Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 06, No. 03, September 2020.

USING STATIC ANALYSIS TOOLS FOR ANALYZING STUDENT BEHAVIOR IN AN INTRODUCTORY PROGRAMMING COURSE

Ibrahim Albluwi[1] and Joseph Salter[2]

(Received: 15-Mar-2020, Revised: 2-May-2020, Accepted: 18-May-2020)

ABSTRACT

Analyzing student coding data can help researchers understand how novice programmers learn and inform practitioners on how to best teach them. This work explores how using static analysis tools in programming assignments can provide insight into student behavior and performance. The use of three static analysis tools in the assignments of an introductory programming course has been analyzed. Our findings confirm previous work regarding that formatting and documentation issues are the most common issues found in student code, that this is constant regardless of major and performance in the course and that there are certain error types which are more correlated with performance. We also found that total error frequency in the course correlates with final course grade and that the presence of any kind of error in final submissions correlates with low performance on exams. Furthermore, we found females to produce less documentation and style errors than males and students who partner to produce less errors in general than students working alone. Our results also raise concerns on the use of certain metrics for assessing the difficulty of fixing errors by students.

KEYWORDS

Introductory programming, CS1, Static analysis, Automated feedback, Coding style, Gender differences.

1. INTRODUCTION

Students generate a great amount of data as they learn how to program. This data is a precious mine for understanding how they learn, what challenges they face and how they interact with tools. This work complements ongoing efforts to analyze student coding behavior by tackling a type of data that has received limited attention so far. Considerable work has been done on analyzing submission and compilation behavior, compile time errors and other issues related to the correctness of student code [1]. This work analyzes issues that do not necessarily affect correctness and that can be detected using *static analysis*; i.e., without having to execute the code, like documentation, testing and style issues.

The importance of this work derives directly from the importance of analyzing student coding behavior in general. The better we understand the types of issues students face, how they address them and how they interact with the tools that report them, the better we are able to improve these tools, adapt teaching to directly address the challenges they face and intervene early to help struggling students. More particularly, studying static analysis errors provides the following two advantages:

- It is *another source of information* beside information available from compiling and executing the code. Having an extra source of information is especially important when data is scarce.
- It provides *another dimension* for looking at novice programmers. Most issues detected by static analysis tools do not prevent the code from running correctly. Such issues are orthogonal to issues that are detected by the compiler or that cause programs to crash or produce incorrect results.

In this work, we analyze a large dataset of student submissions to programming assignments in an introductory programming course. The main goal is to answer questions about static analysis errors along the following two dimensions:

The errors:

1. Which static analysis errors are most frequent in students' code?
2. Which errors appear most in initial submissions? Which errors persist in final submissions?

1. Ibrahim Albluwi is with Princeton University, Princeton, New Jersey, USA. Email: isma@cs.princeton.edu
2. Joseph Salter is with Quartet Health, Caldwell, New Jersey, USA. Email: joesalter17@gmail.com

3. Which errors take longer to fix, and which errors take more submissions to fix?
4. Is there any relationship between the number of errors in student code on assignments and their performance on exams?

The students:

1. Are there any differences in the number and type of errors between students based on their major, prior-programming experience, gender or performance in the class?
2. Are there any differences in the number and type of errors between students who partnered on their assignments and students who worked alone?

By answering such questions, this work implicitly attempts to answer also broader questions on the suitability of using static analysis to understand the behavior of students in introductory programming courses and to predict their performance.

Note that we refer to issues flagged by the static analysis tools as "errors", although these issues do not necessarily represent "errors" in the code. They represent violations to rules that are checked by the tools and that span a wide range of issues related to formatting, documentation, testing, style and possible programming bugs.

This article is organized as follows. Section 2 discusses previous work related to this study and Section 3 describes the data, context and methodology of the conducted analysis. Section 4 reports results at the aggregate level and Section 5 reports results specific to the different student subgroups analyzed in the study. A discussion of the main results and their implications is provided in Section 6 and the limitations of the study are discussed in Section 7. Future work horizons are finally indicated in Section 8.

2. PREVIOUS WORK

2.1 Learning Analytics in Introductory CS Courses

Interest in analyzing student data in introductory programming courses is part of a more general trend of interest in learning analytics. This trend gained momentum with the increased availability of resources for storing and processing data. Generally speaking, learning analytics research aims at: (1) identifying student behavioral patterns and (2) deriving interventions that improve learning [2]. Hui and Farvolden [3] proposed a framework for how and when learning analytics can be used in the classroom and demonstrated the utility of this framework with a case study in a CS1 course. Hundhausen *et al.* [2] also proposed a process model for improving CS courses using tools integrated in the IDE that collect and analyze data and apply interventions. Ihantola *et al.* [1] performed an extensive review on learning analytics in programming courses and provided a thorough discussion of issues and challenges facing the field. One of these issues is that data is mostly private, limited in size and unique to a particular institution, which makes research reproduction and replication difficult. The problem is also amplified with the bias many researchers in the CS education community have against replicating, reproducing and repeating previously published studies, as reported by Ahadi *et al.* [4].

2.2 Analysis of Coding Behavior

There is a wide range of student activity data that could be analyzed. The most relevant data to this work is coding behavior data. The most known tool for automatically recording such data is BlackBox [5], which records program line-edits in the BlueJ IDE. Other examples of tools include online coding environments like PCRS [6] and CloudCoder [7] and IDE plugins like TestMyCode [8].

Several works have analyzed the process through which students work on their assignments. For example, Piech *et al.* [9] tracked the evolution of student code over compilation events and developed a model for predicting struggling students. Karavirta *et al.* [10] classified students into categories based on their pattern of resubmission to an automatic grader. They found that some students used the ability to resubmit in an inefficient way and that limiting the number of submissions for such students at the beginning of the course could improve their performance later on. Blikstein [11] found that novice students tended to copy and adapt large batches of sample code, whereas experienced students tended to code more incrementally. Blikstein *et al.* [12] also found that changes in code update strategies over the semester were correlated with performance on exams. On the other hand, Allevato

and Edwards [13] found that students who had more increasing changes in code between submissions to an automatic grader generally performed better in the course.

Several works have also analyzed errors that novice programmers make. For example, Altadmri and Brown [14] studied 37 million compilation events to analyze the frequencies of different errors and the time taken to fix them. Brown and Altadmri [15] also found that instructor beliefs about error frequencies do not agree with the data. Tabanao *et al.* [16] used compile-time errors and Jadud's Error Quotient [17] to detect at-risk students and predict midterm exam scores, which is an algorithm that has been shown to find correlation between errors and student performance. It is difficult though to tell whether or not it can be used as a reliable predictor [18]-[19].

