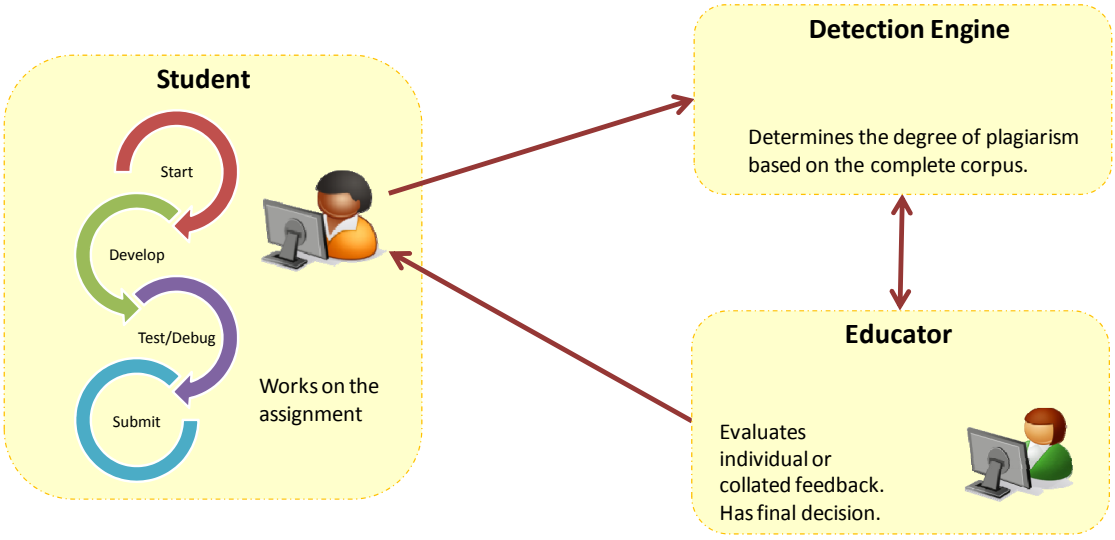

Figure 1: Workflow followed in traditional detection engines

One problem identified with the workflow described above is that the history of how the student created the piece of source code is lost. Students code and develop software in different ways. Due to the nature of programming, and the inclusion of various code constructs and units in the development of a solution, the authors believe that it is important to track and monitor the complete development process rather than just the final product. Because there is no time-line of how the source code was constructed, any direct evidence that the student used to try to conceal plagiarism is also lost.

There are various reasons for students plagiarizing a piece of source code including:

- Lack of technical knowledge
- Self-plagiarising or plagiarising commonly repeated functionality
- Poor time management
- Academic pressure

Power (2009) concluded that students often don't understand what is considered plagiarism. According to Voelker, Love, and Pentina. (2012) a similarity exists regarding how both graduate and undergraduate students understand plagiarism. This suggests that students' lack of understanding about plagiarism is not strongly correlated to a certain level of education. Voelker et al. (2012) go on to mention the way in which many students think plagiarism can be avoided by citation and reference alone.

The problem of plagiarism due to a lack of understanding can be interpreted differently when the task is to write a piece of source code. When writing source code as opposed to writing academic work, there is no generally acceptable rules regarding citing and reference.

The task is to solve a specific problem writing a number of statements in a logical sequence using the syntax of a specific programming language. The lack of understanding in this case may be an inability to use this specific language and its syntax to solve the problem at hand. The student may not have sufficient knowledge of the language or syntax to solve the given problem in the first place. As a result and sometimes a last effort students may revert to plagiarism.

A programming problem can also be solved using code by breaking the problem up into a number of small sub-problems. When unable to solve one of these smaller problems, a student may revert to plagiarism while still being able to solve other sub-problems. In the process of combining the smaller problems, the student may learn what the purpose of the plagiarized code is and how the code works. A student may also use a piece of code that was previously written and well understood to solve a sub-problem.

Students may feel they have not been given enough time to complete an assignment or may generally procrastinate leading to time pressures (Power, 2009). Koul, Clariana, Jitgarun, and Songsriwittaya (2009) concluded that performance oriented students are more likely to plagiarize. This may be due to pressure to from society, family or educators to obtain good grades (Devlin & Gray, 2007).

