3-2015

An Empirical Study of Iterative Improvement in Programming Assignments

Raymond Pettit

John Homer

Roger Gee

Adam Starbuck

Susan Mengel

An Empirical Study of Iterative Improvement in Programming Assignments

Raymond Pettit, John Homer, Roger Gee, and Adam Starbuck
School of Information Technology and Computing
Abilene Christian University
Abilene, TX, USA
{rsp05b, jdh08a, rpg11a, acs11e}@acu.edu

Susan Mengel
Department of Computer Science
Texas Tech University
Lubbock, TX, USA
susan.mengel@ttu.edu

ABSTRACT
As automated tools for grading programming assignments become more widely used, it is imperative that we better understand how students are utilizing them. Other researchers have provided helpful data on the role automated assessment tools (AATs) have played in the classroom. In order to investigate improved practices in using AATs for student learning, we sought to better understand *how* students iteratively modify their programs toward a solution by analyzing more than 45,000 student submissions over 7 semesters in an introductory (CS1) programming course. The resulting metrics allowed us to study what steps students took toward solutions for programming assignments. This paper considers the incremental changes students make and the correlating score between sequential submissions, measured by metrics including source lines of code, cyclomatic (McCabe) complexity, state space, and the 6 Halstead measures of complexity of the program. We demonstrate the value of throttling and show that generating software metrics for analysis can serve to help instructors better guide student learning.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computers and Information Science Education – *computer science education, information systems education, self-assessment*.

General Terms
Experimentation

Keywords
computer science education; computer aided instruction; automated feedback; automated assessment tools

SIGCSE '15, March 4–7, 2015, Kansas City, MO, USA

ACM 978-1-4503-2966-8/15/03…$15.00
http://dx.doi.org/10.1145/2676723.2677279

1. INTRODUCTION
One challenge facing instructors of computer programming is identifying how students learn the material through hands-on practice that takes place beyond the instructor's observation, often outside the classroom.

A better understanding of how students iteratively modify their programs toward a solution can help us improve programming instruction over time. Examining a final submission can determine if a student eventually created a properly working program but does not indicate how efficiently or systematically he or she approached the problem.

This paper examines data collected from more than 45,000 student submissions over 7 semesters in an introductory (CS1) programming course. Each semester, we gave students 75 C++ programming assignments, increasing in difficulty over the course of the semester. Students submitted their work to an online system, Athene (created in-house), which stored, compiled, and ran each submission against a suite of specific test cases and carefully generated random test cases. The Athene system then automatically scored each submission, so students received feedback immediately. Programs that did not pass the entire test suite received either a compile error message or information about one or more failed test cases as feedback. For failed test cases, students received a report of their input, the expected output, and the actual output from their program. Students had the opportunity to modify and resubmit their program in an attempt to improve their grade. A student session consisted of all submissions by one student for a given problem. There was no limit on the number of submissions that a student could attempt until the deadline, although we experimented with "throttling," a technique that limited a student's attempted solutions within a rolling 15-minute time period. This paper also considers the effects of this throttling on submission behavior.

With this approach, we intend to encourage students to complete each assignment, overcoming early errors to eventually reach a final correct solution. Automated scoring is made possible by preprocessing and other source modifications, then compiling and linking with both standard and specialized libraries. The system scores student submissions by direct inspection of required features (such as functions) and validation of program output.

We look more closely at two key moments in each student session in order to see the amount of change and meaningful progress that takes place (1) between a student's first and second submissions and (2) between a student's first and last submissions. In our courses, 83% of student sessions reach a score of 100%.

2. RELATED WORK

Automated assessment tools (AAT) have been in use since 1960 [6]. Ala-Mutka [3], Douce [5], and Ihantola [7] all offer good reviews of the capabilities of tools that have come into existence since. Other relevant tools include ALOHA [1], Style++ [2], JUG [4], BlueJ [8], Autograder [9], ClockIt [10], Fitchfork [11], Mooshak [12], Bottlenose [13], AutoLep [14], and AutogradeMe [15]. We provide a more specific review of selected research below, as comparable examples of automated assessment tools (AATs) provide important context in considering our Athene AAT.

