Characterization of CS1 Student Programming

Nigel Bosch, Dwayne Towell, John Homer
Abilene Christian University, Abilene, TX
{pnb08a, dwayne.towell, john.homer}@acu.edu

Abstract

This paper reports on a quantitative evaluation of data collected from student programming assignments in an introductory programming course, using 44,548 submissions by 192 students over six semesters. Student programs are gathered and automatically scored for correctness; students are permitted to repeatedly submit possible solutions until a specific deadline, and each submission is immediately scored and feedback given about correctness or any errors. We report on the process followed by students (total time spent per problem, number of submissions, rate of completion, etc.*) as well as the results (program length, complexity, state space). Our study reveals some common characteristics in "typical" CS1 student solutions and provides some reflection on the assignment framework.*

Conference FECS 2012

Keywords
Computer Science education, CS1, Learning patterns

1 Introduction

Computer science educators have a significant interest in improving the general understanding of how novice programmers acquire and improve upon basic computer programming skills. Introductory programming (CS1) courses apply varied approaches in the design and assessment of programming assignments, often differing in the number, size, submission policies, and degree of feedback given for assignments. In this paper, we present an evaluation of one programming assignment framework for a CS1 course and discuss the advantages and disadvantages of this approach.

This experience report describes the behavior of students in a CS1 course at Abilene Christian University. Utilizing metrics based on submitted source code, frequency of submission and other factors, we quantify and evaluate some student performance properties.

In our introductory programming class, the programming assignment framework requires students to submit their programs to an online system for automated scoring. The system uses several techniques to score the submission including direct inspection for required features (such as functions), preprocessing and other source modifications, compiling, linking with both standard and specialized libraries, and execution with controlled input. Any step in the scoring process which produces an error or unexpected result generates a report to the student with the details. Students receiving an error message are permitted to iteratively correct and resubmit their solutions until a correct solution is achieved or a predetermined deadline time is reached. The notable characteristics of this approach are the presentation of immediate feedback on either the success or cause of failure and the allowance for frequent and repeated iteration (bounded only by an end-time).

The intention of this programming assignment framework is to encourage students to complete each assignment, overcoming early errors and eventually reaching a final working solution. The CS1 course presents a large number (approximately 75) of assignments in which students are expected to write complete working C++ programs. In this way, students are expected to repeatedly practice and refine their programming skills, gradually increasing in experience and ability throughout the semester.

This paper reports on the analysis of the gathered data. We will address the success rate (eventual completion of programs by students), average number of attempts per problem, time delay between student submissions for the same problem, and the quality of programs produced by students in this approach. Related works are discussed in Section 2. Our data gathering and evaluation is described in Section 3; the results of our analysis are presented in Section 4 and some interpretation is given in Section 5. The paper concludes in Section 6 with a summary of our evaluation of this data and some thoughts about future study.

2 Related Works

Many educators share a common interest in understanding how students learns and what can be done to improve their learning and retention. The study of novice programmers (notably how they think about and create computer programs) has attracted much attention [1, 4, 12, 13]. In recent years, efforts have been made to automatically record and evaluate student-created solutions for programming assignments, and to analyze the behavior exhibited and stages of progress encountered by student programmers [2, 3, 6, 8, 9, 10, 11].

Jadud [6] studied the behavior of students in an introductory course. By logging compilations of student work, he was able to identify the error types most commonly encountered by students and analyze their responses when presented with error messages. He found that many (51%) student compilations were performed less than 30 seconds after the previous attempt; our data corroborates this rapid-response trend in student behavior and we present some analysis of the success rate for such behavior.

Various toolsets have been developed to automatically log and/or score student programs. Spacco, *et al.* introduced Marmoset [11], an automated system for student submission, and used this system to study the development process followed by students in programming assignments. Edwards and Perez-Quinones introduced Web-CAT [3] as a system that "grades students on how well they test their own code." Norris, *et al.* introduced ClockIt [10] to monitor student development practices, with the goal of identifying and encouraging successful practices. Murphy, *et al.* introduced Retina [8] to log and aggregate data about student progress, providing high-level reports (to both the instructor and the students) and specific feedback to each student.