Two other factors that might influence the decision to plagiarize are the consequences of getting caught and how uniformly plagiarism detection techniques are enforced across different subjects (Miller, Shoptaugh, & Wooldridge, 2011; Power, 2009). Other possible reasons for plagiarizing include personal and cultural attitudes towards plagiarism and the desire to test the system (Wan, Md Nordin, Halib, & Ghazali, 2011).

Using traditional detection engines gives no indication of the student's motives or underlying academic reason for plagiarizing. Another academic reason which could motivate a student to plagiarize is the fact that the student had a difficult time in interpreting and understanding a lecture presented by an educator, based on a certain topic. Language barriers could also have an impact on the student's motivation. The metrics used to detect the plagiarism are of no further use once it has been determined that a student has plagiarized.

Defining a new set of metrics that provide indications of both that a piece of source code has been plagiarized and why its creator plagiarized may be beneficial to both the student and educator.

Plagiarism Detection Engine Based On New Metrics

To determine when source code plagiarism occurred, as well as the possible reason(s) for the plagiarism, a detection engine needs to be developed that may track the source code being written by students in real time. The engine may then produce the required metrics to guide the educator in identifying plagiarism while adding value to the academic process.

Evens and Peck (2006) have suggested that the use of light weight analysis may enhance teaching software engineering. By introducing the concepts in a pilot course to test the assumptions made by the researchers, students were asked to record the time spent on each programming assignment. Jones (2001) attempted to use physical metrics – namely number of lines, words, and characters. Further detection relied on the source code, the compilation log, and the execution log.

It should be noted that both of the above approaches only consider source code after submission. In addition, no detection of plagiarism is attempted as the code is being written. Metrics to identify plagiarism of source code, as it is being written include:

- Time spent writing code for the assignment.
- Number of modifications, including the text that was modified each time.
- Length of each modification that was made.

Tracking the time that a student spent writing code for the assignment may be the first indication of possible plagiarism. This metric may also give a unique insight on the student's time management skills as well as insights on how assignments are completed, and in addition, may indicate the possibility of the student having plagiarized some or all of the code in the assignment. This

may be evident especially if there is a big discrepancy between how long the educator expects the student to work on the assignment and the actual time to completion.

Tracking the number of modifications over time may point to possible plagiarism or plagiarism avoidance. This is especially true if the student adds a large number or lines to the code base and then proceeds to make a number of small changes over time. Tracking the text that was inserted or deleted can aid the educator in determining whether the motive for the number of changes was to avoid plagiarism detection or to integrate code that was previously written to solve a similar problem. If data is collected from multiple students with multiple attempts, patterns could be identified and used by the educator to design or to modify future lectures.

Finally, the length of each modification can be used to detect both plagiarism and plagiarism avoidance. When using these metrics, the problem of detecting extra-corpal plagiarism – like sources from the web and textbooks – is largely made irrelevant by the fact that the engine does not need these source documents when analyzing the corpus instead making a deduction based on the metrics mentioned above.

Figure 2 shows an updated workflow, which may support the effort to gather the new metrics identified. This workflow consists of the Code Snapshot Service (CSS), Notification Service (NS), and the Plagiarism Detection Tool (PDT). The student may work on the programming assignment while the CSS takes regular snapshots of the code that is being written. Each snapshot may contain the code for the assignment as it was during the particular time that the snapshot was taken. These snapshots are delivered to the PDT that continuously analyses new snapshots as they arrive.