2.3 Static Analysis Tools

Static analysis tools are useful for detecting code that does not follow programming standards and pointing out potential bugs, performance issues and flaws in logic. The tools used in this study are Checkstyle[1], PMD[2] and FindBugs [20], which are among the most commonly used open source static analysis tools. Checkstyle has a focus on Java style. PMD focuses more on programming flaws and unreadable or non-simplified code than on style. Common PMD checks include finding unused variables, non-simplified expressions and empty statements. FindBugs is run on byte code rather than on source code. Its purpose is to find byte patterns that are often bugs, like assignments to variables that are never used and values that will always be null. Together, Checkstyle, PMD and FindBugs search for a wide range of flaws in programs, both cosmetic and non-cosmetic.

Although these tools can be used for educational purposes, they were originally designed for software professionals. Therefore, much of the research on these tools targets the non-educational community (e.g. [21]-[24]). In an early study, Mengel and Yerramilli [25] argued for the value of using static analysis tools intended for professionals in grading student programs. Nutbrown and Higgins [26] warned against applying direct mark deductions based on errors reported by these tools. They found this to produce results that are very different from how instructors evaluate student code. Truong *et al.* [27] developed a static analysis framework specifically for checking the quality of student programs and their structural similarity to model solutions. Their goal was to provide better-than-standard feedback to students and tips for improvement. Other similar tools were also described in the literature, like PyTA [28], Gauntlet [29] and Expresso [30]. For reviews on how static analysis tools can be used for educational purposes, see Striewe and Goedicke [31] and Rahman and Nordin [32].

2.4 Static Analysis of Student Code

Edwards *et al.* [33] conducted a thorough qualitative study of issues reported by CheckStyle and PMD for around half a million student Java submissions. They found that documentation and formatting issues were the most commonly reported issues by the tools and that issues that are potentially coding flaws were most indicative of performance even when students fixed these issues. They also found that the most common issues were consistently the same among majors, non-majors and students at different experience levels. In our work, we conduct the analysis along the same lines in Edwards *et al.* [33]. Our results confirm the above-mentioned results, with a few differences as will be discussed in Section 6. Our work also expands the analysis, reports new results and provides further insights.

Keuning *et al.* [34] conducted an analysis of errors reported by PMD on student code. However, their analysis was restricted to issues related to code quality, like modularization, decomposition and the use of idioms, which is a subset of what static analysis tools report. Their analysis included issue frequencies and time needed to fix them. Their work also compared students who used static analysis tools to students who did not and concluded that quality issues in student code are rarely fixed and that students typically ignore issues reported by the tools. These results seem to be course-specific, as they are not consistent with what has been observed in this work as will be discussed in Section 6. Liu and Peterson [28] also reported positive results when using PyTA in an introductory course, where they observed (compared to a previous year) a significant reduction in the number of repeated errors per submission, submissions to pass a programming exercise and submissions required to solve the most

1 https://checkstyle.sourceforge.io/
2 https://pmd.github.io/

common errors. Other studies confirming the utility of using static analysis tools in introductory programming classes include Delev and Gjorgjevikj [35] (in C) and Schorsch [36] (in Pascal).

3. METHODOLOGY

3.1 Context

COS 126 is the introductory programming course at Princeton University. The course is required for CS majors and engineers and open for non-majors. The course has 9 Java programming assignments (46% of the course grade) and a machine code assignment (4% of the course grade). Students are allowed to work with a partner on the last 4 of the Java assignments.

On assignments where partnering is allowed, students are required to follow the pair programming protocol [37]. The collaboration policy on the course website provides the details of this protocol and instructors emphasize verbally in their sections that students who partner must follow it. Students who partner must also state in a readme file submitted with each assignment that they have followed the pair programming protocol. However, there is no way to tell whether or not students actually followed the pair programming protocol beyond taking their word for it.

Students submit their work to an online system that runs automated tests for checking correctness, performance and API adherence. The system also runs CheckStyle, PMD and FindBugs and includes issues reported by these tools in the automated feedback. Students are allowed to receive feedback from the system as many times as they wish before marking their submission as ready for grading. Human graders then use a rubric to grade the final submissions and provide personalized feedback on correctness, performance and style. The feedback in the first assignment includes a warning for students who do not address all the issues flagged by the tools and indicates that a deduction will be applied in later assignments. These deductions are minor (typically ≤ 3 points out of 20) and depend on how many issues flagged by the static analysis tools are left unaddressed. The rubrics are not provided to the students beforehand, so it is not directly clear to them what the deductions are before their work is graded. Note that students are not directly "taught" how to format or comment their code in the classroom but are provided with a link to a page that lists the style guidelines for the course.

The IDE used in the course was Dr. Java, which does not automatically format the code, but provides a menu option for achieving that. While the course did not require using Dr. Java, it did not provide support for other IDEs and used a custom version of the IDE pre-packaged with the libraries required in the course. This made almost all students use it instead of other IDEs. Anecdotally, the number of students who used IDEs other than Dr. Java was very small and insignificant.

The course has two programming exams (7.5% each) and two written exams (17.5% each). Programming exams are conducted on-campus under exam conditions, where students are expected to complete within 80 minutes a few programming exercises that are much smaller in size than the programming assignments. Grading is done based on code correctness only, without any consideration to style, commenting or testing issues. Written exams are also timed (80 minutes each) and conducted on-campus under exam conditions. However, they test programming knowledge as well as other computer science concepts, like algorithm analysis, digital logic and theory of computation (state machines, computability and intractability). Programming knowledge in these written exams is tested with questions that do not involve code writing (e.g. multiple-choice, true/false, …etc.)

The course has an average score of 87.2% and a standard deviation of 7.2%, where around 0.6% receive an F grade at the end of the semester and around 5.3% drop out after the first written exam. Students who drop out in the first few weeks of the semester (before the first exam) are typically students who "shop" for courses rather than students who struggle with the material. Moreover, not all students who drop out after the first exam are struggling students, as it is not uncommon for some students to drop the course if they feel that they won't achieve a grade of an A or an A-.

3.2 Dataset

Data has been collected in COS 126 in three semesters; Fall 2016, Spring 2017 and Fall 2017. The data includes every Java file in every submission for every student who attended the final exam of the course in these three semesters. This includes intermediate submissions that were not considered for

grading. The data also includes the text of every piece of feedback students received from the automated feedback system, as well as timestamps for when the files were submitted. Overall, the dataset has a total of 1,051,105 occurrences of 304 distinct issues flagged by the static analysis tools covering a total of 968 students who completed the course. This is around 87.8% of the students who were enrolled in the course on the first day of classes in each of the three semesters. Around 13.2% of the students dropped out at different points in the semester and were excluded from the analysis, as it is impossible to tell for each of them until which point in the semester they were taking the course seriously. As discussed before, a large proportion of these students did not sign up for the course with the intention of completing it.