Some previous work has focused on the value of feedback in assisting students toward a working solution. Hattie and Timperley [5] reviewed the general use of feedback in education, addressing the varying effects of specificity, timing, and frequency of feedback, and considering positive vs. negative feedback. Regarding timing of feedback, they note some indication of beneficial effects in delaying task-level feedback, but providing immediate process-level feedback. Our data shows that students make relatively poor use of immediate feedback on program correctness; although as educators we intend for programming assignments to help students to develop process-level skills, this behavior suggests that they see the assignments simply as tasks to be completed and respond to feedback accordingly.

Nordquist [9] introduces a submission system much like the one presented in this paper, with immediate feedback on correctness and an opportunity for repeated submissions; Nordquist's system also allows for students, having completed their own solution, to request and receive the instructor's solution. Edwards and Perez-Quinones [2] discuss the value of automated grading and address strategies for providing student feedback.

3 Methodology

We gathered data from and about programs that students created and submitted for CS1 courses offered over several semesters. Each course section is given approximately 75-80 programming assignments (in C/C++) to complete during the semester. Each assignment, or group of assignments, increases in difficulty, as new concepts are introduced by lecture, reading and examples; our curriculum emphasizes procedural skills early and introduces classes and objects in the third semester. The CS1 programming assignments are designed to be completed sequentially through the semester, utilizing skills and occasionally whole functions developed through previous work.

Students attempt to solve each problem by submitting source code to a system that timestamps, stores, compiles and runs their code to verify the correctness of their solution. Correctness is primarily determined by verifying that correct output is generated for specific test cases and carefully generated random test cases. If an error is discovered, either run time or logical, students receive immediate feedback about the failed test case: the given input, the expected output, and the actual output of their program. Students are allowed to repeatedly submit attempts for each assignment, as often and as many as they desire, until the specified deadline.

3.1 Process Metrics

Our policy permitting iteration allows us to investigate some aspects of the problem-solving process followed by students, such as number of attempts and time spent between attempts while working on a specific problem. In combination with computed scores for each submission we can measure the following aspects of each student's progress.

Work Sessions

A work session is the span of time in which the student is actively working on their solution to a specific problem. We assume that an elapsed time between

submissions of 30 minutes or less indicates that the student was actively working on their solution during that time. Thus a work session begins at some point before the first submission and continues as long as the previous submission was less than 30 minutes before. A gap of more than 30 minutes is taken to indicate that the student took a break, and the next submission was made after the start of a new session of work.

Iteration Interval

The iteration interval is the span of time between consecutive submissions within a work session. Iteration interval provides a quantitative measure of the step size employed by the student during problem solving. For our purposes here, individual submissions are grouped as occurring (relative to the student's previous submission) within one minute, between one and two minutes, between two and five minutes, between five and fifteen minutes, between fifteen and thirty minutes, or as the initial submission for a new session (more than thirty minutes after previous).

Work Time

Work time is the total number of minutes a student worked on a specific problem. An estimate of total work time for a student can be computed as the total of iteration intervals plus the time before the first submission of each work session. Under the session assumptions described above, total work time includes the iteration interval if the time is 30 minutes or less. For the initial work period and intervals of more than 30 minutes, we rely on adjusted self-reported data from the students to estimate these time spans. The mechanism for collecting this information is explained in the next section.

3.2 Survey

In addition to the computed metrics above, a survey was conducted to gather information about student perceptions and to help estimate total work time. As noted above, the initial work interval at the start of a session cannot be determined automatically and yet may be the bulk of the total work time. The survey asked the following questions:

```
1. Typically, how many minutes did you
work on a problem before making the first
submission?
2. Typically, how many minutes did you work
on a problem during one sitting?
3. Typically, after leaving a problem for
a time and coming back to it again later,
how many minutes did you work before making
another submission?
4. Typically, how many total minutes did you
spend on each problem?
5. Typically, how many submissions did you
make for each problem?
```

Questions 1 and 3 above are the core of this instrument, seeking to learn the "start up" time in a working session before submitting an attempted solution. Questions 2 and 4 can be used to check consistency with 1 and 3, since the iteration interval times computed automatically can be deducted, leaving time before the first submission and time before other sessions. Question 5, whose answer is known from gathered data, allows us to crudely measure the accuracy of student responses.