In June 2005, Kirsti M. Ala-Mutka surveyed a range of assessment tools available at that time, reviewing their abilities in dynamic testing to construct a secure running environment and check program functionality, efficiency, and student testing skills; and in static testing to check coding style and programming errors, collect software metrics, and assess design [3]. In September 2005, Christopher Douce, et al., provided a historical overview of the development of AATs and a survey of related research, concluding with a series of useful criteria for evaluating such tools, including whether the system "does what it is supposed to do," whether "it is liked by its users," and whether "it helps students become more proficient at programming" [5]. In 2010, Petri Ihantola, et al., followed up with a review of AAT development between 2006–2010, determining that the most significant differences among programs had to do with "how tests are defined, how resubmissions are handled, and how the security is guaranteed" [7].

Since the publication of these surveys, several key papers have devoted expanded attention to more recent programs. For example, Mark Sherman, et al., looked at the web-based Bottlenose framework and considered how students responded to instant automated feedback in contrast to time-delayed instructor feedback, finding that students made 50% more submissions per assignment when using the AAT. Sherman's article does not discuss final submission quality with or without the AAT, but does note the straightforward effect that with the AAT, students continue to modify and re-submit assignments, presumably in response to the system's feedback—an advantage in a course where they might otherwise submit several assignments before receiving instructor feedback [13]. The article does not go into detail regarding what Bottlenose assesses in providing feedback; for examples of how such programs work, we can look at papers including those from Manuel Rubio-Sánchez, et al. [12] and Tiantian Wang, et al. [14].

Rubio-Sánchez's group reviewed the widely available Mooshak system, using both qualitative and quantitative analysis to evaluate its effectiveness in student learning. Responding to earlier studies that had noted a negative correlation between the introduction of Mooshak and student drop-out rates, Rubio-Sánchez's group observed that those studies had not held other variables constant—in particular, the studies had changed teaching methodology simultaneously with introducing Mooshak. In their study, then, Rubio-Sánchez's group included both test and control groups in the form of courses with near-identical syllabi and teaching methodologies, some of which used Mooshak and some of which did not. Qualitatively, students self-reported appreciating instant feedback but had complaints about Mooshak specifically, because while it is effective in assessing whether a program has succeeded or failed, it does not provide feedback to help students make changes along the way. The Mooshak tool was originally created for use in programming contests and still lacks features present in many tools designed for use in courses. Holding other factors constant, Rubio-Sánchez's group did not find a statistical change in the dropout rate of courses using Mooshak.

In contrast, Wang's group wrote about AutoLEP, an AAT they developed at the Harbin Institute of Technology in Heilongjiang, China, which combines static analysis with dynamic testing to provide students with location-specific feedback on the syntactic, structural, and logical features in their programs. AutoLEP seems more like Bottlenose in allowing for feedback and multiple submissions, but with the apparent advantage of more precision due to its simultaneous static and dynamic feedback.

The issue of whether AATs ought to allow for multiple submissions comes up again in Vrada Pieterse's paper on the use of AATs—in particular, his group's Fitchfork software—in massive open online courses (MOOCs) that teach programming. Pieterse argues that throttling (limiting the number of submissions per assignment) is inappropriate in a MOOC environment in part because the open enrollment format means students are working voluntarily and should have the opportunity to use AATs for repeated revisions as needed to learn the material. A correlative argument, however, is that unlimited submissions carries a risk to the traditional, credit-based classroom where students may be more tempted to "game the system" in an attempt to get a desired grade rather than to master learning the material [11]. Such a conjecture corresponds to an observation by Ihantola's group, who write, "we believe that the very fact that the assessment is automatic is likely to change how some students approach the exercise. Knowingly submitting a weak or even incorrect solution that gets accepted by a machine is quite likely more socially acceptable than trying to cheat a person." Pieterse's paper, then, raises the challenge to AAT developers to consider an appropriate level of throttling for the system's target user environment.

Overall, these articles point to a series of features relevant in developing AATs, including secure running environments (sandboxes), static and dynamic testing, and resubmissions and throttling. In our Athene program, we chose to give students feedback primarily from dynamic testing while running static analysis later for examining its usefulness. We also implemented a throttling rule for several semesters to see how that changed student behavior.

