Introduction of the automated assessment of homework assignments in a university-level programming course

Marko Poženel, Luka Fürst, and Viljan Mahnič
University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, SI-1000 Ljubljana, Slovenia
Email: {marko.pozenel,luka.fuerst,viljan.mahnic}@fri.uni-lj.si

Abstract—Modern teaching paradigms promote active student participation, encouraging teachers to adapt the teaching process to involve more practical work. In the introductory programming course at the Faculty of Computer and Information Science, University of Ljubljana, Slovenia, homework assignments contribute approximately one half to the total grade, requiring a significant investment of time and human resources in the assessment process. This problem was alleviated by the automated assessment of homework assignments. In this paper, we introduce an automated assessment system for programming assignments that includes dynamic testing of student programs, plagiarism detection, and a proper presentation of the results. We share our experience and compare the introduced system with the manual assessment approach used before.

Keywords: automated grading, dynamic testing, programming education

I. INTRODUCTION

The ever-growing use of information technology tools in education compels us to reconsider many facets of the education process [1]. Several tasks that once had to be performed manually have been at least partially automated, such as the assessment of the knowledge absorbed by the students. In the area of programming education, the idea of automated assessment of programming assignments, the goal of which is to write a program to perform a given task, has been around for several decades [2]. In 1960, Hollingsworth [3] described probably the first program for grading programming assignments. He immediately recognized the savings in time and a possibility to teach larger group of students. We can conclude that automated assessment (and grading) of computer programs has been present since the very beginning of programming education.

The assessment of student obligations represents an important and often challenging part in the pedagogical process. Focusing on programming assignments, substantial resources are needed to manage their manual assessment, since there are many aspects related to a good program that have to be considered [4], [5]. As a result, the idea of automation or at least partial automation of the assessment process began to appear [6]. The automated assessment process is not suitable for all types of student assignments and cannot be equally effectively used in all areas. Hollingsworth [3] realized that the assignments (problems) have to be properly structured for the automated assessment to be effective. Douce et al. [7] stated that student task specifications should be defined more precisely if they are to be assessed automatically. It follows that automated assessment can be used effectively if the student’s assignment is clearly defined and if its result is measurable and unambiguous and can be verified according to a strictly defined procedure [8]. One of the fields that meet all the requirements is programming.

Ala-Mutka [5] summarized the results of an internal survey on assessment practices in computer science courses. Almost three quarters of the respondents assessed the students in their courses on the basis of their practical work. While the assignments were most commonly submitted electronically, they were still predominantly assessed in a manual way. Ala-Mutka concluded that many teachers still see the possibilities of computer aided assessment to be limited to simple tasks such as multiple-choice questions at quizzes.

In theory, automated assessment of programming assignments is not very difficult to achieve. Above all, the assessment system should check the output results of student programs against the predefined correct values. However, by having access to appropriate tools and metrics, the quality and structure of programs can also be checked [9], [4]. Unfortunately, automated assessment of programs combined with hundreds of student submissions can lead to a serious problem — source code plagiarism. However, by incorporating tools for detecting plagiarism, we can also provide a greater validity of the assessment [6].

Depending on the degree of automation, assessment applications are divided into automated and semi-automated ones. Both groups use automated testing of programs. The difference is that a fully automated system tests individual components and evaluates the assignment according to the predefined criteria. It is usually used for smaller assignments. For larger assignments, it is not always possible to assess all aspects of a good programming solution. In that case, a semi-automated system is often used. In such a system, the human evaluator uses the partial results of the automated testing of less complex program parts, focuses on the assessment of demanding tasks, and makes decisions to assign correct grade. The final grade thus still depends on the human evaluator [5], [10].

Several approaches to automated program assessment can be found in literature. Typically, they are all based on two basic principles: testing the program in action, or evaluation
of its code and structure. Program assessment is based on the assumption that certain measurable attributes can be extracted from the program and can then be compared to the model or solution [5]. For an educational use, the results of the comparison also have to be backed up with the pedagogical goals of the course. The assessment of a program includes the assessment of its quality, such as correctness, complexity, reliability, coding standards, etc., and therefore involves practices and techniques that are used in software testing. Software testing can be classified into two groups according to whether the program has to be executed during evaluation or has to be evaluated statically from the program code: dynamic analysis and static analysis.