3.3 Source Code Metrics

The solution to each problem presented is source code of a working program able to complete the task described in the problem statement. Another avenue of evaluation for student submissions considers properties of the source code using common metrics. Solutions for each problem have been created by instructors and teaching assistants, although it is important to note that, while the "correct" solutions were created by knowledgable programmers, they do not necessarily represent the best possible implementation. These programs were primarily written to be clear and straightforward implementations. Additionally, they are written to reflect the solutions that can be reasonably expected to be within reach of the students, given their experience level and the course material covered prior to that point. Thus, these solutions provide a baseline against which student solutions can be measured and quantified. Since students are allowed and expected to iterate toward a working solution, we will consider only those submissions which successfully solve the problem.

Since our CS1 curriculum is primarily concerned with procedural concepts, we concentrate on "size" metrics as a surrogate for overall solution difficulty. We have chosen to utilize these measurements: source lines of code, McCabe cyclomatic complexity, and number of declared variables. Together, these metrics provide some indication of the difficulty level for a specific program.

Source Lines of Code

Source Lines of Code (SLOC) is the number of lines of code in a program after removing comments, character strings, and blank lines. Removing comments, character strings and blank lines is an attempt to reduce discrepancies between authors; intuitively, this provides a more accurate measurement of the meaningful source code. SLOC has traditionally been used as a rough measure of size and effort; it is included here

because it is easy to measure and has been widely used in the past. In addition (and anecdotally), SLOC appears to be used by the students themselves to assess the relative difficulty of a program.

Cyclomatic (McCabe) Complexity

Cyclomatic complexity [7] is a measurement of the number of linearly independent paths of execution in a program, a quantitative measurement of the logical complexity of a program. The easiest way to compute cyclomatic complexity is as the sum of the number of flow-control branches plus the number of separate functions in source code. Since student submissions are limited to C++, the only branches to take into account are conditional statements (`if`, `case`, and the `?:` tertiary operator) and looping constructs (`for`, `while`, and `do`). Cyclomatic complexity correlates with control-flow complexity and is used here as a measure of the cognitive load or difficulty of the problem in that sense.

Declared Variables

Declared variables count the variables of types common to CS1 programs (`int`, `float`, `double`, `bool`, `long`, `unsigned`, `char` and `string`) which are declared in the source code. This metric considers all declared variables in global scope as well as within the scope of functions, however it excludes function parameters. Declared variables roughly captures the state space of the program, representing another dimension of program complexity.

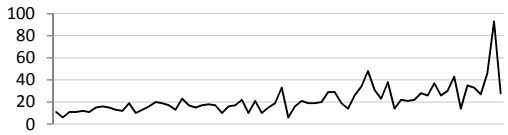

Figure 1. SLOC over semester

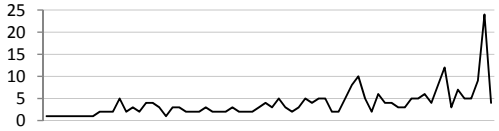

Figure 2. McCabe complexity over semester

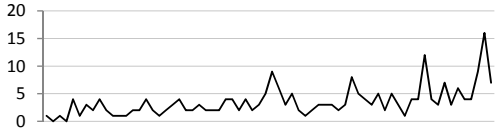

Figure 3. Declared variables over semester

Figures 1, 2, and 3 depict (respectively) the expected SLOC, McCabe complexity measurement, and declared variables for each solution throughout the course of a semester. Each measurement reflects a gradual increase in difficulty over time. There are small peaks later in the semester, indicating more difficult assignments that often combine several concepts, but generally the difficulty level rises steadily as new concepts are introduced and students become more proficient at programming.

Although a number of different metrics could be applied to source code, we believe that these metrics provide a reliable measurement of the effort required to create CS1-level programs.

Figure 4 shows one possible solution to a typical assignment from the curriculum (approximately halfway through a semester): displaying Pascal's Triangle. Students are encouraged to use a previously-created implementation of the `Factorial` function and create the `Combinations` function to calculate binomial coefficients. In this implementation, SLOC is 28, McCabe complexity is seven (three functions plus four loops), and declared variables is six.

