nExaminer: A Semi-automated Computer Programming Assignment Assessment Framework for Moodle

Zheng Cheng, Rosemary Monahan, Aidan Mooney

zcheng, rosemary, amooney {@cs.nuim.ie}

Department of Computer Science

National University of Ireland, Maynooth

Co.Kildare, Ireland

Abstract
In this paper, we present the *nExaminer* framework, a learning support tool for the Virtual Learning Environment, called Moodle. *nExaminer* is a framework for the semi-automated assessment of computer programming assignments. The motivation for developing the framework is based on the observation of a major problem associated with traditional assignment assessment in Moodle - managing an effective relationship between the instructor and the students is difficult. Providing a tool that will support both the student (through feedback on progress) and the instructor (through support the correction of assignments) will help to reduce this problem. The design and implementation of our proposed solution, the *nExaminer* framework, is discussed in this paper. The benefits that our framework provides to both students and instructors are also presented. Experimental results with the *nExaminer* framework have been encouraging. These show that the framework provides instructors with the ability to semi-automatically generate subjective feedback and automate the process of objective assessment within Moodle in an efficient manner. Moreover, the results also manifest that students are motivated by the usage of automated objective assessment in the *nExaminer* framework.

Keywords
Semi-automation, assignment assessment, computer programming, increasing code coverage, Moodle.

1. Introduction

Computer programming is an essential but difficult skill that many students are expected to learn. It is important that the students solve enough discipline-specific problems in order to understand and remember certain concepts (McCracken et al., 2001). The discipline-specific problems for Computer Science are usually presented to students as programming assignments. While the assignment feedback plays an important role in student's learning experience (Dafoulas, 2005), sheer number of assignment submissions and the complexity involved in assessment often prevent instructors from offering feedback in a timely manner. As a result, students do not learn from their efforts as feedback becomes available too late in their learning process.

Moodle [1] is an internet-based virtual learning environment that enables instructors and students to communicate and exchange knowledge. It is a free web application that instructors can use to create an online learning site for their courses. The application allows instructors to upload courses resources and view student activity. Its support for students includes access to course resources, course centric discussion forums and assignment upload facilities.

There are several advantages identified for using Moodle to teach computer programming (Robling et al., 2010). However, generating feedback for each student is still a challenging task for the instructors.

To manage computer programming assignment activities with a reasonable amount of resources in Moodle, we propose the nExaminer framework. It uses an existing tool called JUnit[2] to drive automated objective aspects of assessment (e.g. to ensure that the required functionalities are presented in students' solution), and present the students with immediate feedback. Meanwhile, uploading restrictions are applied to prevent the automated nature being abused. nExaminer also facilitates instructors to review subjective aspects by utilizing a novel code-structure analyzing tool, namely AgitarOne[3]. Consequently, students can have immediate objective feedback that helps them to learn computer programming, and instructors can review assignments efficiently.

The rest of the paper is organised as follows. The background to the problem is presented in Section 2. Section 3 designs our solutions to the problem. Section 4 evaluates our work using an experiment. Section 5 critically analyses our work. Finally, a conclusion and future work are provided in Section 6.

2. Background

Traditional computer programming assignment assessment involves counterproductive activities. These activities become obstacles to the well-known learning model - Bloom's taxonomy, and thus make the learning process less-efficient.

2.1. Bloom's Taxonomy

Bloom's taxonomy (Krathwohl, 2002) identifies three domains of learning. Each domain has its learning objectives. In the most known cognitive domain (shown in

[1] http://moodle.org/about/
[2] http://www.junit.org/
[3] http://www.agitar.com/

Figure 1), the objective is to remember what has been taught. In order to achieve this ultimate objective, a list of activities is presented as a series of levels or prerequisites. This suggests that one cannot effectively address higher levels until those below have been managed.

To interact with this model, the instructor usually acts as a counselor or a collaborator, providing positive advice and motivation at each level to help students eventually remember the taught material.

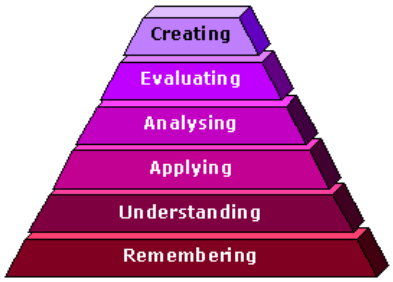

Figure 1: Cognitive domain of Bloom's taxonomy

2.2. The Assignment Assessment Challenge

Traditionally, computer programming assignment assessment in Moodle takes place when all students submit their assignments. Although consensus on the best way of evaluating assignments has not been established, previous research (Helmick, 2007) takes the position that two types of criteria are applied in the assessing procedure:

- Objective assessment (e.g. required functionality).
- Subjective assessment (e.g. structure, style, efficiency).