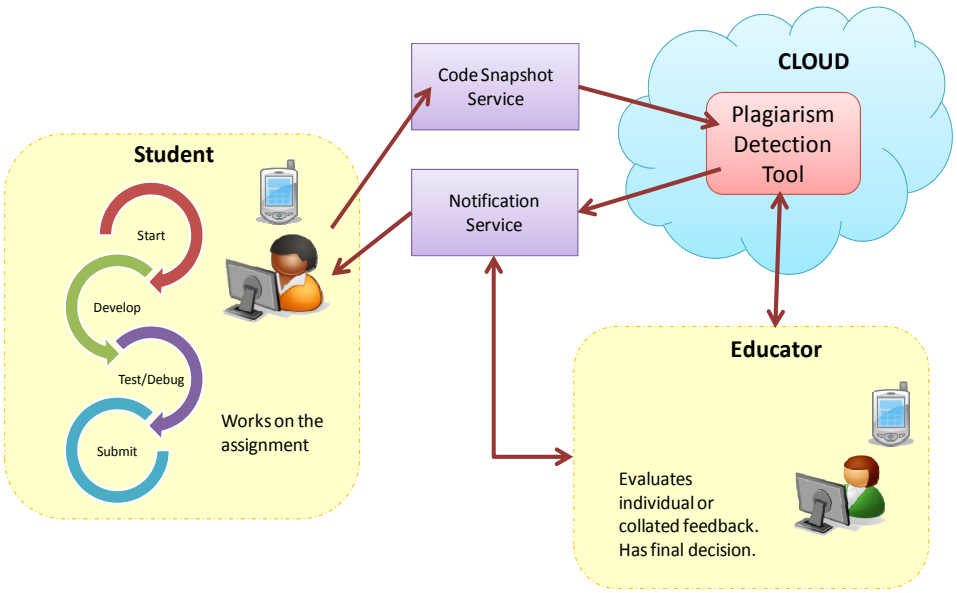

Figure 2: Workflow to support metrics identified

To generate the metric that indicates time spent on writing code for the assignment the time difference between when the first snapshot and last snapshot were received may be used. The number of modifications and the text that was modified may be determined by comparing the source code of successive snapshots. Each snapshot taken over time may also indicate the length of the addition which has significant value in the detection process.

Because the PDT does not compare different assignments in the corpus with each other, but rather analyzes the snapshots based on the new metrics (e.g., time spent on writing code, number of

modifications including the text that was modified, and the length of each modification), the tool does not need to wait for all documents to be present in the corpus before analyzing for possible plagiarism.

The notification service may be responsible for notifying students about the determination the PDT makes in real time. In contrast to traditional detection engines, the proposed method allows the student to be notified that the PDT has determined that possible plagiarism is occurring, while the student still has time to take corrective action.

After the student submits the final version of the assignment, the NS notifies the educator, who may use all the metrics gained by the PDT to make a final determination in each case.

In addition to being used to detect whether plagiarism has occurred, the new metrics gained from the PDT can provide clues on why the code was plagiarized.

Enhancing Teaching and Learning by Providing Possible Reasons for Plagiarism

Figure 3 shows how the metrics gained from the PDT can be used by the educator to identify possible reasons for why a student plagiarized a programming assignment. As the metrics are gathered by the PDT they are analyzed by the educator to identify the possible reason for plagiarism. Either a single metric or a combination of metrics may be used to identify the reason for plagiarism.