3.3 Metrics

The analysis in this work broadly follows the analysis performed by Edwards *et al.* [33]. The first two of the metrics used in the analysis are commonly used in error analysis studies and were implemented in the same way as reported by Edwards *et al.* [33]. The third metric was not used by Edwards *et al.*

- ***Error Frequency***. Error counts for each source file were normalized by the number of lines of code and considered as occurrences per thousand lines of code (KLOC).
- ***Time-to-Fix***. This is an estimate for how long an error stays unfixed in the student's code after it has been reported by the tool. An increase in the frequency count of an error from a submission to another is considered as an introduction of new errors, whereas a decrease in the count is considered as a resolution of errors. The *time-to-fix* is the difference between timestamps of introduction and resolution events for a certain error.
- ***Submissions-to-Fix***. This is an estimate for the number of submissions taken by the student to fix the error. The same protocol used for computing the time-to-fix is used, but the submission number is recorded instead of the timestamp.

Note that the time-to-fix is not necessarily the same as the actual time-on-task by the student trying to fix the error. The student might take breaks between submissions or put off fixing the error if they feel that it is not important to fix the error immediately.

3.4 Error Categories

Static analysis tools report many errors on different types of issues. Edwards *et al.* [33] grouped these errors into categories that we adopt in this work:

- ***Coding Flaws***: "Constructs that are almost certainly bugs (such as checking for null after a pointer is used ...)".
- ***Excessive Coding***: "Size issues, such as methods, classes or parameter lists that exceed the expected limits and may indicate readability problems".
- ***Formatting***: "Incorrect indentation or missing whitespace".
- ***Naming***: "Names that violate capitalization conventions, are too short or are not meaningful".
- ***Readability***: "Issues other than formatting that reduce the readability of the code".
- ***Style***: "Code that can be simplified or that does not follow common idioms".

The following categories were adopted with modification:

- ***Documentation***: Unlike [33], Checkstyle Javadoc comment checkers are ignored, since Javadoc comments are not taught in the course. The issues considered instead are: not preceding a field or a method by a comment, leaving an empty method body without a comment, not adding a header comment for a file and not formatting a header comment according to the course guidelines.
- ***Testing***: Formal unit testing was not required in the course. Therefore, this category includes only one custom CheckStyle error that checks whether or not the student calls all public methods in the main method. Understandably, this is a naive check. A student might call a public method in main, but not genuinely test it (by printing out the result, for example). The check does not catch such instances, but submissions flagged by the check certainly lack sufficient testing.

Table 1. Examples for each of the 10 most frequently occurring errors.

Error	Category	Example
`WhitespaceAround`	Formatting	`x=4` instead of `x = 4`.
`Comment`	Documentation	A method is not preceded by a comment.
`RegexpSingleline`	Documentation	The student name or ID is missing from the header comment.
`IllegalTokenText`	Formatting	`/*comment*/` instead of `/* comment */` or `//comment` instead of `// comment`
`MainCallsAllPublicMethods`	Testing	A public method is not called in main
`WhitespaceAfter`	Formatting	`(int)x` instead of `(int) x`.
`RegexpMultiline`	Style	Not starting every line in a multiline comment block with an asterisk.
`LineLength`	Formatting	A line has more than 85 characters.
`NumericLiteral`	Style	Using a numeric literal instead of defining and using a constant.
`RegexpHeader`	Documentation	Header comment is missing from a file.

Edwards *et al.* [33] also include a Braces category for errors related to missing optional braces. This category was omitted, because such errors were not encountered in our analysis. It must also be noted that Edwards *et al.*'s analysis includes 112 distinct errors across Checkstyle and PMD, whereas ours included 304 distinct errors across Checkstyle, PMD and FindBugs. Errors that were not seen by Edwards *et al.* were categorized into the above groupings[3].

4. Results: Aggregate Student Analysis

We first performed an analysis on average error statistics across all students to describe the behavior of a typical student in the course.

4.1 Error Category Frequency

After grouping errors into the categories described in Section 3.4, the resulting frequencies are displayed in Figure 1. The figure shows the average error frequency across all submissions compared to error frequencies in the initial and final submissions. To put the frequency rates per KLOC into perspective, student assignments in this study contained an average of 194 lines of code, which means that an average submission has around 6 errors in the Formatting category, 3.7 in the Documentation category and 2.4 in the Style category.

Considering the average error frequencies, Formatting errors are most prevalent, with an average of 31.24 per KLOC, followed by Documentation errors at 19.16 per KLOC. Together, Formatting and Documentation errors account for 60% of the total errors in the student code. Note that Formatting and Documentation errors encompass only 19 of the 304 unique error types used in the analysis. So, 60% of the error count is covered by just 6.25% of the error types seen. Appending Style to this count adds 121 unique error types and 15% to the total error count. The top three error categories, then, account for 75% occurrences of all errors. The vast majority of errors being made are thus limited to a small number of different error types.

Looking at final submissions, it is clear that *most* of the errors get fixed, with Documentation errors getting fixed at a very high rate. Overall, only 2.3% of the total errors occurred on final submissions. Formatting errors remain by far the most common, which could be the result of formatting error messages being less clear to novices compared to error message on missing comments. It could also be that students are less receptive to changing their style in formatting the code. The small discrepancy between Testing errors on initial *versus* average submissions suggests that students might be adding tests only in their final submissions to silence the error, as opposed to writing tests to actually test their code. More on this issue will be discussed in Section 6.

3 A full listing of the errors along with their groupings and occurrence per KLOC can be found at: https://figshare.com/s/cbe48ead7dcc7b15a5bf

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 06, No. 03, September 2020.

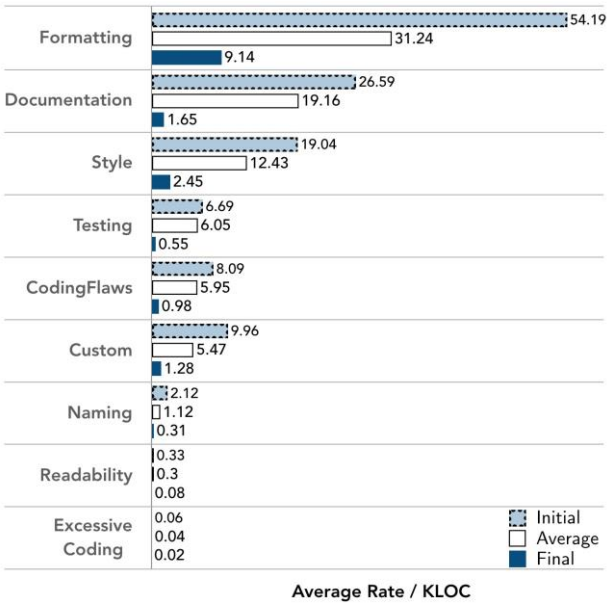

Figure 1. Error rates (in KLOC) on initial, average and then final submissions.