3. METHODOLOGY

3.1 Population Characterization

We collected data from 290 students in a Programming I course over seven semesters. We taught the course using the C++ programming language. Our curriculum is a late objects curriculum and subject matter in the course is typical of a CS1 course, including input/output, basic data types, decision structures, repetition, functions, and arrays.

The course serves primarily as a first programming course required of Computer Science majors, but students also include majors in Engineering, Physics, Mathematics, Information Technology, and other related disciplines.

3.2 Description of Data Collected

Each semester, we give students 75 programming assignments for homework, most of which are completed outside of class. The data included in this paper comes from all 75 unique assignments.

The students receive their assignments through the Athene online automated system.

Figure 1 shows a representative assignment, typically assigned in the third week of a 15-week semester, shortly after introducing decision structures. In this case, students are assigned to write a console-based program that asks the user to enter 3 integers and returns the largest of the 3. The goal of this assignment is to give the students practice in using `if-else` statements. As with all Athene assignments, the student is given a problem description, along with at least 1 test case and the expected output for that test case.

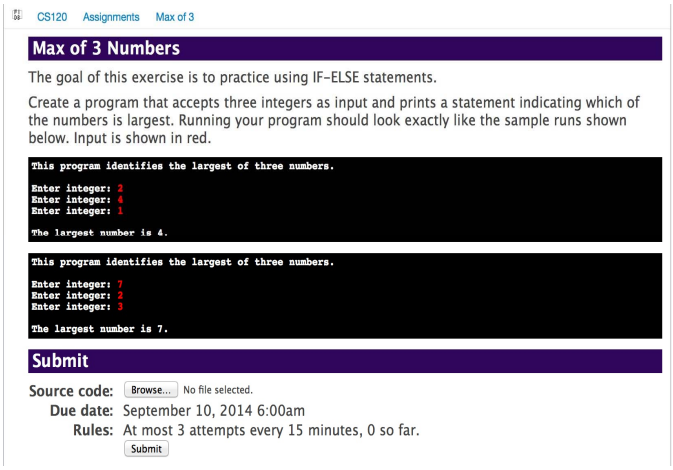

Figure 1. Representative assignment as it appears in the Athene online automated system.

For each assignment, a student writes a program and submits the source code to the Athene system, which checks grading and provides feedback. The student may re-submit a program repeatedly until he or she has successfully written the code.

After the student submits a source code file, the Athene system immediately compiles, runs, and tests the program against established test cases to provide a response. Figures 2 through 5 show a series of representative submissions and their corresponding feedback, all from the same student session in attempting to solve the assignment shown in Figure 1.

Each time a student submits an assignment, the automated system records the following information:

- user id
- filename of the submitted source code
- time/date of the submission
- full source code submitted
- score
- input and expected output for the first failed case
- actual output of the first failed test case

From the data that is stored in the submission database, we can later go back and analyze summary data of student behavior.

Examining the source code of a single submission, we calculate:

- the number of source lines of code (SLOC)
- the state space (number of unique variables)
- the cyclomatic (McCabe) complexity of the program
- the 6 Halstead measures of complexity of the program

We compute SLOC by counting all source lines, then deleting all blank lines and comment lines. We compute the McCabe complexity by adding the total number of branch possibilities (if, for, while, and case statements, adding in short-circuit analysis of boolean conditionals) to the total number of functions defined.

The 6 Halstead complexity measures are: Vocabulary, Length, Computed Length, Volume, Difficulty, and Effort. The Halstead numbers are computed by counting the number of unique and total operators and operands and using them in the appropriate Halstead formulas.

Figure 2 shows feedback given to a particular student after an early attempt at solving the assignment shown in Figure 1. This feedback is displayed almost instantaneously after the student submits a source code file. The top of the Athene page displays the student's ID, submission time, score achieved, course in which he or she was enrolled, and the assignment name. The middle section of the feedback page displays the first expected output line the student's submission failed to produce, followed by the actual output that the student's submission did produce. The bottom section of the feedback page shows the contents of the student's submitted source code file.