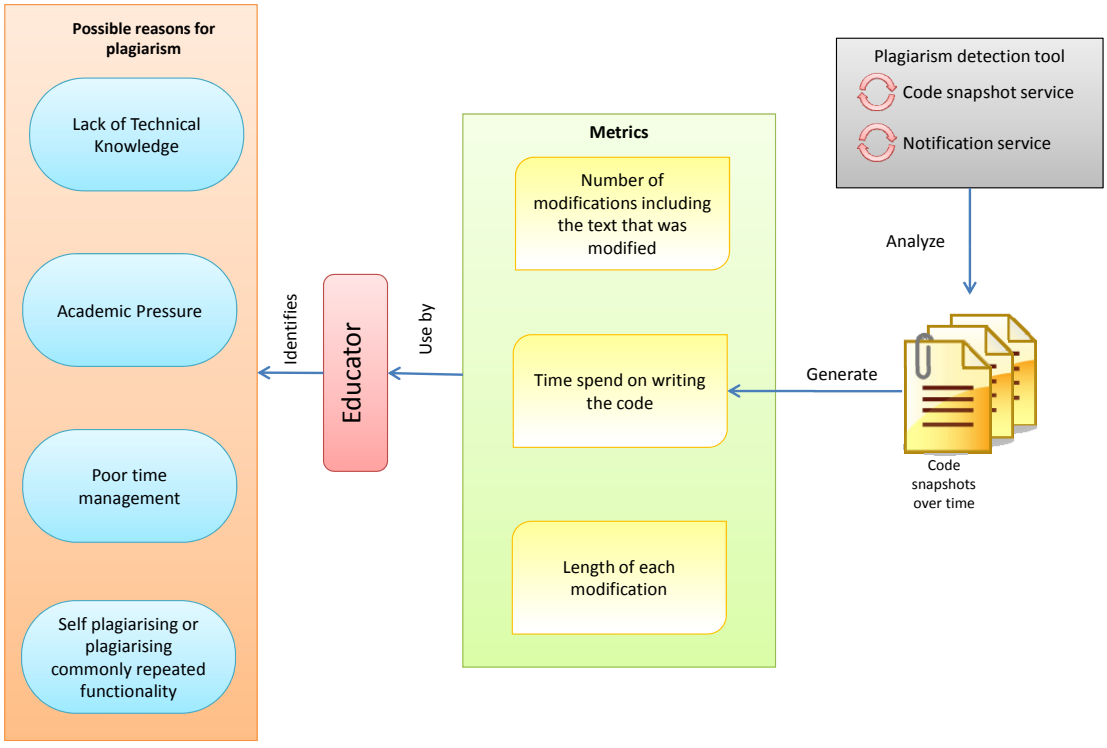

Figure 3: Metrics gathered by PDT to indicate possible reasons for plagiarism

First, a lack of technical knowledge can be identified by looking at the number of modifications including the text that was modified in each successive snapshot. If the code that was inserted in

each successive snapshot varies widely between snapshots, it may indicate that the student is attempting to fit code to the situation blindly and hoping to find a possible solution. Further analysis of the text that was modified can be compared to publicly available code repositories to pinpoint the sources that the student consulted in order to find the solution. This behavior may also point to academic pressure as a possible reason for plagiarism. If a large change occurs between snapshots, and if the code that is changed between successive snapshots shows that both snapshots solve the same problem in a different way, the student may have found code which, in the student's mind, may get a better academic result then the original code written by the student.

Second, poor time management can be identified by looking at the time spent on writing the code. Since the PDT receives regular snapshots while the assignment is being completed, the total time spent should be relatively accurate by considering when the first and last snapshots were received. Successive snapshots with no code difference can be ignored. By combining information regarding the number of modifications including the text that was modified and the time spent on writing code, it is possible to detect students that have added a large number of lines to their code base close to the submission deadline. This indicates that possible plagiarism because of time constraints has occurred. Combining the metric for all submissions may also provide an average time of completion for all students and can be compared to previous assignments with the same level of difficulty by the educator.

Third, self-plagiarizing, or plagiarizing commonly repeated functionality, can be identified by detecting that a student has inserted many lines of code at once and then proceeded to make many small changes. This activity can be detected when a snapshot reveals that a large amount of code has been inserted, and then subsequent snapshots reveal only minor changes being made. This may likely indicate that the student is using code previously written and is adapting the code to fit a given problem. By analyzing the code that was inserted and deleted in each successive snapshot, it is possible to determine whether the small changes the student made were due to self-plagiarism or whether the student made those changes in an attempt to avoid plagiarism detection.

Because an educator may make the final determination as to whether plagiarism has occurred only after all final submissions have been received and reviewed, students can be warned in real-time as the assignment is being completed that they run the risk of plagiarizing their assignments. This real-time feedback provides a more proactive approach to detecting plagiarism than the reactive approach followed by current detection engines. In addition, the proposed method can enhance teaching and learning as the student completes the programming assignment.

Using Plagiarism Detection Tools to Provide Feedback to Students in Real-Time

As part of the student learning through effective feedback project, Juwah et al. (2004) have identified seven principles of good feedback practice in academic environments. According to the reasearchers' view, a good feedback practice is one that:

1. Facilitates the development of self-assessment in learning.
2. Encourages teacher and peer dialog around learning.
3. Helps clarify what good performance is.
4. Provides opportunities to close the gap between current and desired performances.
5. Delivers high quality information to students about their learning.
6. Encourages positive motivational beliefs and self esteem.
7. Provides information that can be used to help shape the teaching.

The authors have also developed a conceptual model for information feedback. This 'formative assessment and feedback model' is based on a model developed by Butler and Winne (1995). The

biggest problem the originally proposed model aims to address is the problem that feedback is usually only available after a learning activity has been completed. In the adapted model the student is placed in a central role regarding the feedback process. The student and educator are at all times actively involved in monitoring and regulating the goals as set out by the educator. Figure 4 shows how the PDT can be incorporated into the formative assessment and feedback model.