Dynamic testing techniques can be further divided into two categories: black-box testing and white-box testing. In black-box testing, a program is evaluated on the basis of its functional behavior. White-box testing comprises the analysis of the program code and of the code structure. The aim of the above two techniques is to create test cases that clearly show whether the tested program has errors or not. The test cases should have a good coverage over the program model and specifications in order to avoid misleading feedback information that can further lead to incorrect final grades [4]. When testing the students' programs using dynamic testing, we have to take into account bugs, possible malicious code inserts, and programs that could accidentally cause the collapse of the system [5].

In the introductory programming course at the Faculty of Computer and Information Science, University of Ljubljana (UL-FCIS in the following text), we recently, in 2014/15, adopted automated grading of programming assignments, mostly as a result of the growing awareness of the deficiencies of our previous assessment approach. In this paper, we present the system that we introduced for the purpose of the automatic grading of programming assignments. The system can be broadly classified into the dynamic black-box testing category. However, in one of the assignments, a part of the grade was also obtained using code analysis. In the future, we might increase the proportion of white-box testing techniques in the grading process.

The rest of this paper is structured as follows: In Section II, we briefly present the introductory programming course at UL-FCIS. In Section III, we describe our assessment system before the introduction of automated grading. In Section IV, we present our automated grading system in detail. With Section V, we conclude our paper.

II. THE INTRODUCTORY PROGRAMMING COURSE AT UL-FCIS

The course 'Basic Programming' at UL-FCIS teaches the fundamentals of procedural and object-oriented programming in Java. The syllabus of the course covers control structures, arrays, classes and objects, inheritance, and fundamentals of computer graphics. The course is taught in the first semester of the first year. It comprises 15 weeks of formal lectures and 13 weeks of lab sessions. In each block of lectures (three consecutive hours per week), the lecturer introduces a programming subject and presents several basic examples to the entire body of approximately 250 students simultaneously. The lab sessions are held in small groups (15–30 students) supervised by one or two teaching assistants, depending on the size of the group [11].

In the Basic Programming course, we strive for constant improvement. In the academic year 2009/10, UL-FCIS adopted the so-called Bologna reform, which calls for a more active, student-centered way of learning with a greater emphasis on regular work. In the spirit of the Bologna reform, we introduced a system for the cooperative development of computer programs [11] to the lab sessions. In our programming course, we are also faced with a very diverse student population with varied prior knowledge. In order to animate the course and motivate students with programming skills, we introduced board game programming competitions [12]. Our last improvement was the introduction of the automated grading of homework assignments.

III. THE GRADING SYSTEM BEFORE INTRODUCING AUTOMATED ASSESSMENT

Throughout the history of the Basic Programming course, several grading systems have been in use. In this section, we describe the grading system between the academic years 2009/10 (i.e., immediately after the adoption of the Bologna reform) and 2013/14 (before the introduction of automated assessment), inclusive. The Bologna reform prompted us to redesign the grading system, since it recommends that approximately 50% of the student's final grade be obtained through his or her regular work during the semester, e.g., by homework assignments. Before that, the grade was determined almost exclusively by the final written exam. The homework assignments were part of the student's obligations, but they contributed at most a few bonus points to his or her grade.

During the years 2009/10–2013/14, the students were given three programming assignments per semester. For each assignment, the students had one week to finish it at home and submit it. At the next week's lab sessions, they had to implement three independent upgrades to the homework assignment. The teaching assistants then assessed the submitted homework assignments together with the upgrades. Both were assessed during the lab sessions right after the deadline for finishing the upgrades. A greater attention was devoted to the upgrades (3 upgrades × 2 points = 6 points out of 10) than to the homework assignment (4 points out of 10).

The grading system described above did have some advantages. Above all, plagiarism did not make much sense. A student who did not implement the basic assignment and did not understand the code in detail had little or no chance to implement the upgrades. Besides that, the student's work was assessed in a face-to-face manner right after the submission deadline, thus giving the student an immediate, albeit a severely time-limited feedback. (On average, we had only two minutes per student.)
Unfortunately, our old assessment system also suffered from a number of disadvantages. The greatest problem was that a lot of effort was required to prepare comprehensive and easily upgradeable homework assignments and diverse upgrades. The teaching staff for the programming course strove to prepare a set of mutually distinct homework assignments for each of the three grading weeks in a semester and for each lab session group so that each group of students could have a unique pair of a homework assignment and a set of three upgrades. The students were typically divided into 9 groups, which means that, in principle, as many as 9 distinct assignments and 27 distinct upgrades had to be prepared for each of the three grading weeks during the course. In practice, the number of distinct homework assignments was somewhat lower (usually, we prepared 5–6 distinct assignments per grading week), but then the number of distinct upgrade sets per assignment had to be even larger. The uniqueness of the upgrade sets was a means to fight against academic dishonesty [13], [14], but the effort required to ensure it was tremendous. Besides that, it was often difficult to prepare a balanced set of assignments and upgrades for different groups, and the students might have had a different view of what ‘balanced’ is than the teaching staff.