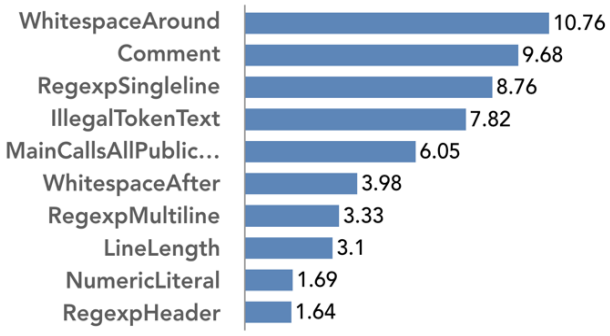

Figure 2. Average rate per KLOC for the 10 most frequently occurring errors.

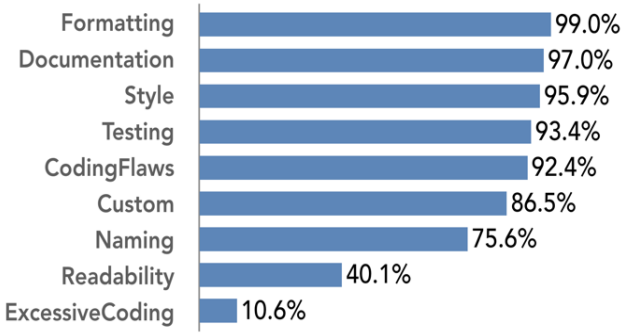

Figure 3. Percentage of students who make errors from each category.

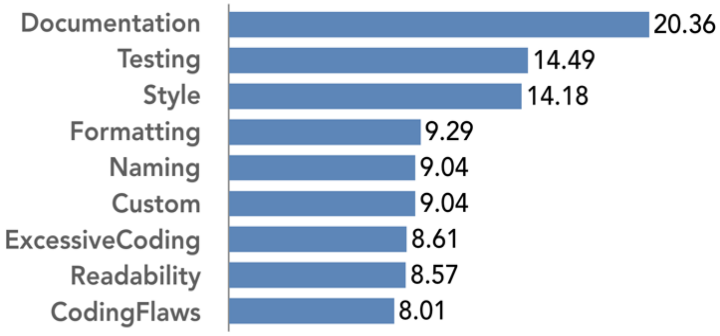

Figure 4. Average number of hours taken to fix errors from each category.

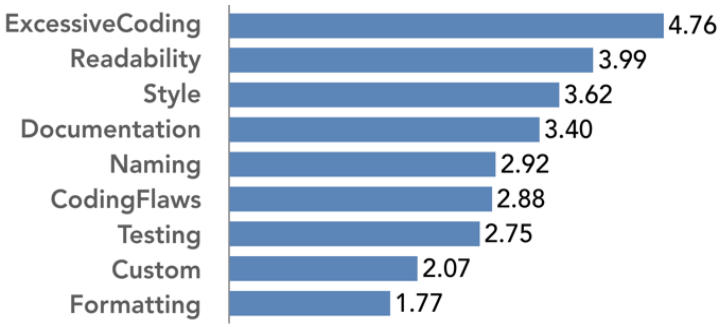

Figure 5. Average number of submissions taken to fix errors from each category.

4.2 Error Frequency and Popularity

Considering the errors individually, Figure 2 shows the 10 most frequently occurring errors, where Formatting and Documentation errors account for seven of the top 10. Table 1 shows examples for each of these errors to explain what they mean. Figure 3 shows the percentage of students who make errors from each category at some point during the semester. Almost all students make Formatting, Style, Documentation and Coding Flaw errors, while slightly less make Testing and Naming errors. Readability and Excessive Coding errors are much less common. This order closely resembles the order of frequency presented in Figure 1. We will describe later in more detail which types of students produce more errors from each category.

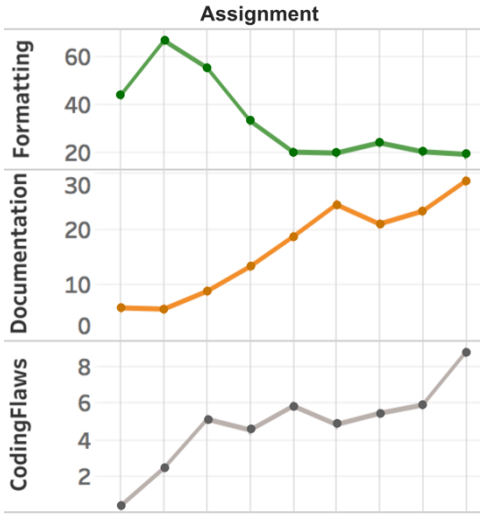

Figure 6. Error rates per KLOC over time for assignments 1 to 9.

Table 2. Average exam scores for students who leave at least one error in a final submission for any assignment *versus* those who fix all errors.

	Written Exam		Programming Exam	
	Errors	No Errors	Errors	No Errors
CodingFlaws	69.4%	76.8%	79.3%	87.4%
Testing	67.0%	75.8%	78.0%	86.0%
Style	72.2%	77.0%	83.0%	86.7%
Documentation	68.8%	76.3%	79.5%	86.5%
Formatting	72.4%	76.7%	83.1%	86.5%

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 06, No. 03, September 2020.

4.3 Error Progression over Time

Figure 6 displays the average rates of the error categories that show an interesting trend line. Error categories not shown have arbitrarily fluctuating trend lines or error rates that are too small for the trend line to be meaningful. The clear decline in Formatting errors suggests that students eventually gain a grasp on what their code should *look* like. This is interesting, as the only instruction students receive in the course on formatting is through the error messages produced by the static analysis tools. Another possible explanation for this decrease in Formatting errors is that more students into the semester might start using the code formatting option in Dr. Java, which is hidden in one of the menus. However, this is difficult to tell from the data.

On the other hand, Figure 6 shows an increase in Documentation and Coding Flaw errors over the semester. The increase in Documentation errors could be explained by lack of care; over time, students may grow tired of Documentation errors and choose to ignore fixing them until the final submissions. The increase in both the Documentation and Coding Flaw errors could also be related to the growing complexity of assignments over the semester.

4.4 Correlation with Exam Performance

We examined the total error counts, as well as the error counts from each error category, *versus* grades in programming exams and written exams. No apparent correlation was revealed in any case. However, we found a correlation between the *presence* of error categories in final assignment submissions on both written and programming exam performance. Table 2 displays average written exam scores for students who left at least one error from the given category in a final submission for any assignment *versus* those who did not. Only error categories with statistically significant differences are shown (T-test p-val. < 0.0001). For reference, the average score on written exams was 74.2% with a standard deviation of 12.8% and on programming exams was 84.7% with a standard deviation of 12.9%.