A learning activity like a programming assignment may start with the educator providing the criteria and other goals that should be accomplished by the student. From there, most activities take part as part of the internal student process.

The student may start by studying the provided goals and criteria and may draw on previous domain knowledge to develop a number of personal goals to be achieved while completing the assignment. The next stage involves the student applying a number of tactics and strategies to complete the assignment, or part of the assignment, by producing a learning outcome. In this case the learning outcome may be a document(s) containing source code.

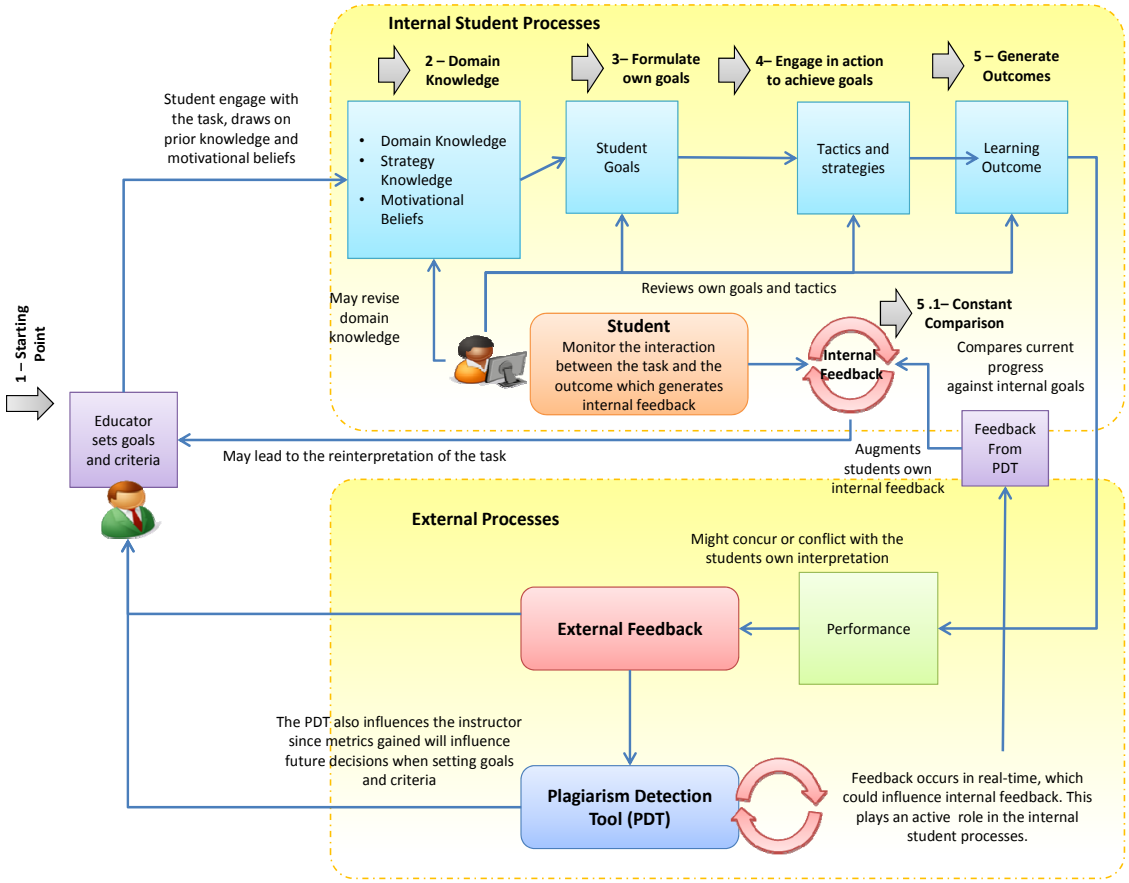

Figure 4: Incorporating the PDT into the formative assessment feedback model

The process of writing the source code for the assignment may be influenced by internal feedback. This feedback may be generated by the student on a continuous basis. This may lead to the student re-assessing personal goals. It may even lead to the student revising and updating existing domain knowledge which stimulates cognitive development. This internal feedback is continuously augmented by the PDT via the Notification Service (NS) based on results from analyzing the code provided by the Code Snapshot Service (CSS).