After evaluation has been completed, the instructor combines objective and subjective assessment results into a final report and updates them, usually via Moodle.

In this section we analyse the difficulties involved in traditional computer programming assignment assessment.

2.2.1. Objective Assessment

The major problem of objective assessment is that students' solutions lack a consistent format. Thus, instructors have to examine each submission individually (Hayes et al., 2007). This is a time consuming process, particular where large class sizes are involved (as is typical of early stage computer programming courses). Furthermore, each solution may provide the functionality required in a different manner. For example, an assignment that requires the student to write a program to sort a sequence of numbers into ascending order could be solved by a strategy that locates the smallest number in the sequence and insert that number into its correct location in the sorted number sequence. Alternatively, the solution might sort the first two numbers in the sequence, then the first three numbers, and so-on, until all numbers in the sequence have been arranged in ascending order. Such diversity in assignment solutions ranges from assignment to assignment and requires the instructor to spend time understanding and deciphering the students' solution. Providing support to test if the solution is at least a valid one which provides the required functionality would greatly assist in the turnaround time for assignment correction.

2.2.2. Subjective Assessment

Usually, computer programming assignment assessment becomes complicated when subjectivity is involved. Assessing a program structure is a sub-area of subjective assessment, which usually takes place after objective assessment (Maxim and Venugopal, 2004). It mainly tries to discover the sections of student's solution that are not used in calculating the solution to a particular instance of the problem being solved. We refer to these unused sections as unexecuted code, and the sections that are used in solving a particular instance of the problem as executed code. For example, when we give an odd number to a program solution that checks the parity of any given value, the executed code will be the sections that check for the odd number, and the unexecuted code will be the part for checking the even number. In this way, new behaviours of the Assignment Under Test (AUT) can be hidden under objective testing. Code coverage (Roper, 1994) is a metric to measure the amount of executed code for a given program. The instructor tries to provide enough problem instances that increase the code coverage, so that new behaviours of unexecuted code can be observed and validated (Roper, 1994). However, in our experience, increasing code coverage manually can be difficult. This is because:

- There is no intuitive way to tell whether code coverage can be increased, e.g. a piece of code that will never execute.
- Increasing code coverage manually requires tracking the internal workflow. This manual check can be error-prone and time-consuming (Roper, 1994).

2.2.3. Revising the Solution

In the cognition domain of Bloom's taxonomy, the applying level is defined as applying known knowledge on a discipline-specific problem [Krathwohl, 2002]. It might not directly lead the learner to the understanding level because the application of known knowledge can fail. Thus, multiple application activities might occur. In such cases, the learner learns from previous experience, enriches their known knowledge and eventually moves to the understanding level. We refer to the multiple applications of knowledge as revising. The feedback from 37 students of an introduction programming course at NUIM reveals that, over 86% (32/37) of students do not revise their assignment in any form after their initial submission. We observed, with interest, that when the instructor assessed the 37 student submissions, none of them were error-free. We analyse the reasons behind these facts:

- Previous research (Norman, 2003) finds that anxiousness tends to narrow the thought process so that people will concentrate on problems that are directly relevant. Thus, students do not revise their solutions before their final submission perhaps because they are often too anxious about their grade when they are constructing their solutions.
- Students do not revise their solutions after their initial submission perhaps due to the fact that feedback is not provided (or not given within an appropriate timeframe), so students are not motivated to revise (Yeh, 2005).

As a result, the difficulties involved in the traditional computer programming assignment assessment compromises the learning experience of students and the instructor's role in the Bloom's taxonomy model.

3. Framework Design

An automatic objective testing component and an automatic increased code coverage component are integrated into a framework to solve the learning and assessment difficulties identified in Section 2. This integration forms the basis of our nExaminer framework. In this section, we provide an overview of the framework and briefly explain how our solution is implemented. Further technical details are provided in Cheng, 2011.