5. RESULTS: STUDENT SUBGROUP ANALYSIS

5.1 Low-Performing *versus* High-Performing Students

We have already examined the relationship between error frequencies and exam performance. We provide here a more detailed analysis to compare low- and high-performing students in the course in general. To perform this analysis, we break students into groups. We approached this in the following three different ways:

1. Break students into three evenly sized tertiles based on their final course average.
2. Create a C+ and-below subgroup, a B-range subgroup and an A-range subgroup based on final course average, using the standard cutoff points (80% for B; 90% for A).
3. Group students by taking course averages of at least 1 standard deviation below the mean, within one standard deviation of the mean and at least one standard deviation above the mean.

It turned out that the three options listed above produced very similar results in terms of significant differences in error behavior between subgroups. Consequently, we choose to present Option 1 and we define the three subgroups as low-performers, medium-performers and high-performers. For reference, the mean course grade was 87.2% with a standard deviation of 7.2%.

Table 3 shows statistics for each subgroup individually compared to all the students together. The “Avg. Duration" column is the average duration between initial and final submissions. Note that the “Avg. Duration" does not necessarily reflect the actual number of hours that a student spent working on the assignment. It describes the duration between the first and last files submitted.

As seen in the table, low-performers produce more errors in overall than the average student, while high-performers produce significantly less. A one-way ANOVA shows that the error rates for each performance group are statistically significant ($F = 42.8$, $p < 0.0001$) and Tukey’s HSD test[4] shows that the average error rates for low-, medium- and high-performers are statistically different.

4 Tukey’s HSD is a statistical tool often used as a *post hoc* test with ANOVA. ANOVA provides a significance test; Tukey’s HSD compares all pairs of means and determines which pairs are statistically different.

Table 3. Student performance groups.

Performance Level	Grade Range	Number of Students	Errors/ KLOC *	Average Duration *	Average # of Submissions **	Hours to Fix **	Submissions to Fix **
Low	47% - 85.9%	329	88.2	41.2 hrs.	16.7	12.6	5.14
Medium	85.9% - 91.1%	318	75.2	54.0 hrs.	17.4	15.6	5.05
High	91.1% - 99.2%	321	66.6	63.5 hrs.	14.4	15.25	4.3
All Students	47% - 99.2%	968	74.3	52.8 hrs.	16.17	14.5	4.85

*: Significant differences between all groups.
**: Significant differences between high- and low-performers and between high- and medium-performers.
Calculated by ANOVA and Tukey's HSD.

The table also shows that low-performers, interestingly, take less time, but more submissions, to fix errors compared to high-performers and these differences are significant (time-to-fix: $F = 36.7$, $p < 0.0001$; submissions-to-fix: $F = 118.1$, $p < 0.0001$) and different from each other, as suggested by Tukey's HSD. This may suggest that low-performers submit their code more blindly than high-performers and cram their work into shorter periods of time, which requires them to submit more and at a faster rate. On the other hand, high-performers seem to rely less on the automatic grader, spending more time in overall on assignments while using less submissions.

Figure 7 displays the difference in average rate per assignment across error categories for each performance group, omitting the statistically insignificant categories (Naming, Readability and Excessive Coding). It is evident that high-performers produce less errors in all categories than medium- and low-performers and medium-performers produce less errors than low-performers in all categories but Documentation and Testing. An ANOVA suggests that the most significant differences among the three are in Formatting and Coding Flaws and the least significant differences are in Documentation and Testing. The disparity in Coding Flaws suggests a higher level of misunderstanding amongst low performers. Interestingly, all three performance groups seem to be testing their code relatively equally.

Further analysis reveals that differences in error counts on *initial* submissions are statistically insignificant for all categories except for Coding Flaws. Low-performers produce 11.5 of these types of errors per KLOC in initial submissions, compared to 8.92 from medium-performers and just 5.79 for high-performers ($F = 23.5$, $p < 0.0001$). The course average is 8.75. This suggests that low-performers are producing more buggy code on their initial submissions.

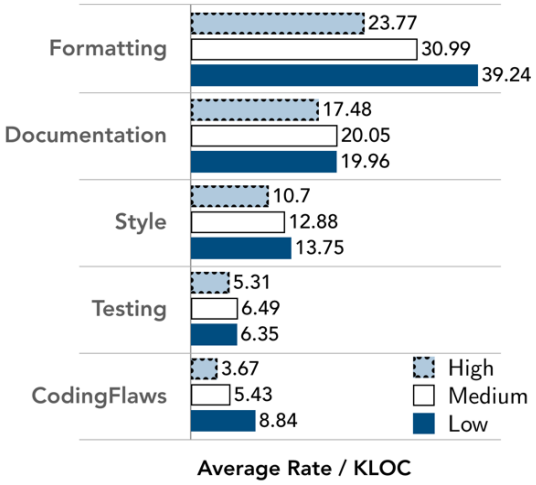

Figure 7. Error category rate per KLOC for different performance groups.

Table 4. Student major groups.

Major Group	Number of Students	*Avg. Course Grade	**Avg. Duration	***Avg. # of Submissions
CS	97	89.7%	55.4 hrs.	14.8
Engineering	230	86.6%	46.3 hrs.	15.9
Other Science/Math	40	87.8%	54.5 hrs.	13.9
Social Science	46	84.2%	47.9 hrs.	15.0
Humanities	555	87.4 %	56.0 hrs.	16.8

* Significant differences between CS and all other majors except Science/Math.
** Significant differences between Engineering and Humanities.
*** Significant differences between (CS and Humanities) and (Humanities and Other Sciences).
Calculated by ANOVA and Tukey's HSD.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 06, No. 03, September 2020.

5.2 Majors *versus* Non-Majors

Students were grouped into five major groups, as shown in Table 4. Analyzing error counts per KLOC and the number of submissions taken to *fix errors* (not shown in the table), we found no significant differences between the major groups. However, we found CS students to take significantly longer to fix errors than each of the other groups (17.6 hours per error). Non-CS Engineering students take just 12.7 hours, which is significantly shorter than humanities and social science students (14.8 hours). This difference in the time-to-fix, but not in the submissions-to-fix, suggests that CS majors might be working longer before submitting or taking longer breaks between submissions. Engineers on the contrary seem to be cramming their work into a shorter amount of time. This contrast between CS majors and engineers seems difficult to explain only using the data.

We also performed three binary comparisons: first grouping CS and engineering students together and comparing them to their counterpart and second comparing students in STEM fields *versus* those not in STEM fields. These yielded no significant results. However, comparing CS majors and non-CS majors reaffirmed our earlier statement that CS majors take more time to fix errors ($F = 30.2$, $p < 0.0001$). It also showed that CS majors produce less Coding Flaw errors on initial submissions, averaging 5.9 occurrences per KLOC *versus* 9.1 from non-majors ($F = 5.8$, $p = 0.003$).

5.3 Students with Different Prior Programming Experience

Students in the course entry survey indicated their level of programming experience by choosing between "None", "Some", "Plenty" and "I've worked on some sizable projects". Omitting the latter two categories due to their small size, Table 5 contains the description of the former two.

Table 5. Comparison between student groups.