If we examine the source code, we can see that the student wrote a program that would perform correctly for both examples given in the problem description shown in Figure 1. But this student didn't consider different types of test cases, such as two of the three input numbers being the same value. So the submitted source file show in Figure 2 passed many of the test cases and was awarded a score of 64 (out of 100), but failed for the first time when the test input was `5 9 9`. The student received the feedback message, "`expected output: The largest number is 9`," and could then review to see that his or her actual output did not contain that statement.

Figure 3 shows the student's next attempt at the assignment. The student added some additional `if` statements to catch the test case he or she had just missed but still did not cover all possible test cases. [Also worth noting, the student should have corrected his or her existing `if` statements, instead of adding more of them. The Athene system did not provide feedback on this point.]

Figure 4 shows the student's third attempt at the assignment. The student once again added some additional `if` statements to catch the missed test case, but still did not cover all possible test cases. Because of the additional test cases passed, the student achieved a score of 88.

Figure 5 shows the student's fourth attempt at the assignment. To catch the test case in which all input numbers are the same value, the student added an additional `if` statement checking specifically for that case. With that addition, all test cases were successfully passed and the student received a score of 100, although the code is overly complex.

Reviewing this assignment and the given student's 4 submissions, we can see that the student eventually turned in a solution that produced the correct output but was not written to be efficient. The metrics for the source code from each of these submissions is shown in Table 1. Looking at metrics, specifically McCabe, can tell an instructor a great deal about the student's solution. For this assignment, the expected McCabe value is 7. The function counts for 1 and each `if` statement counts for 2, given the possibility of short circuit evaluation for each. When an instructor sees a McCabe value of 17 for this assignment, the instructor can recognize that the student has not created the expected solution.

User: [redacted]
Time: 2013-02-11 22:13:11
Score: 64
Class: 1320_CS12001
Problem: 03-Max_of_3

expected output:

```
The largest number is 9.
```

actual output:

```
This program identifies the largest of three numbers.

Enter integer: 5
Enter integer: 9
Enter integer: 9
```

max3.cpp

```
#include <iostream>
using namespace std;

int main()
{
        int x, y, z;

        cout << "This program identifies the largest of three numbers." << endl << endl;

    cout << "Enter integer: ";
    cin >> x;
    cout << "Enter integer: ";
    cin >> y;
    cout << "Enter integer: ";
    cin >> z;
    cout << endl;

    if (x > y && x > z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y > z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z > y)
    {
        cout << "The largest number is " << z << ".";
    }
}
```

Figure 2. Representative first submission with Athene feedback.

User: [redacted]
Time: 2013-02-11 22:22:29
Score: 76
Class: 1320_CS12001
Problem: 03-Max_of_3

expected output:

```
The largest number is 13.
```

actual output:

```
This program identifies the largest of three numbers.

Enter integer: 13
Enter integer: 13
Enter integer: 2
```

max3.cpp

```
#include <iostream>
using namespace std;

int main()
{
        int x, y, z;

        cout << "This program identifies the largest of three numbers." << endl << endl;

    cout << "Enter integer: ";
    cin >> x;
    cout << "Enter integer: ";
    cin >> y;
    cout << "Enter integer: ";
    cin >> z;
    cout << endl;

    if (x > y && x > z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y > z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z > y)
    {
        cout << "The largest number is " << z << ".";
    }
    if (x > y && x == z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y == z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z == y)
    {
        cout << "The largest number is " << z << ".";
    }
}
```

Figure 3. Representative second submission with Athene feedback.

User: [redacted]
Time: 2013-02-11 22:27:00
Score: 88
Class: 1320_CS12001
Problem: 03-Max_of_3

expected output:

```
The largest number is 43.
```

actual output:

```
This program identifies the largest of three numbers.

Enter integer: 43
Enter integer: 43
Enter integer: 43
```

max3.cpp

```
#include <iostream>
using namespace std;

int main()
{
        int x, y, z;

        cout << "This program identifies the largest of three numbers." << endl << endl;

    cout << "Enter integer: ";
    cin >> x;
    cout << "Enter integer: ";
    cin >> y;
    cout << "Enter integer: ";
    cin >> z;
    cout << endl;

    if (x > y && x > z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y > z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z > y)
    {
        cout << "The largest number is " << z << ".";
    }
    if (x > y && x == z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y == z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z == y)
    {
        cout << "The largest number is " << z << ".";
    }
    else if (x > z && x == y)
    {
        cout << "The largest number is " << x << ".";
    }
}
```

Figure 4. Representative third submission with Athene feedback.