3.1. Framework Overview

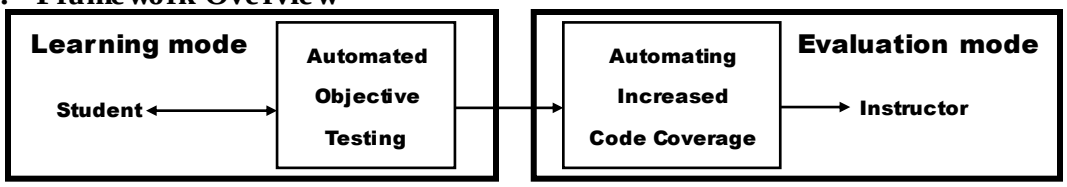

Figure 2: nExaminer framework overview

As Figure 2 demonstrates, the nExaminer framework can be used in two modes: a learning mode and an evaluation mode. The instructor will first provide students with their assignment and an accompanying solution template which encourages all submissions to be in a consistent format. When the student completes their assignment (in this case writing their computer program), their solutions will be uploaded onto Moodle and the first part of our learning scaffolding, automated objective testing will commence. The objective testing results will then be stored in the Moodle database and displayed to the students accordingly.

Students can draw on the objective results to revise their work, and maintain their attempts in the nExaminer framework. We refer to this as the learning mode. In this mode, students learn from previous experience and challenge themselves to construct improved solutions. This is repeated until either the student is satisfied with their work or is restricted by the framework. The final submitted solution will be examined by the instructor for subjective review. We refer to this as the evaluation mode. In this evaluation mode, the student's solution is subjectively reviewed by the instructor, facilitated by automating increased code coverage to find and validate new behaviours of the AUT. Finally, the instructor will generate feedback and update it on Moodle. A diagrammatic overview of the information flow in nExaminer framework is presented in Figure 3.

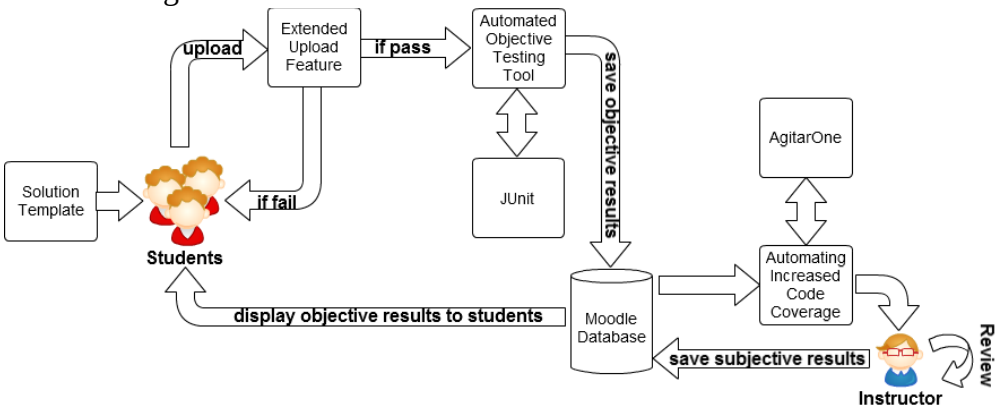

Figure 3: The diagrammatic overview of automated objective testing

3.2. Automated Objective Testing

As we discussed in Section 2.2.3, anxiety and slow feedback prevent students revising their assignment solutions. Previous research finds that the interactive nature of automated objective testing can motivate students to revise their work more frequently with the assistance of immediate functionality results (Reek, 1989). In this work, we implement an automated objective testing tool by using a combination of the existing JUnit tool and a solution template that is supplied by the instructor at the time that the assignment is set. In the case of assessing computer programming assignments, the template that we give to students is a Java[4] class file. This file documents the essential components (Java methods) of the solution. Each essential component is provided with a name, the information that it requires to obtain a correct solution to the component of the assignment that it solves and the components required functionality. Thus, all students construct their solutions on top of the same solution template. This allows JUnit to send the same set of problem instances to the component under test in each submission, and compare the real output from the solution with the expected one, thereby achieving objective assessment without human interference (Helmick, 2007).

We focus on integrating automated objective testing with Moodle and how to prevent the students from abusing its interactive nature.