As part of the external processes, the student's performance is measured and feedback is provided by an external entity. Usually, this external feedback occurs only after a learning outcome is achieved. The new knowledge gained from the external feedback is only used and tested by the student during the next learning activity, whilst some academic value is lost in the current instance.

Because the PDT may provide students with feedback in real time, the external feedback processes can play an active role in the students' own internal feedback and learning process. The feedback generated by the PDT can also influence the educator. In addition to being useful for detecting plagiarism, the metrics gained by the tool may influence an educator's decisions regarding the goals and criteria for future programming assignments. It may also impact and highlight certain areas in the curriculum which needs additional modifications in instruction. The metrics, and the way that they are being gathered, may also allow for different types of programming assignments to be used when assessing students.

Plagiarism Detection Tool to Aid Learning on All Cognitive Levels

Buck and Stucki (2000) argue strongly that teaching computer programming should start at a low cognitive level and slowly progress to a higher level. This approach of moving from a lower level to a higher level often relies on Bloom's taxonomy of cognitive learning (or on the revised digital version thereof (cf. Churches, 2009)) for its structure.

Bloom's revised taxonomy follows the process of learning through a number of categories, starting with lower order thinking skills and progressing to higher order thinking skills. Unlike Bloom's original taxonomy, the revised taxonomy names each category by using a verb. Beginning with the lowest order, marked by the name *remembering*, the revised taxonomy progresses through categories named with the verbs *understanding*, *applying*, *analyzing*, and *evaluating*. The taxonomy ends with the highest order thinking skill being named *creating*. As a learning process, Bloom essentially requires a concept to be remembered before it can be understood. Once understood, the concept can be applied and then analyzed to evaluate the impact. It is only after all the other categories have been adhered to that creation can take place (Churches, 2009; Krathwohl, 2002).

A plagiarism detection tool that provides real time feedback with metrics indicating how the code was constructed can help educators evaluate and ensure the academic reliability of different assessment methods and strategies beyond those requiring students to write complete programs. Some of these alternative assessment methods may include the use of skeleton programs, code inspection, and self-assessment.

Lister (2000) describes an alternate approach to letting students write complete programs as soon as possible in the academic calendar year. That approach concentrates on the first four levels of Bloom's taxonomy; namely *remembering*, *understanding*, *applying*, and *analyzing*. Multiple-choice questions were used in combination with the completion of skeleton programs where only a skeleton is provided and the students should complete the program by writing lines of code that had been left out of a complete program. Lister argues that skeleton programs teach students good programming practice and guide their thinking into a productive learning pattern.

Using traditional plagiarism detection engines that rely on comparing text may be problematic in skeleton programs. This is because the code submitted may not vary significantly between students. In addition the code may consist of a number of small changes to the original program. In contrast, with the PDT the educator can evaluate the time spent on writing the code and what changes were made between each successive snapshot. If the snapshots are made in quick enough

succession they should give a good indication of how given code was inserted to complete the skeleton.

McMeekin, von Konsky, Chang, and Cooper (2009) conducted a pilot study that required undergraduate students to inspect code. The authors concluded that code inspections can lead to students developing higher cognitive levels. Alaoutinen and Smolander (2010) noted how a goal-oriented learning environment can motivate students and result in meaningful learning. The authors go on to say that this goal-oriented environment can be created by involving self-assessment in the teaching process, but self-assessment is often disregarded because of reliability issues. We believe that combining self-assessment with code inspection while maintaining reliability may be made possible by the implementation of the PDT.

Using the PDT the educator can provide students with some code to inspect and to comment on. Using the metric gained by the PDT (e.g., time spent on writing code, number of modifications including the text that modified, and the length of each modification), the educator can assess how students interpreted the code. This has great teaching value for both the educator as well as the students.

Since the PDT gives a complete overview on how code was constructed line by line, the successive snapshots and the differences between the snapshots can be used by a student to not only assess his or hers own work but also the work of other students.

The authors believe that the utilization of the PDT could aid in the presentation of programming, by allowing process to evaluate student's performance on assessments, based on each of the digital taxonomy levels as presented by Churches, (2009).