User: [redacted]
Time: 2013-02-11 22:29:42
Score: 100
Class: 1320_CS12001
Problem: 03-Max_of_3

Success.

max3.cpp

```
#include <iostream>
using namespace std;

int main()
{
        int x, y, z;

        cout << "This program identifies the largest of three numbers." << endl << endl;

    cout << "Enter integer: ";
    cin >> x;
    cout << "Enter integer: ";
    cin >> y;
    cout << "Enter integer: ";
    cin >> z;
    cout << endl;

    if (x > y && x > z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y > z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z > y)
    {
        cout << "The largest number is " << z << ".";
    }
    if (x > y && x == z)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (y > x && y == z)
    {
        cout << "The largest number is " << y << ".";

    }
    else if (z > x && z == y)
    {
        cout << "The largest number is " << z << ".";
    }
    else if (x > z && x == y)
    {
        cout << "The largest number is " << x << ".";
    }
    else if (x == y && x == z)
    {
        cout << "The largest number is " << x << ".";
    }
}
```

Figure 5. Representative fourth submission with Athene feedback.

Table 1. Source code metrics for example submissions

	1st Submission (Figure 2.)	2nd Submission (Figure 3.)	3rd Submission (Figure 4.)	4th Submission (Figure 5.)
SLOC	17	23	25	27
State Space	3	3	3	3
McCabe Complexity	7	13	15	17
Halstead Vocabulary	28	35	37	39
Halstead Length	95	145	162	179
Halstead Computed Length	108.28	148.09	160.27	172.66
Halstead Volume	456.1	743.75	843.93	946.09
Halstead Difficulty	16.07	16.27	16.71	17.09
Halstead Effort	7976.46	12101.37	14103.39	16167.95

4. RESULTS

Table 2 describes the overall data that was analyzed in this paper.

Table 2. Overview of data collected.

Total # of Semesters	7
Total # of Students	290
Total # of Submissions	45,128
Total # of Sessions	12,452
Percent of Sessions Completed Successfully	83.2%

In analyzing the data, we gave extra attention to factors that changed when students showed positive progress. Table 3 represents the data from only those submissions that come out of multiple-attempt sessions, and we always ignored the first attempt (as there would not be a previous submission to compare it against). We call these submissions "new maximums." 32.2% of eligible submissions were new maximums.

Table 3. Average changes students made in achieving a new maximum submission.

Change from Last Submission	New Max
Average score increase for a new max	57
Average SLOC increase	0.43
Average State Space increase	0.04
Average McCabe Increase	0.33
Average Halstead Vocabulary increase	0.73

Also of special interest to us is the effect that throttling had on what students changed from submission to submission. For semesters 1–4, we allowed students unlimited submissions in any time period without throttling; for semesters 5–7, we established a throttle that limited students to 3 submissions per 15-minute period. This action corresponded with distinctions in student behavior and data outcomes. This data is shown in Table 4.

5. DISCUSSION

We recognize two primary areas of new knowledge emerging from this study. First, we see that throttling of submissions does indeed have an impact on the quality of student submissions. Table 4 shows that the average score of multiple attempt sessions that eventually scores 100% increased from 11 all the way to 28. Knowing that submissions were throttled made students more careful in making their first submission, hopefully putting more thought into their work and doing independent testing instead of only relying on the grading system.

Second, dynamic testing is important, but instructors and students can also benefit from considering style and content. In January 2004, Ala-Mutka published an executive study on the use of Style++ to promote good style practices in students. The AAT was able to discern and respond to a number of unhealthy programming practices with an appropriate grade and feedback on how the student should improve the efficiency of their program. Ala-Mutka found that "students implement more reliable and understandable programs" after having only been required to submit assignments to Style++ for a year [2]. However, Ala-Mutka's study focused on independent student use of the Style++ tool in advance of submitting final assignments, allowing instructors to "concentrate on giving feedback on the more advanced features of program design and course specific issues."