3.2.1. Extended Upload Feature

The student who wants to repeat the assignment evaluation process is motivated by the prospect of improving their grades on successive attempts, and also is interested in feedback (Dafoulas, 2005). However, immediate resubmission after a previous attempt may yield an undesired outcome: students tend to make small changes to their solutions without thinking thoroughly, and keep submitting to see the feedback changing (Dafoulas, 2005). Thus, we believe this resubmission procedure provides little progress in learning.

Previous research proposes penalising students for sending failed solutions more than a specified number of times (Reek, 1989). We worry that students may be intimidated by this restriction and stop revising once penalized. Therefore, we propose adding upload limitations to Moodle in order to let students inspect their coding behaviour thoroughly before their next submission. Specifically, we design the extended upload feature on Moodle to check that the following rules hold when students submit:

- There should be a predetermined time interval between two submissions.
- Total submission times should not exceed certain times.

If students can successfully submit their solutions on Moodle, we assume these solutions are the results of thorough thinking.

3.2.2. Integration with Moodle

When the student uploads a solution, it will be sent to our extended upload feature to check against a set of upload rules. If any rules are violated, the upload procedure will terminate. Otherwise, the submission is uploaded to Moodle, and then assessed by the automated objective testing tool. The objective results will be

[4] http://java.com/

generated by the automated objective testing tool and stored in the Moodle database. By the end of the upload procedure, the student can see whether their submitted solution behaves as expected. An example of the objective result is shown in Figure 4.

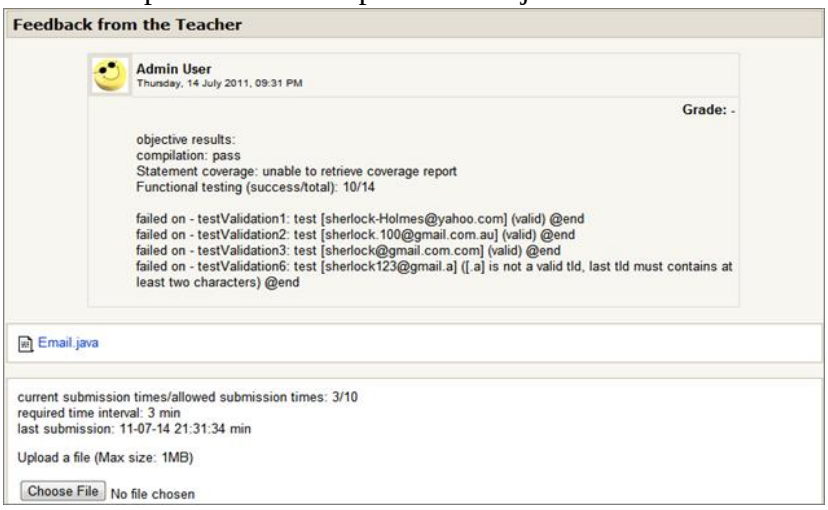

Figure 4: Example of objective testing results

As a result, the automated objective testing provides immediate feedback that allows students to have an extent of confidence and enhanced motivation to revise their solutions. Meanwhile, the students can still use Moodle as a platform to share knowledge.

3.3. Automating Increased Code Coverage

As presented in Section 2.2.2, increasing code coverage manually can cause the instructor great difficulty when performing subjective reviewing. Various techniques have been proposed to automate the process of increasing code coverage to observe new behaviours of the Software Under Test (SUT) (Visser, 2004; Kannan and Sen, 2008). Among these, Dynamic Symbolic Execution (DSE) is especially suited to our work because of its capability of generating effective inputs that reach high code coverage of the SUT (Kannan and Sen, 2008).

We integrate an existing tool that implements DSE with our nExaminer framework. The tool that we use is called AgitarOne (Boshernitsan et al., 2006). The integration of these two tools will automate the process of increasing code coverage. Inputs can be generated with maximum code coverage for the AUT using AgitarOne, thereby ensuring all components of the student's solution is tested. The instructor can then subjectively review each input to determine its validity. Determining the validity can be achieved through checking real output of the solution with the expected output. If there is a piece of the solution that remains unused while solving the problem instances generated, then the instructor can determine that piece of the students solution as unreachable. The submission of solutions that contains components that are unreachable could be discouraged through feedback to the student.

Consequently, the task of increasing code coverage is accelerated by the facility from the nExaminer framework. It allows resources to be distributed to other subjective tasks that are difficult to automate. When other subjective tasks have finished, the instructor can update subjective feedback on Moodle and give the final grade.