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

int Factorial(int n)
{
    int f = 1;
    for (int i=2; i<=n; i++)
        f *= i;
    return f;
}

int Combinations(int n,int k)
{
    return Factorial(n) /
          (Factorial(k) * Factorial(n-k));
}

int main()
{
    int n;
    cout << "How many rows? ";
    cin >> n;

    for (int i=0; i<=n; i++) {
        for (int j = 0; j < n-i; j++)
            cout << "   ";
        for (int j = 0; j <= i; j++)
            cout << setw(6) << Combinations(i,j);
        cout << endl;
    }
}
```

Figure 4. Pascal's Triangle

4 Results

In this section, we present the results of collecting the information described in Section 3. In the next section we provide our interpretation of these results.

Data was collected from programming assignments in six semesters of a CS1 course, Spring 2009 through Fall 2011, in which 192 students were each expected to complete 75-80 problems. Students were allowed (and expected) to submit multiple attempts while working toward a solution; nearly all students who attempted a given problem eventually completed it. Table 1 shows the distribution of final results, after iteration, for all student-created solutions where the student submitted at least one attempt.

Best result reached	Count	Percent
Submitted, but did not compile	284	2.8%
Compiled, but not fully complete	322	3.2%
Successfully completed	9532	94.0%

Table 1. Final Results for Student Solutions

4.1 Process

The 9,532 correct student solutions required 44,548 submissions, or an average of approximately 4.7 submissions and 1.3 sessions per problem to reach a solution. If only problems with "correct" McCabe complexity of seven or higher are considered the average number of sessions grows to 1.5.

Table 2 displays student submission counts for successfully completed problems, grouped by iteration interval. The rightmost column of Table 2 shows how many (and overall percentages) of the submissions from each time "bucket" resulted in an increase over the previous highest grade in that assignment for the student. The average interval between attempts within the same session is 2.6 minutes.

Since Last Submission	Count	New Highest Score
First submission	9532	
< 1 min	14628	5332 (36%)
1-2 min	7652	2429 (32%)
2-5 min	5729	1753 (31%)
5-15 min	3079	936 (30%)
15-30 min	946	279 (29%)
New session	2982	919 (31%)
Total	**44548**	**11648 (26%)**

Table 2. Submissions by Iteration Interval

4.2 Survey

We sent 384 surveys to current (57) and recent (327) students and received 92 apparently valid responses, a 24% response rate. Invalid responses (*e.g.*, answering 240 to every question) have been removed. Still, a wide variation in responses was seen, as indicated by the standard deviations for each question. Student responses are characterized in Table 3.

	Minutes before first submission	Total minutes in typical session	Minutes on a new session before submission	Minutes spent on typical problem	Avg submissions for a problem
Average	34	52	18	67	6
Std. Deviation	40	35	15	64	6

Table 3. Survey of Time Required

4.3 Source Metrics

For our problem set, the average SLOC in the "correct" solutions is 23.8; over all student solutions, the average SLOC is 28.

Table 4 displays the McCabe complexity of student programs in comparison to the "correct" (or expected) solutions.

Table 5 displays the number of declared variables in student programs in comparison to the "correct" solutions.

5 Interpretation

From the data shown here, it is clear that most student submissions fail in both the overall goal of solving the stated problem and the more immediate goal of simply improving their score. Students submissions tend to occur in spurts of several rapid attempts interspersed with longer intervals. Half of all student submissions (50.0%) occurred within two minutes of the previous submission; approximately one-third (32.8%) of all submissions occurred within one minute of a previous submission.

An examination of a random sampling of these quick-turn-around submissions reveals several different types of changes being made. As might be guessed, many include trivial corrections for typographical errors such as simple compile errors (*e.g.,* missing semicolon) or simple output mistakes (*e.g.,* printing the

McCabe Complexity of "Correct" Solution	5 or more below	2 - 4 below	correct ± 1	2 - 4 above	5 or more above	Average McCabe Complexity	Number of Student Solutions	Problems Given
1			.986	.013	.001	1.07	1177	77
2			.912	.077	.011	2.41	2167	163
3		.001	.896	.078	.025	3.66	1984	151
4		.020	.871	.083	.026	4.55	870	63
5	.002	.061	.689	.140	.109	6.79	1244	103
6		.023	.713	.179	.085	7.41	613	64
7	.051	.145	.589	.145	.069	7.22	331	29
8	.053	.220	.542	.163	.022	7.82	227	23
9		.279	.311	.200	.211	10.95	190	19
10		.014	.771	.129	.086	11.11	70	9
11	.101	.242	.450	.087	.121	11.67	149	15
12	.050		.660	.160	.130	13.15	100	10
13	.019	.028	.556	.194	.204	14.94	108	12
14	.273	.045	.205	.023	.455	18.39	44	5
15			.154	.231	.615	20.62	13	3
All	**.022**	**.032**	**.804**	**.093**	**.048**	4.95	9532	770