	# of Students	Avg. Grade*	Avg. Duration*	Avg. # of Submissions
No Prior Experience	433	85.1%	49.6 hrs.	16.5
Some Prior Experience	458	88.7%	56.6 hrs.	16.0
Male	545	87.8%	53.1 hrs.	16.1
Female	390	86.7%	53.2 hrs.	16.2

* Significant differences between students with some and with no prior programming experience (ANOVA).

Table 6. Male *vs.* female characterization.

	Male	Female
% Majoring in CS	11.7%	7.95%
% Majoring in Engineering	24.8%	19.7%
% Majoring in Humanities	54.1%	64.6%
Avg. Prior Experience	0.69	0.56

Table 7. Differences in male *vs.* female error frequencies on initial, average and final submissions.

Gender	Testing			Style			Documentation		
	Initial	Average	Final	Initial	Average	Final	Initial	Average	Final
Female	6.46	Insignificant	Insignificant	17.5	10.7	1.76	24.0	15.9	0.91
Male	7.73	Insignificant	Insignificant	21.6	13.6	2.77	32.5	21.3	2.09

Table 8. Average partner statistics.

	Course Grade	Duration*	Number of Submissions*
Pair	87.2%	54.4 hrs.	17.7
Solo	87.9%	97.4 hrs.	24.9

* Significantly different (ANOVA).

Table 9. Significant differences in error rates on initial submission.

	Coding Flaws	Testing	Style	Doc	Formatting	Naming
Pair	8.45	9.2	17.3	32.1	22.3	1.3
Solo	12.0	12.5	25.6	43.2	32.6	2.2

Students with no prior programming experience and those with some show no statistically significant differences in their error rate or time-to-fix. An ANOVA on average number of submissions-to-fix

1. Ibrahim Albluwi is with Princeton University, Princeton, New Jersey, USA. Email: isma@cs.princeton.edu
2. Joseph Salter is with Quartet Health, Caldwell, New Jersey, USA. Email: joesalter17@gmail.com

does yield significant results ($F = 12.12$, $p < 0.0001$), but the submission numbers are very close together to have any meaning. Breaking down the errors into categories, we again find one significant result regarding Coding Flaw errors on initial submissions. While the average student produces 8.75 of this error per KLOC in their initial submissions, students with no experience generate an average of 10.6 and students with some experience generate 7.61 ($F = 12.7$, $p < 0.0001$). This shows a repeated trend with Coding Flaw errors on initial submissions. Aside from this finding, though, prior programming experience does not seem to be clearly related to the static analysis error behavior.

5.4 Gender Differences

The gender breakdown of the students is displayed in Table 5 and Table 6 attempts to characterize the typical male *versus* the typical female in our course. There are more females in the humanities than males and more males in CS and Engineering than females. Male students also typically enter the course with more prior programming experience than female students. The averages in Table 6 (0.69 and 0.56) were computed by considering "no prior experience" a 0 and "some prior experience" a 1.

An analysis of the aggregate error counts showed that female students produce an average of 67.8 static analysis errors per KLOC on assignments, while males produce 78.3 ($F = 13.0$, $p < 0.0001$). Females also take slightly less time to fix errors – 13.5 hours *versus* 15.2 for males – and also slightly less submissions – 4.7 *versus* 5 (time-to-fix: $F = 21.8$, $p < 0.0001$; submissions-to-fix: $F = 27.5$, $p < 0.0001$). The differences in these figures are not enough to justify a deep analysis.

Breaking down error frequencies based on error categories, Table 7 displays the significant differences found in initial, average and final submissions between male and female students. As the table shows, female students produce less *Style* and *Documentation* issues on all submissions. Female students also test their code more on *initial* submissions, but the differences eventually even out in later submissions. These differences were not seen when comparing students by major or by prior experience, which suggests that they are gender related.

5.5 Pair *versus* Solo Programmers

To test whether students who work with a partner produce less errors than students who work individually, we limited our analysis to only assignments where students were given the option to partner (the final four assignments in the course). This ensures that, when comparing partnering *versus* non-partnering students, all students in the analysis were working on assignments of equal difficulty. Note that on average, around 43% of the students worked individually in the assignments we analyzed.

Table 8 shows statistics on partnering *versus* non-partnering students in assignments where partnering was allowed. Immediately clear is the significantly fewer hours and submissions spent on assignments by students who partner. On an aggregate level, students who partner produce 61.4 errors per KLOC per assignment on average, compared to 77.3 for those working alone ($F = 19.2$, $p < 0.0001$). Partnering students also take 15.9 hours to fix errors on average compared to 19.3 for non-partnering students ($F = 29.6$, $p < 0.0001$). In terms of the number of submissions taken to fix errors, partnering students take slightly more than non-partnering students, but the difference is too small to report.

Partnering students also make fewer errors across all categories. Table 9 displays the differences for error categories that yielded significant results. The table shows that partners make significantly less errors on initial submissions for both non-cosmetic and cosmetic issues. The fact that partnering students produce less Coding Flaw and Testing issues on initial submissions suggests that students working in pairs are able to notice potential bugs more efficiently and are keener on testing their code early on than students working alone. The variance in the remaining error categories suggests that partnering students help each other conform to style standards. It is also interesting to note that differences in error rates eventually even out in final submissions. This suggests that students working individually are solving their errors by the end of the submission process. But, because they spend more time and more submissions on assignments, they are working harder to do so.

6. DISCUSSION

The previous sections report many positive and negative results that vary in their significance. In this section, we summarize and discuss some of the main findings.

Effectiveness of Feedback from Static Analysis Tools. It is clear from Figure 1 that students fix most of the issues reported by the static analysis tools, where the number of errors drops drastically from initial to final submissions. There is also evidence that students might be learning from some of these error messages, as the number of formatting errors drops as students progress in the semester (see Section 4.3). This is consistent with what is reported by Edwards *et al.* [33], but not with what is reported by Keuning *et al.* [34]. Since there were rubric deductions in this study and in Edwards *et al.*'s study for not addressing issues raised by the static analysis tools (but not in Keuning *et al.* [34]), students might be fixing errors reported by the tools, because they want to avoid deductions. However, if these tools were not present, it is not clear whether or not deductions alone would have been enough for students to fix the issues!

Gender Differences. Results in Section 5.4 show that female students make a statistically significant lower number of errors compared to male students in the Documentation and Style categories in initial, intermediate and final submissions. Such differences were not spotted with other student groups. For example, high-performing students, students with more programming experience and students who partner make fewer errors in *all* categories. Moreover, low-performing students and students who do not partner produce more errors in *initial* submissions in the *Coding Flaws* category. This strengthens the hypothesis that female students in the course are more careful with writing comments and code that follows recommended style.

Up to our knowledge, this result is new. Previous work pointed out gender differences in learning programming [38]-[39], debugging [40], interacting with software [41]-[42], performance in CS degrees [43], self-efficacy in programming [44] and programming contests [45]. However, we are not aware of studies showing the presence or absence of gender differences in coding style.