4. Results

We invited 60 students to use the nExaminer framework. The students were first year students with similar academic background. Each student was provided with the template described in Section 3.2. In addition, students were restricted to wait three minutes before their next submission in order to let them think thoroughly through what they were submitting. Table 1 shows the usage statistics of the automated objective assessment in the nExaminer framework.

Count	Total Submission	Average test cases success rate	Average interval submission in sec
1	60	62.63%	
2	50	63.08%	470
3	31	65.55%	392
4	26	73.33%	273
5	15	78.66%	288

Table 1: Results of automated objective assessment

We observed that most students were encouraged by the nExaminer framework to perform multiple submissions. Their performance showed a steady improvement – we observed that two students' solutions passed all the predefined test cases. One of these successes was achieved through three submissions; another student submitted six times and then constructed an error-free solution. We can report that other students also benefited from the system - as can be seen by the increasing test cases success rate on each submission. Our observations indicate students are willing to use the nExaminer framework to construct better solutions.

We also gathered feedback from 60 students who used the nExaminer framework. The feedback shows that 46 out of 60 students think the system assisted their learning process. Students surveyed also reported that they like the form in which the feedback is presented.

To get an indication whether automating increasing code coverage can assist the instructors in assignment assessment, we randomly selected a students' solution. This solution was then evaluated by two instructors. One of the instructors used the traditional method discussed in Section 2.2 to perform subjective reviewing, and the other instructor was facilitated by the nExaminer framework to subjectively review the solutions. The results are shown in Table 2:

	nExaminer framework	Traditional
Total Time spend on subjective reviewing	4 min	21 min
Total Number of new behaviour identified	1	0

Table 2: Results of subjective testing experiment

The results show that time saved on subjective reviewing was considerable. Using dynamic symbolic execution can effectively accelerate the subjective assignment reviewing process.

In conclusion, the results of this experiment are promising and reveal the performance and acceptance of the nExaminer framework on Moodle. The nExaminer

framework not only motivates students to reach a higher standard of grading criteria but also semi-automatically helps the instructors in both subjective and objective reviewing. This can save a great amount of time that could be potentially distributed to activities that are not easily automated.

5. Critical Analysis

Automated computer programming assignment assessment is an active research area (Helmick, 2007; Reek, 1989; Maxim and Venugopal, 2004; Edwards, 2003). Over a dozen mature and experimental systems (or frameworks) have been proposed in recent decades to enhance the programming learning performance of students.

Among these, the most similar work to the work presented in this paper is Web-CAT (Edwards, 2003). We both use the testing framework and allow the product to be accessed from the internet. However, Web-CAT emphasises that students should be given the responsibility of providing test data to fully test their own code, which ensures high code coverage of submitted solutions. Their assumption is that students will have a better knowledge of solutions that they build rather than their instructor. This is indeed a valid point; however, we feel this test-driven development enforced by Web-CAT will only provide feedback that is of limited help for students to locate and remove errors from their solutions. To find out more about errors in their own solutions, it will be necessary for students to write new test cases which seem like an endless practice. We believe students do their best work when they concentrate on one thing. Thus, we narrow students' responsibility to focus on meeting certain specification, while using automatically generated objective results and facilitating subjective review to allow the instructor to perceive behaviours of the AUT efficiently. As our experiment has shown, the nExaminer framework which we designed, motivates students to meet the requirements through constant revision of their work. Meanwhile, by drawing on the facility from the nExaminer framework, namely AgitarOne, inputs are automatically generated to ensure a full picture of the student's solutions. Thus, the performance of observing new behaviours in solutions is enhanced for instructor.

Furthermore, to incorporate with web technology, we effectively integrated the nExaminer framework with Moodle. Moodle is responsible for organising and managing learning activities. The nExaminer framework facilitates this by automatically generating feedback, thereby efficiently providing materials which help students to evaluate and analyse their work. As described in Section 4, most students are motivated to interact with nExmainer with the intention of constructing a better solution.

The interaction between the system and the students can be abused. The work of TRY (Reek, 1989) motivates our design. Instead of penalising students for failures after a threshold for submissions, we add extra extensions when uploading. In this way, we create a buffer for students to inspect their coding practice thoroughly before the next evaluation.