Table 4. Flow Complexity of Student Solution

Number of Variables in "Correct" Solution	5 or more below	2 - 4 below	correct ± 1	2 - 4 above	5 or more above	Average Number of Variables	Number of Student Solutions	Problems Given
0			.285	.649	.066	2.24	847	50
1			.508	.448	.044	2.48	1527	107
2		.033	.850	.105	.012	2.42	1938	152
3		.064	.790	.114	.032	3.44	1934	152
4		.090	.705	.157	.049	4.65	1518	124
5	.005	.154	.636	.143	.062	5.30	629	59
6	.022	.255	.484	.122	.117	6.37	368	41
7	.015	.145	.512	.275	.052	7.66	324	33
8	.007	.132	.523	.245	.093	8.92	151	17
9	.163	.297	.302	.151	.087	7.83	172	16
10	.050	.250	.550	.100	.050	9.45	40	8
11	.500	.063	.063	.313	.063	10.06	16	2
12	.364		.636			9.73	11	3
14	.714	.036	.143	.107		6.25	28	2
All	**.008**	**.068**	**.648**	**.231**	**.045**	3.82	9532	770

Table 5. State Space in Student Solutions

wrong variable or a misspelled input prompt). However, another category of submissions exist which contain minor changes made hurriedly and seemingly somewhat at random. In many cases the change is unrelated to the feedback received on the previous attempt. It may be that the policy allowing unlimited attempts is being abused in these cases. However, the number of successful completions of student problems, given that any attempt was made, leads us to believe that allowing students to iteratively submit, test, and alter their programs until a solution is reached encourages learning and problem-solving.

We see that students, on average, write their programs with a couple more lines than are really needed to solve the problem. Such a small difference in length, however, may indicate nothing more than small variances in stylistic preferences, rather than significant differences in logic and implementation.

5.1 Survey

The total amount of time T that students work on a typical problem is clearly the sum of the time spent in all sessions working on that problem, or the sum of the lead-in time in the first session plus the lead-in time in later sessions plus the average working time after the first submission of each session. So, clearly we have:

$$T = [F] + [B * (S-1)] + [(M-S) * N], \text{where}$$

F is the average lead-in time in the first session
B is the average lead-in time in any later session
M is the average number of submissions per session
N is the average iteration interval (between submits)
S is the average number of sessions per problem.

The typical iteration interval, number of submissions per session, and number of sessions per problem can be calculated from student submission data; the values S, M, N used below are calculated for just the survey respondents and so may vary slightly from the overall averages.

The values F, B are not measurable by automatically gathered data and so rely on student-reported data. By survey results, students estimated an average $F = 34$ minutes spent on the problem before their initial submission (question #1) and $B = 18$ minutes at the beginning of each later session (question #3).

$$\begin{aligned}\text{Then, } T &= F + B * (S-1) + (M-S) * N \\ &= 34 + 18 * (1.3-1) + (4.6-1.3) * 2.6 \\ &= 48\end{aligned}$$

This conclusion is lower than the student-reported average of 67 minutes per problem. We believe that the difference is attributable to the imprecision of student estimates. Students also tended to overestimate the number of submissions for problems, as compared with known data. Despite differences, the rough similarity of known and student-reported data supports an assumption that, on average, students do spend about an hour per problem. Given approximately 75 problems in the course, this implies that students spend about 75 hours working on programming assignments or approximately 5 hours per week (in a 15-week semester).

5.2 Source Metrics

The average McCabe complexity for the expected solutions is is 3.40, as compared to an average McCabe complexity value of 4.95 for student solutions. It may be possible to craft a solution with a lower than expected McCabe complexity measurement, by employing optimizations not yet introduced in class or by discovering and exploiting gaps in the automated scoring of the problem. However, despite the fact that expected solutions place more emphasis on clarity than optimal McCabe complexity, 14.1% of all student submissions were two or more above the McCabe complexity for the solutions, as opposed to only 5.4% of submissions that were two or more below the McCabe complexity for the expected solution. While most students were able to complete assignments within one McCabe complexity of the "correct" solution, the average difference of one McCabe complexity reflects students' overall tendency to solve problems with higher complexity than required.

The larger average complexity is likely caused in part by an incomplete grasp of the assignment, and thus a tendency to introduce additional, unnecessary logic to handle cases not required by the problem statement. Students may also occasionally use an unnecessarily complicated and circuitous algorithm to solve an assignment, but even for those submissions which use the same algorithmic strategy as the expected solutions, additional McCabe complexity may arise from an imperfect understanding of Boolean algebra used to reduce logical expressions and control flow to a more minimal level.