Since the assistants could, on average, devote only two minutes to each student, the grading procedure was highly superficial: run the program once or perhaps twice, take a quick look at the program code, exchange a few words with the student, and decide on the grade within the scale of 0–10. Such a grading approach could hardly be considered optimal. Not only that it assesses the student’s work on the basis of shallow, first-impression ‘metrics’, but it also features some typical human-factor problems such as the halo effect and disagreements between different assessors (teaching assistants) about the relative importance of individual aspects of the student’s work.

IV. INTRODUCTION OF AUTOMATED ASSESSMENT

A. Objectives

After a few years of experience, the disadvantages of our grading system have made themselves increasingly felt. Besides that, we started to participate in the organization of national programming competitions, where automated assessment of programming assignments had already been in effect. Both factors served as an incentive to introduce an automated assessment system. Thereby, we hoped that we could achieve the following goals:

1) Increase the objectivity of the assessment process.
2) Increase the amount of ‘productive’ time spent with the students.
3) Impart on the students the importance of creating correct programs according to a given specification, and of verifying them through a comprehensive set of tests, without actually naming concepts such as ‘unit test’, ‘test-driven development’, etc.
4) Reduce the excessive burden placed on the teaching assistants, and enable them to invest their time into the improvement of lab sessions, homework assignments (with a fewer number of distinct assignments, we can devote more time to each of them), and the grading process itself.

Whether, and to what extent, we have achieved these objectives will be discussed in Section V. Before that, we shall describe our assessment system in detail.

B. Overview of the assessment system

By the introduction of our new grading system, we abolished the upgrades and the distinct sets of assignments for each grading week. The entire body of students received the same set of assignments. This decision made it possible to increase the number of assignments to 10, approximately one per week. On the one hand, weekly assignments encourage the students to practice on a regular basis, while on the other, the relative importance of each individual assignment is lower than in our old system.

After the teaching assistants prepare an assignment, the students have to submit their solution within 9 days (between Friday and next Sunday) to the Moodle online classroom, which, at UL-FCIS, has been successfully used for several years. To facilitate the automated assessment, the names of the students’ programs (Java classes) have to follow a strict naming scheme. For each assignment, a student might earn a maximum of 10 points, giving 100 possible points in total. The points received from the assignments contribute one half to the final grade; the other half is earned on the final exam.

All assignments are assessed automatically. The teaching assistants prepare the test cases, feed the students’ submissions and the test cases to the system, and wait for the results. The necessary (but not sufficient) condition for a student’s program to receive any points at all is that it successfully compiles. If it does, it is evaluated by a number of test cases. The proportion of successfully passed test cases determines the grade. For example, if the student’s program passes 34 test cases out of 50, he or she will receive \((34/50) \times 10 = 6.8\) points. (In 2014/15, the points were actually rounded up to the nearest integer. In the future, however, we shall refrain from rounding and work with fractional points.)

For each assignment, the students were given 10 so-called public test cases, which were divided into classes of increasing difficulty. One of the assignments, for instance, required that the students write a program for counting the blobs (4-connected contiguous areas of ones) in a given binary (zero-one) matrix. The public test cases for this assignment were divided into 6 classes:

- In the test cases 1–4, every blob in the input matrix consisted of a single cell, and so a program that simply counts the number of ones in the matrix would pass all four tests.
- In the test cases 5–6, blobs could also take the form of horizontal or vertical lines.
- In the test case 7, blobs could also take the form of full rectangles. Still, no general flood-fill algorithm was required for this case.
• In the test case 8, every line of each blob consisted of a single consecutive sequence of ones. Even in this case, the students could do without performing flood-fill.
• In the test case 9, blobs could take an arbitrary shape. However, they were comparably small so that a recursive flood-fill algorithm would not run out of stack space.
• In the test case 10, blobs could take an arbitrary shape and could fill up the entire 1000-by-1000 matrix, which means that — owing to Java’s default stack limitations — only an iterative flood-fill algorithm was guaranteed to work in this case.