While the result fits a stereotypical view of females being more concerned about aesthetics than males, we should be careful when interpreting the results. For example, the results do not imply that there is a certain programming style distinct to females, nor do they necessarily imply that style is a good predictor of gender. In a study by Carter and Jenkins [46], instructors were asked to identify the gender of students by examining pieces of code randomly selected from student solutions to assignments. Results revealed that the aforementioned stereotype was clearly present in how the instructors attempted to identify the gender of the students. However, empirical results supporting this stereotype (beyond the results reported here) are still absent.

Effect of Partnering. Our results show that students who partnered produced fewer static analysis errors, completed their assignments faster and submitted less to the automatic grader. This result is consistent with the literature on pair programming in introductory programming courses (*e.g.* [47] and [48]). However, the results reported here are different, since they relate to code quality issues that do not affect correctness, while most previous work measured code quality by the number of passed test cases or exam and assignment grades [49]-[50]. As mentioned in Section 3.1, it is important to note that the pair programming protocol was required for students who worked in pairs, but there was no way to enforce it, as students worked outside the classroom environment.

Commenting Behavior. Results showed that Documentation errors are the second most common error type. They stay un-fixed longer than any other type and students receive more of them as the semester proceeds. This is equally the same regardless of the student major, performance in the course or prior programming experience. The only difference was with female students who produced a statistically significant lower number of Documentation errors than males. This pattern of ignoring Documentation errors is also consistent with the results reported by Edwards *et al*. [33]. What is interesting is that Javadoc checks were ignored in this study, while they were the main measure for Documentation in the study of Edwards *et al*. and yet the results were the same. This suggests that these results relate to writing comments in general.

What is interesting about this result is not that the number of Documentation errors is high, as this could be explained by the fact that a distinct error is generated for every missing comment for a field or a method, which means that a student who chooses to comment his/her code at the end would receive *many* Documentation errors for every submission he/she makes except for the last one. What is interesting about the results is that students delay writing comments until the end. It is difficult to tell whether or not this behavior is because students add comments only to avoid mark deductions or

because they believe that it is unimportant to write comments *while* coding. Hence, an implication of these results is that instruction should directly emphasize the importance of commenting in general and that instructors who wish their students to develop the habit of commenting *while* coding should explicitly teach them to do so.

Testing Behavior. Although the assignment instructions required a very simplistic form of testing, the data shows that students seemed to avoid writing such tests until the very end (Section 4.1), most probably to avoid losing the testing points. The results also show that both low- and high-performing students produced a similar amount of testing errors, even though high-performing students produced fewer errors in overall (Section 5.1).

This behavior is likely the result of allowing students to receive feedback from the automatic grader without limits, which made it easier for students to use the automated tests than write tests on their own. Moreover, the course did not teach testing and did not teach or require test-driven development, which could explain why although the same unlimited submission policy is used in [33], their results on testing do not agree with ours. In fact, Edwards argued for teaching Test-Driven Development early in the curriculum to move students away from trial-and-error programming [51]. Another way around this behavior, suggested by Karavirta *et al.* [10], that does not involve formally requiring unit testing or teaching it, is to limit the number of allowed submissions to the automatic grader in the first few assignments, so that students could develop better testing habits at the beginning of the semester.

Association between Performance and Static Analysis Errors. Results showed an association between static analysis errors and performance in two ways:

1. The mere *presence* of errors in *final* submissions was found to be associated with lower performance on *exams* (see Section 4.4). This is interesting, because these errors are not directly related to the material students are tested on during exams. One possible explanation for this result is that students who are content with leaving issues in their final assignment submissions (despite seeing the automated feedback) are lazier students in overall, whose laziness shows also as weaker performance on exams compared to students who make sure to address all the automated feedback they receive. Another plausible explanation is that students who are unable to address all the issues by their final submission are students struggling with programming in overall.
2. Lower-performing students, non-majors and students with less prior programming experience were found all to have *more* errors in their *initial* submissions from the Coding Flaws category.

This suggests that these two indicators are good candidates for further investigation on their ability to predict at-risk students and the possibility of their use as features in machine learning models (see Hellas *et al.* [52] for a review on predicting academic performance in computing education).

7. LIMITATIONS

One of the goals for using the time-to-fix and submissions-to-fix measures was to investigate which errors students struggle with most. These measures have also been used by other researchers for the same purpose (*e.g.* [53] and [14]). However, inferring meaning from results using these measures proved to be tricky. Taking longer time or more submissions to fix an error can plausibly be attributed to the student ignoring the error instead of not being able to fix it. The data can also be distorted by students work habits. For example, students who start early or take breaks while working on the assignment will show longer time-to-fix values than students who complete their work in one sitting. Also, students who tend to rely more on the automatic grader will show higher submissions-to-fix values regardless of whether these submissions were actually attempts to fix the errors.

Results reported by Edwards *et al.* [33] for error categories that take longer to fix were actually almost the reverse of ours. This shows that these measures could be highly sensitive to course-specific factors, like how assignments are graded, what instructions are given to students and what material is emphasized more. These shortcomings of the measure should be kept in mind by researchers using it. For more discussion of time metrics used in the analysis of student programming behavior, see Leinonen *et al.* [54].

The variance in student working habits and submission strategies affects also how error rates per KLOC should be interpreted. As shown in Section 4.1, the vast majority of errors belong to the Formatting, Documentation and Style categories, which students might be ignoring until their last submission, thus compounding the total number of errors. Therefore, more emphasis should be placed on Coding Flaw errors, as they could reflect bugs and misunderstandings that need direct attention. As Edwards *et al.* [33] note, these errors are often masked by the abundance of cosmetic errors which students place less emphasis on. It might be useful to think about how stronger emphasis could be placed on Coding Flaw errors in automated feedback systems to make students keener on fixing them.

Another limitation of this work stems to emanate from how errors were grouped into categories. It is reasonable to expect the results to be dependent on which error groups were used in the study and which errors were assigned to each category. Also, the fact that students received feedback from the static analysis tools *only* through the submission system, rather than by directly running the tools or having them embedded in the IDE, might have affected when and how the students fixed the errors. Finally, this study was conducted at a highly selective school (acceptance rate < 7%), which means that the average student in this study might be academically stronger and more motivated than the average student at other schools. This is clear from the low failure and drop-out rates described in Section 3.1 when compared to the failure rates reported at other institutions [55]. The analysis also excluded students who dropped out from the course (see Section 3.1 and Section 3.2), which must have left some of the lower-performing students unconsidered.

8. FUTURE WORK

While performing the analysis on error categories has provided many insights, future work could benefit from studying individual errors instead of broad categories. Looking at individual errors can provide more fine-grained information on the types of difficulties and misconceptions novices face. This is especially true for errors in the Coding Flaws and Style categories. Errors associated with performance in the course and errors that appear most commonly in student code should be addressed explicitly by instructors in their teaching.