Again, it may easily be possible to use fewer variables; the "correct" solutions were not intended to be fully optimized by any standard, but simply to reflect expected student behavior at this point in the course. The average number of variables per "correct" solution is 2.91, as compared to an average variable count of 3.82 for student solutions. Student submissions once again exhibit higher complexity than the expected solutions, in the form of increased state space, with 27.6% of student submissions using two or more variables beyond the number appearing in the "correct" solutions. This is to be expected, as students are still learning the basic concepts of programming and may not fully grasp when it is necessary or even helpful to add an additional variable to the program.

Having examined metrics for the complexity, state space, and length of student solutions, we see that students tend to craft solutions that are slightly larger (in every dimension) than is required. This excess likely stems from their iterative approach to problem-solving, identifying and correcting one problem at a time and persisting until no problems are apparent. It may also be indicative of the average student's incomplete understanding of concepts like state space and Boolean algebra.

6 Future Work and Conclusion

In this paper, we have presented data gathered from CS1 sections over the past few years and reported on measurements of both the process and results of student programming when using an iterative assignment framework. This data suggests that students that the advantage of very high completion rates may be offset by more complicated solutions when given instant feedback and unlimited retries. Probably because the framework lessens the need for (and perceived importance of) full understanding. When encountering issues, students "fix" the problem by introducing additional control flow structures, thus increasing the complexity of the solution.

In future work, we will observe the effect induced by making some part of the student's score dependent on the McCabe (or similar) complexity measurement of their submission, relative to the expected value. Similarly, we will investigate the effects of "throttling" submissions by introducing either a maximum number of attempts, or minimum submission interval. We also plan to introduce measurements of student confidence in the correctness of their program. Given the high failure rate of student submissions before reaching a solution, we think it would be valuable to assess the confidence students feel in the correctness of their programs prior to submitting them for scoring.

References

[1] S. Edwards. Improving student performance by evaluating how well students test their own pro-

grams. *Journal on Educational Resources in Computing*, 3(3):1–24, Sept. 2003.

[2] S. Edwards. Experiences using test-driven development with an automated grader. *Journal of Computing Sciences in Colleges*, 22(3):44–50, Jan. 2007.

[3] S. Edwards and M. Perez-Quinones. Web-cat: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITICSE'08)*, pages 328–328. ACM, 2008.

[4] S. Edwards, J. Snyder, M. Perez-Quinones, A. Allevato, D. Kim, and B. Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the fifth international workshop on computing education research workshop (ICER'09)*, pages 3–14. ACM, 2009.

[5] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, March 2007.

[6] M. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, March 2005.

[7] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE 2(4):308–320, December 1976.

[8] C. Murphy, G. Kaiser, K. Loveland, and S. Hasan. Retina: helping students and instructors based on observed programming activities. In *Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE'09)*, pages 178–182. ACM, 2009.

[9] P. Nordquist. Providing accurate and timely feedback by automatically grading student programming labs. *Journal of Computing Sciences in Colleges*, 23(2):16–23, Dec. 2007.

[10] C. Norris, F. Barry, J. F. Jr., K. Reid, and J. Rountree. Clockit: collecting quantitative data on how beginning software developers really work. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITICSE'08)*, pages 37–41. ACM, 2008.

[11] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th annual conference on Innovation and technology in computer science education (ITICSE'06)*, pages 13–17. ACM, 2006.

[12] T. VanDeGrift, T. Caruso, N. Hill, and B. Simon. Experience report: getting novice programmers to think about improving their software development process. In *Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE'11)*, pages 493–498. ACM, 2011.

[13] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on computing education research workshop (ICER'09)*, pages 117–128. ACM, 2009.