The public test cases are called ‘public’ because every student can view them and test his or her program on them. However, the programs are graded on a separate set of 50 hidden test cases, which are made publicly available only after the grading process for the current assignment has finished. The hidden test cases are guaranteed to be divided into the same number of classes as the public ones, and in each class, the number of hidden test cases is proportional to the number of public ones. In the example given above, the hidden test cases 1–20 belonged to the first class, the test cases 21–30 to the second, the test cases 31–35 to the third, etc.

By dividing the test case set into different classes, we encourage the students to develop their programs in a stepwise manner; especially a beginner is expected to start with the first class (matrices with single-cell blobs in the blob-counting example), then move up to the second, etc. Despite the diversity of the public test cases, they are, of course, not guaranteed to cover all possible boundary cases. Some boundary cases might be deliberately covered only by the hidden test cases. The students are therefore warned not to rely on the public test cases exclusively; they are advised to write additional test cases on their own.

Having the test cases divided into classes of increasing difficulty serves another purpose: to make the students think about the efficiency of their programs, without actually operating with the $O(.)$ or $\Theta(.)$ notation. Although computational complexity belongs to a course on algorithms and data structures rather than to a basic programming course, the earlier the students become aware of the fact that efficiency matters, the better.

### C. Input-output assessment

For a majority of the assignments, including the blob-counting one presented in Section IV-B, each public and hidden test case is divided into a test input and the expected output for that input. The program reads its input from the standard input and writes its output to the standard output, and the grading system compares, for each hidden test case, the produced output with the expected one. In the text of the programming assignment, we provide the exact format of both the input and the output, as well as the bounds of individual components of the input. For instance, we might state that the first line of the input contains an integer $a \in [1, 10^9]$, a space, and an integer $b \in [a, 10^9]$. To enable the students to focus on the programming problem rather than on the output format, the expected output is, in most assignments, composed of a single integer. We use real numbers only when absolutely necessary.

### D. Assessment through test classes

In some assignments, the students have to write a set of Java classes containing a set of methods with predefined signatures. In such cases, each public and hidden test case consists of a test class and the corresponding expected output. The test class creates a set of objects of the classes written by the student, invokes one or more methods of them, and prints the results. The results are then compared to the corresponding expected output. Again, to be able to pass any test case at all, the student’s classes have to compile successfully.

In one of the assignments of this type, the students had to write a pair of classes (Point and Line) to represent points and lines in the two-dimensional space. Each class had to contain a number of methods. Here are three of them:

- **double distance(Point p)** in the class Point: Returns the distance between the point $p$ and this point.
- **Point projection(Point p)** in the class Line: Returns the orthogonal projection of the point $p$ onto this line.
- **double distance(Point p)** in the class Line: Returns the distance between the point $p$ and this line.

This was the only assignment in which the grade was not obtained exclusively by black-box testing. In addition to the number of correct method call results, we also considered the ‘elegance’ of individual methods, which contributed up to $20\%$ of the grade. We defined (and made it known to the students) that a method is elegant if it calls other methods in a meaningful way. For example, the most elegant way to calculate the distance between a line $l$ and a point $P$ is to determine the orthogonal projection $P'$ of $P$ onto $l$ (by calling the method projection) and calculate the distance between $P$ and $P'$ (by calling the first method distance). The orthogonal projection of a point $P$ onto a line $l$ can itself be found in an ‘elegant’ way: as the intersection between $l$ and the orthogonal line passing through $P$ (both operations had to be programmed as methods, too). Since, for this assignment, we could uniquely define a canonical graph of method calls, the elegance was determined simply by comparing the canonical graph with the graph implicitly defined by the student’s program. We counted the number of edges present in the program’s graph and obtained an objective and meaningful measure of elegance.

### E. Assessment of computer graphics assignments

The goal of a computer graphics assignment is to write a program that produces a bitmap based on given data using the Java graphics framework. The bitmap in Fig. 1, for example, is based on an array representing the relative heights of the bars. In such assignments, each test case is composed of a test class and the corresponding bitmap file. The test class calls the student-supplied drawing method, passing to it the desired width and height of the bitmap and the underlying data for producing the bitmap. The bitmap created by that method is...
then compared against the reference bitmap on a pixel-by-pixel basis. To allow for rounding-off errors, the grading system considers that the two bitmaps match at a given position \((i, j)\) if for the pixel at \((i, j)\) in the reference bitmap there exists a same-colored pixel within the square \((i - 1, j - 1), \ldots, (i + 1, j + 1)\) in the program-created bitmap, and vice versa. The points contributed by individual pairs of matching pixels are inversely weighted by the total number of pixels of that color, since we want to ensure that a short thin blue line counts exactly the same as a large red-filled rectangle. To be able to compare the bitmaps on an element-by-element basis, every logical element (or set of elements) of the bitmap has to be painted with a distinct color. For example, the bitmap in Fig. 1 consists of four sets of elements: orange bar fillings, red bar borders, blue connecting line, and white background. Each set of elements contributes 25% to the grade.