We argue that the examples in the session shown in Figures 2 through 5 show us that instructors can learn a great deal more about student submission by employing some basic metric analysis of submitted code. By using these other types of analysis, we can identify gaps in understanding, even when a student finishes with a score of 100%. With a focus on style, the student may be more capable of thinking in terms of efficiency and efficacy for each line in their code, which can help prevent situations similar to that in Figures 2-5 wherein the student needlessly increased complexity and length of code rather than rewriting existing code to achieve the desired output.

Overall, we can infer that both an emphasis on technique and use of throttling submissions encourage a reflective perspective of one's work.

6. FUTURE WORK

In the future, we would like to more seamlessly integrate static analysis tools—giving students more feedback (such as reporting to them the actual complexity level of their submitted program and the expected level).

Another interesting project would be to automatically analyze individual problems to identify the most common student problems, so instructors can address these issues more effectively in class.

Table 4. Effect of throttling on submissions.

	All Semesters	No Throttling (Semesters 1-4)	With Throttling (Semesters 5-7)
Total students	290	170	120
Total submissions	45,128	31,753	13,375
Total sessions	12,452	7,949	4,503
Average submissions per session	3.62	3.99	2.97
Average score of first submission for sessions with multiple attempts that eventually scored 100%	19	11	28
Average score change from first to second submission	30	31	30
Average McCabe change from first to last submission	0.69	0.81	0.58
Average Halstead Vocabulary change from first to last submission	2.34	2.81	1.87

7. ACKNOWLEDGMENTS

Special thanks to Heidi Nobles and Kayla Holcomb for their editorial support in preparing this manuscript.

8. REFERENCES

[1] Ahoniemi, T., and Reinikainen, T. ALOHA - a grading tool for semi-automatic assessment of mass programming courses. In *Proceedings of the 6th Baltic Sea Conference on Computing education research* (Baltic Sea, 2006). Koli Calling ’06. ACM, New York, NY, 139-140.

[2] Ala-Mutka, K. M., Uimonen, T., and Järvinen, H. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education* 3,1 (2004), 245-262.

[3] Ala-Mutka, K. M. A survey of automated assessment approaches for programming assignments. *Computer Science Education* 15,2 (2005), 83-102.

[4] Brown, C., Pastel, R., Siever, B., and Earnest, J. JUG: a JUnit generation, time complexity analysis and reporting tool to streamline grading. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (2012). ITiCSE ’12. ACM, New York, NY, 99-104.

[5] Douce, C. Livingstone, D., and Orwell, J. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing* 5,3 (September 2005), Article 4.

[6] Hollingsworth, J. Automatic graders for programming classes. *Communications of the ACM* 3,10 (October 1960), 528-529.

[7] Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (2010.) Koli Calling ’10. ACM, New York, NY, USA, 86-93.

[8] Jadud, M. C. A first look at novice compilation behaviour using BlueJ. *Computer Science Education* 15,1 (2005), 25-40.

[9] Nordquist, P. Providing accurate and timely feedback by automatically grading student programming labs. *Journal of Computer Science Coll.* 23,2 (December 2007), 16-23.

[10] Norris, C., Barry, F., Fenwick, J. B., Jr., Reid, K., and Rountree, J. ClockIt: collecting quantitative data on how beginning software developers really work. *SIGCSE Bull.* 40,3 (June 2008), 37-41.

[11] Pieterse, V. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research* (2013). CSERC ’13. M. van Eekelen, e. Barendsen, P. Sloep, and G. van der Veer, Eds. Open Universiteit, Heerlen, The Netherlands, Article 4.

[12] Rubio-Sánchez, M., Kinnunen, P., Pareja-Flores, C., and Velázquez-Iturbide, Á. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior* 31 (February 2014), 453-460.

[13] Sherman, M., Bassil, S., Lipman, D., Tuck, N., and Martin, F. Impact of auto-grading on an introductory computing course. *Journal Computing Sciences in Colleges* 28,6 (June 2013), 69-75.

[14] Wang, T., Su, X., Ma, P., Wang, Y., and Wang, K. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education* 56,1 (January 2011), 220-226.

[15] Zimmerman, D. M., Kiniry, J. R., and Fairmichael, F. Toward instant gradeification. In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training* (2011). CSEET ’11. IEEE Computer Society, Washington, DC, 406-410.