Conclusion

Literature suggests that plagiarism remains an active and ongoing problem and threat to academic institutions. Reactive and proactive methods of plagiarism prevention and detection do not align and complement each other. In traditional source code plagiarism detection engines that consider plagiarism detection a pattern matching problem, no indication is given on the reason for the student choosing to plagiarize. The proposed implementation of detecting plagiarism by utilizing metrics gained over time which is reported in real-time aims to improve the link between pro-active and re-active methods of plagiarism detection.

The formative assessment and feedback model presented as part of this paper aims to limit the practice of plagiarism by providing real-time feedback to the student as well as the educator. This process could act as a deterrent for the practice as well as enhance the learning processes of the individual student. By incorporating real time feedback the student is proactively warned of possible plagiarism infringement and can correct the situation. The educator will still have the final say on a case by case basis maintaining the ability to reactively respond to plagiarism.

The next phase of this research is addressing the issues identified in the literature review by the creation of a prototype based on the PDT framework. Research needs to be conducted on how best to implement the PDT and the associated services. Once the PDT prototype is created experiments can be conducted to investigate if the method of plagiarism detection presented in this paper acts as an effective plagiarism deterrent.

References

Alaoutinen, S., & Smolander, K. (2010). Student self-assessment in a programming course using Bloom's revised taxonomy. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '10* (pp. 155–159). New York, NY, USA: ACM. Retrieved November 21, 2012, from http://doi.acm.org/10.1145/1822090.1822135

Buck, D., & Stucki, D. J. (2000). Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. *SIGCSE Bulletin, 32*(1), 75–79.

Butler, D. L., & Winne, P. H. (1995). Feedback and self-regulated learning: A theoretical synthesis. *Review of Educational Research, 65*(3), 245–281.

Churches, A. (2009). *Bloom's digital taxonomy*. Retrieved November 21, 2012, from http://montgomeryschoolsmd.org/uploadedFiles/departments/techtraining/homepage/BloomDigitalTaxonomy2001.pdf

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

Devlin, M. (2006). Policy, preparation, and prevention: Proactive minimization of student plagiarism. *Journal of Higher Education Policy and Management, 28*(1), 45–58.

Devlin, M., & Gray, K. (2007). In their own words: A qualitative study of the reasons Australian university students plagiarize. *High Education Research & Development, 26*(2), 181–198.

Evans, D., & Peck, M. (2006). Inculcating invariants in introductory courses. *Proceedings of the 28th International Conference on Software Engineering* (pp. 673–678). Retrieved July 31, 2012, from http://dl.acm.org/citation.cfm?id=1134388

Gitchell, D., & Tran, N. (1999). Sim: A utility for detecting similarity in computer programs. *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education, SIGCSE '99* (pp. 266–270). New York, NY, USA: ACM. Retrieved November 15, 2012, from http://doi.acm.org/10.1145/299649.299783

Howard, R. M. (2002). Don't police plagiarism: Just teach! *Education Digest, 67*(5), 46-49.

Jones, E. L. (2001). Metrics based plagarism monitoring. *Journal of Computing Sciences in Colleges, 16*(4), 253–261.

Joy, M., Cosma, G., Yau, J. Y., & Sinclair, J. (2011). Source code plagiarism #x2014 :A student perspective. *IEEE Transactions on Education, 54*(1), 125 –132. doi:10.1109/TE.2010.2046664

Juwah, C., Macfarlane-Dick, D., Matthew, B., Nicol, D., Ross, D., & Smith, B. (2004). *Enhancing student learning through effective formative feedback*. Higher Education Academy (Generic Centre). Retrieved November 21, 2012, from http://www-new2.heacademy.ac.uk/assets/documents/resources/resourcedatabase/id353_senlef_guide.pdf

Koul, R., Clariana, R. B., Jitgarun, K., & Songsriwittaya, A. (2009). The influence of achievement goal orientation on plagiarism. *Learning and Individual Differences, 19*(4), 506–512.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into Practice, 41*(4), 212–218.

Lancaster, T., & Culwin, F. (2005). Classifications of plagiarism detection engines. *E-journal ITALICS, 4*(2). Retrieved July 26, 2012, from http://www.ics.heacademy.ac.uk/italics/Vol4-2/Plagiarism%20-%20revised%20paper.htm

Leedy, P. D., & Ormrod, J. E. (2004). *Practical research: Planning and design* (8th ed.). Prentice Hall.