![Fig. 1: The expected output for a sample assignment.](image)

In addition to producing an output bitmap, the student might have to write additional methods for computing the dimensions or offsets of certain elements based on the bitmap size. The student might thus receive some points for understanding basic coordinate-system transformations, even if he or she fails to produce a bitmap. In the example of Fig. 1, the student could be requested to write a method to compute the height of a given bar based on the bitmap dimensions and the corresponding data value.

**F. Plagiarism**

To detect plagiarism, we employ the Stanford University Moss system, which quickly and fairly reliably points out potential occurrences of plagiarism even in the face of substantial code changes. Unfortunately, as many as 75 (out of 279) students were caught at least once, either as plagiarists or as overly ‘generous’ authors. The actual number of plagiarism occurrences was probably even higher, but in several borderline cases we decided against a punishment. One of the reasons for a high number of offenders was the mild penalty: zero points for each plagiarized assignment, with no additional penalty for repeated offenses.

**G. Time and memory controls, security, etc.**

For each test case, the program is given 3 seconds to produce an output. After that period, the Linux `timeout` command forcibly terminates the program and considers that test case to be failed. Memory consumption is bounded by the Java virtual machine itself. Security is ensured by running the programs under a designated non-privileged user account on a Linux machine.

**H. User interface**

The students obtain a command-line tool to grade their programs against a given set of test cases. This simple-to-use tool supports all three modes of testing (input-output, test classes, and bitmap-vs-bitmap). The tool presents the test results in form of an HTML report, which can be viewed in a browser. Both in the output of the command-line tool and in the HTML report, different possible test outcomes (pass, fail, timeout, exception) are represented by different colors. For each test case, the report shows both the input (or the test class) and the expected output; in the case of a mismatch, the program-produced output is shown as well. If a test case triggers an exception, the accompanying message is fully displayed. The student can thus quickly spot the test cases that he or she still has to work on. Figure 2 displays a sample run of the command-line tool and the corresponding HTML report.

**V. DISCUSSION AND CONCLUSION**

Have we fulfilled the objectives stated in Section IV-A? First, the assessment process is now undoubtedly more objective than it used to be. The computer grader is immune to the halo effect and other human-related imperfections. Second, the amount of ‘productive’ time spent with students has also increased: before the year 2014/15, three out of 13 lab session weeks were completely devoted to the assessment of assignments (through upgrades). Now, we are able to devote all 13 weeks to solving sample programming assignments, answering student questions, etc. The third objective was constantly being addressed through the rigorous definitions of input and output, through a diverse and imperfect set of public test cases (which covered a significant ‘area’ of the space of possible inputs but deliberately not every boundary case), by encouraging the students to write additional test cases themselves, and by the ‘ruthless’ automated grading system itself, in which only the number of passed test cases count. As for our fourth objective, however, we have to admit that the assistants might have had even more work than before. However, a large amount of time was devoted to the construction of the system itself; to the conversion of our old lab session programming assignments and more than 60 extra assignments (intended for ungraded practice) to a new format, and to the preparation of the corresponding test case sets; to the preparation of course material for three extra lab session weeks; to the renovation of the lab sessions themselves (the assistants shifted further towards the ‘guide on the side’ principle, while still functioning as a ‘sage on the stage’ when presenting the solutions to the assignments to the entire classroom); and, last but not least, to plagiarism detection and highly unpleasant email ‘negotiations’ with the
While the test cases themselves cannot check adherence to coding standards or ‘elegance’, these concepts could, to a certain degree, be checked as well. In Section IV-D, we showed an example where ‘elegance’ was translated into a well-defined concept that could be easily checked. In the future, we shall investigate other ways to extend the system beyond simple pass/fail checks.

Despite certain imperfections, we consider the introduction of automated assessment to be a step forward in our pedagogical process. We have not yet conducted a survey about the students’ views on automated assessment, but judging from the general anonymous remarks that we received at the end of the semester, the students seem to be satisfied with the new system. For the next year, we plan a more thorough preparation of the students to the grading system. In the lab sessions, we shall consistently use the system from the very beginning, and we shall introduce an ungraded pre-assignment to enable the students to become fully familiar with the system before the actual grading takes place.

REFERENCES