We acknowledge that there is a tradeoff in the time spent on assignment design (e.g. test case design) and semi-automated assignment assessment. However, the implications can be far reaching – not only can students and instructors benefit from

the nature of semi-automated evaluation, but a well-designed assignment might also contribute to the modern pedagogy that benefits more than one generation of students. A further benefit for the instructor is that the time spent on assignment design will pay off where large class sizes, and hence when large volumes of assignment correction, are involved.

6. Conclusion and Future Work

In this paper, we presented the nExaminer framework which provides solutions to address problems in traditional methods for teaching and learning computer programming. Our nExaminer framework is incorporated with Moodle through its modified automated objective assessing and facilitated subjective assessing. The feedback received from students shows a better learning environment as a result of iterative feedback to students in the early stage of their solution submissions. Feedback from instructors is also positive, reporting a faster turnaround time on assessment correction and hence providing more time to provide detailed feedback to students.

In our experiment we observed that although assignment correction rates (in terms of objective testing results) are continuously increasing, few of them finally reach a 100% correction rate. There are a range of reasons why this occurs, for example, when students receive the functional report generated from the system, they might spend a great deal of time solving the hardest problem first, and have no time left for the easy ones; or they randomly pick one failed test case from the report instead of trying to solve one specific problem.

Thus, we feel the need of decompose the whole test suite into smaller sets, and then prioritise them according to their difficulties. Students' submission will be evaluated by the next-level-test-cases only if their work passed the threshold test cases first. As a result, students are solving the problem in an iterative manner. Moreover, only the functionality report of the current level of test case(s) will be displayed. This focused display of information is more likely to focus the student to improve particular components of their assignment solution.

We will use nExaminer for a course where all assignments are submitted by students and assessed by instructors through nExaminer and we are confident that our findings to date will be confirmed by the expansion of our users.

We also plan to find an intuitive way to present the results from the nExaminer framework in Moodle, so that the instructor will have further support for grading of the assignments.

Reference

McCracken, M. and Almstrum, V. and Diaz, D. and Guzdial, M. and Hagan, D. and Kolikant, Y.B. and Laxer, C. and Thomas, L. and Utting, I. and Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bulletin, Volume 33 Issue 4.

Robling, G. and McNally, M. and Crescenzi, P. and Radenski, A. and Ihantola, P. and Sánchez-Torrubia, G.M.(2010). Adapting moodle to better support CS education. Proceedings of the 2010 ITiCSE working group reports on Working group reports (ITiCSE-WGR '10).

Yeh, H. (2005). The use of instructor's feedback and grading in enhancing students' participation in asynchronous online discussion. Advanced Learning Technologies (ICALT 2005).

Dafoulas, G.A. (2005). The role of feedback in online learning communities. Advanced Learning Technologies (ICALT 2005).

Helmick, M.T. (2007). Interface-based programming assignments and automatic grading of Java programs. Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE 2007).

Reek, K.A. (1989). The TRY system -or- how to avoid testing student programs. Proceedings of the twentieth SIGCSE technical symposium on Computer science education (SIGCSE '89), Vol. 21, No. 1.

Maxim, M. and Venugopal, A. (2004). FrontDesk: an enterprise class Web-based software system for programming assignment submission, feedback dissemination, and grading automation. Advanced Learning Technologies.

Edwards, S.H. (2003). Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. Proceedings of the International Conference on Education and Information Systems: Technologies and Applications(EISTA 2003).

Visser, W. (2004). Test input generation with Java PathFinder. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA 2004), Vol. 29.

Kannan, Y. and Sen, K. (2008). Universal symbolic execution and its application to likely data structure invariant generation. Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA 2008).

Boshernitsan, M. and Doong, R. and Savoia, A. (2006). From daikon to agitator:

lessons and challenges in building a commercial tool for developer testing. Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA '06).

Krathwohl, D.R. (2002). A Revision of Bloom's Taxonomy: An Overview. Theory into Practice, Vol.41, No. 4.

Norman, D.A. (2003). Emotional Design: Why We Love (or Hate) Everyday Things. Basic Civitas Books.

Hayes, A. and Thomas, P. and Smith, N. and Waugh, K. (2007). An investigation into the automated assessment of the design-code interface. Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE 2007).

Roper, M. (1994). Software Testing. The McGraw-Hill Companies.

Cheng, Z. and Monahan, R. and Mooney, A. (2011). nExaminer: A Semi-automated Computer Programming Assignment Assessment Framework for Moodle. Master thesis in Computer Science (Software Engineering) in National University of Ireland, Maynooth.