Lim, V. K. G., & See, S. K. B. (2001). Attitudes toward, and intentions to report, academic cheating among students in Singapore. *Ethics & Behavior, 11*(3), 261–274.

Lister, R. (2000). On blooming first year programming, and its blooming assessment. *Proceedings of the Australasian Conference on Computing Education, ACSE '00* (pp. 158–162). New York, NY, USA: ACM. Retrieved November 21, 2012, from http://doi.acm.org/10.1145/359369.359393

Lukashenko, R., Graudina, V., & Grundspenkis, J. (2007). Computer-based plagiarism detection methods and tools: An overview. *Proceedings of the 2007 International Conference on Computer Systems and Technologies* (Vol. 285). Retrieved November 15, 2012, from http://moodle.unitec.ac.nz/pluginfile.php/5525/mod_resource/content/0/Articles/a40-lukashenko.pdf

McMeekin, D. A., von Konsky, B. R., Chang, E., & Cooper, D. J. A. (2009). Evaluating software inspection cognition levels using Bloom's Taxonomy. *22nd Conference on Software Engineering Education and Training, 2009. CSEET '09* (pp. 232 –239).

Miller, A., Shoptaugh, C., & Wooldridge, J. (2011). Reasons not to cheat, academic-integrity responsibility, and frequency of cheating. *The Journal of Experimental Education, 79*(2), 169–184.

Olt, M. R. (2002). Ethics and distance education: Strategies for minimizing academic dishonesty in online assessment. *Online Journal of Distance Learning Administration, 5*(3). Retrieved November 27, 2012, from http://www.westga.edu/~distance/ojdla/fall53/olt53.html

Park, C. (2003). In other (people's) words: Plagiarism by university students–literature and lessons. *Assessment & Evaluation in Higher Education, 28*(5), 471–488.

Plagiarism. (2012). In Merriam-Webster.com. Retrieved November 10, 2012, from http://www.learnersdictionary.com/search/plagiarism

Power, L. G. (2009). University students' perceptions of plagiarism. *The Journal of Higher Education, 80*(6), 643–662.

Sheard, J., Dick, M., Markham, S., Macdonald, I., & Walsh, M. (2002). Cheating and plagiarism: Perceptions and practices of first year IT students. *ACM SIGCSE Bulletin, 34*, 183–187). Retrieved July 31, 2012, from http://dl.acm.org/citation.cfm?id=544468

Voelker, T. A., Love, L. G., & Pentina, I. (2012). Plagiarism: What don't they know? *Journal of Education for Business, 87*(1), 36–41.

Wan, R., Md Nordin, S., Halib, M., & Ghazali, Z. (2011). Plagiarism among undergraduate students in an engineering-based university: An exploratory analysis. *European Journal of Social Sciences*. Retrieved November 19, 2012, from http://www.europeanjournalofsocialsciences.com/ISSUES/EJSS_25_4_08.pdf

Whale, G. (1990). Identification of program similarity in large populations. Comput. J., 33(2), 140–146.

Wise, M. J. (1992). Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. *Proceedings of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education, SIGCSE '92* (pp. 268–271). New York, NY, USA: ACM. Retrieved November 15, 2012, from http://doi.acm.org/10.1145/134510.134564

Zobel, J. (2004). *Writing for computer science* (2nd ed.). London: Springer.

Biographies

Fredre Hattingh is currently enrolled for his M-Tech in Technical Applications at TUT. His research interests include Living Labs, Virtualization and Plagiarism detection. Other areas of interests include Open Source Software and emerging technologies.

Bertie Buitendag is currently enrolled for his D-Tech in Enterprise Application Development at TUT under the supervision of Prof JS van der Walt. His core research area includes ICT Knowledge support for emergent farmers and Living Labs. Other areas of interest include the: Semantic Web, (WEB 3.0) and WEB 2.0, and ICT's for community upliftment.

Prof. Jacobus (Potjie) van der Walt has been intensively involved into ICT research over the past 27 years at the Tshwane University of Technology (TUT). His core research interests currently pertain into the study of emergent community-oriented ICT support, with specific reference to portal based applications for emergent farmers. He has successfully supervised numerous post graduate students. He was one of the first academia in South Africa to publish a paper regarding community oriented Living Labs. He also started the Soshanguve LL initiative.