More work can also be done to investigate the effect of different programming languages on the errors students make and their ability to resolve them. Another interesting direction of research is to study how errors reported by the tools can be accounted for in rubrics used by automatic graders and how these tools can report issues in a way that emphasizes potentially serious issues more than other less important issues. This could affect how student respond to the error messages they see.

Finally, more work needs to be done on how students perceive commenting and to better understand the relationship between gender, pair programming and coding style.

REFERENCES

[1] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers and others, "Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies," Proceedings of the 2015 ITiCSE on Working Group Reports, pp. 41-63, [Online], Available: https://doi.org/10.1145/2858796.2858798, 2015.

[2] C. D. Hundhausen, D. M. Olivares and A. S. Carter, "IDE-based Learning Analytics for Computing Education: A Process Model, Critical Review and Research Agenda," ACM Transactions on Education (TOCE), vol. 17, no. 3, pp. 1-26, 2017.

[3] B. Hui and S. Farvolden, "How Can Learning Analytics Improve a Course?," Proceedings of the 22nd Western Canadian Conference on Computing Education (WCCCE'17), pp. 1-6, [Online], Available: https://doi.org/10.1145/3085585.3085586, 2017.

[4] A. Ahadi, A. Hellas, P. Ihantola, A. Korhonen and A. Peterson, "Replication in Computing Education Research: Researcher Attitudes and Experiences," Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16), pp. 2-11, [Online], Available: https://doi.org/10.1145/2999541.2999554, 2016.

[5] N. C. C. Brown, M. Kölling, D. McCall and I. Utting, "Blackbox: A Large-scale Repository of Novice Programmers' Activity," Proc. of the 45th ACM Tech. Symposium on Computer Science Education (SIGCSE '14), pp. 223-228, [Online], Available: https://doi.org/10.1145/2538862.2538924, 2014.

[6] D. Zingaro, Y. Cherenkova, O. Karpova and A. Petersen, "Facilitating Code-writing in PI Classes," Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13), pp. 585-590, [Online], Available: https://doi.org/10.1145/2445196.2445369, 2013.

[7] A. Papancea, J. Spacco and D. Hovemeyer, "An Open Platform for Managing Short Programming Exercises," Proceedings of the 9th Annual Int. ACM Conference on International Computing Education Research (ICER '13), pp. 74-52, [Online], Available: https://doi.org/10.1145/2493394.2493401, 2013.

[8] A. Vihavainen, T. Vikberg, M. Luukkainen and M. Pärtel, "Scaffolding Students' Learning Using Test My Code," Proc. of the 18th ACM Conf. on Innovation and Technology in Computer Science Education (ITiCSE '13), pp. 117-122, [Online], Available: https://doi.org/10.1145/2462476.2462501, 2013.

[9] C. Piech, M. Sahami, D. Koller, S. Cooper and P. Blikstein, "Modeling How Students Learn to Program," Proc. of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12), pp. 153-160, [Online], Available: https://doi.org/10.1145/2157136.2157182, 2012.

[10] V. Karavirta, A. Korhonen and L. Malmi, "On the Use of Resubmissions in Automatic Assessment Systems," Computer Science Education, vol. 16, no. 3, p. 229–240, 2006.

[11] P. Blikstein, "Using Learning Analytics to Assess Students' Behavior in Open-ended Programming Tasks," Proceedings of the 1st International Conference on Learning Analytics and Knowledge (LAK '11), pp. 110-116, [Online], Available: https://doi.org/10.1145/2090116.2090132, 2011.

[12] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper and D. Koller, "Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming," Journal of Learning Sciences, vol. 23, no. 4, p. 561–599, 2014.

[13] A. Allevato and S. H. Edwards, "Discovering Patterns in Student Activity on Programming Assignments," ASEE Southeastern Section Annual Conference and Meeting, [Online], Available: http://people.cs.vt.edu/~edwards/index2.php?option=com_content&do_pdf=1&id=288, Virginia Polytechnic Institute and State University Blacksburg, Virginia, 2010.

[14] A. Altadmri and N. C. C. Brown, "37 Million Compilations: Investigating Novice Programming Mistakes in Large-scale Student Data," Proceedings of the 46th ACM Technical Symposium on Computer Science Education, pp. 522- 527, [Online], Available: https://doi.org/10.1145/2676723.2677258, 2015.

[15] N. C. C. Brown and A. Altadmri, "Novice Java Programming Mistakes: Large-scale Data *vs.* Educator Beliefs," ACM Transactions on Education (TOCE), vol. 17, no. 2, pp. 1-21, 2017.

[16] E. S. Tabanao, M. M. T. Rodrigo and M. C. Jadud, "Predicting at-Risk Novice Java Programmers through the Analysis of Online Protocols," Proceedings of the 7th International Workshop on Computing Education Research, pp. 85-92, [Online], Available: https://doi.org/10.1145/2016911.2016930, 2011.

[17] M. C. Jadud, "Methods and Tools for Exploring Novice Compilation Behaviour," Proceedings of the 2nd International Workshop on Computing Education Research, pp. 73-84, [Online], Available: https://doi.org/10.1145/1151588.1151600, 2006.

[18] M. C. Jadud and B. Dorn, "Aggregate Compilation Behavior: Findings and Implications from 27,698 Users," Proceedings of the 11th Annual International Conference on International Computing Education Research, pp. 131-139, [Online], Available: https://doi.org/10.1145/2787622.2787718, 2015.

[19] Ma. Mercedes T. Rodrigo, T. C. S. Andallaza, F. E. V. G. Castro, M. L. V. Armenta, T. T. Dy And M. C. Jadud, "An Analysis Of Java Programming Behaviors: Affect, Perceptions And Syntax Errors among Low-achieving, Average And High-achieving Novice Programmers," Journal Of Educational Computing Research, vol. 49, no. 3, Pp. 293-325, 2013.

[20] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler and J. Penix, "Using Static Analysis to Find Bugs," IEEE Software, vol. 25, pp. 22–29, 2008.

[21] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl and M. A. Vouk, "On the Value of Static Analysis for Fault Detection in Software," IEEE Transactions on Software Engineering, vol. 32, no. 4, pp. 240-253, 2006.

[22] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix and Y. Zhou, "Evaluating Static Analysis Defect Warnings on Production Software," Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 1-8, [Online], Available: https://doi.org/10.1145/1251535.1251536, 2007.

[23] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum and G. Rothermel, "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach," Proceedings of the 30th International Conference on Software Engineering (ICSE'08), pp. 341-350, [Online], Available: https://doi.org/10.1145/1368088.1368135, 2008.

[24] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," Communications of the ACM, vol. 53, no. 2, pp. 66-75, 2010.

[25] S. A. Mengel and V. Yerramilli, "A Case Study of the Static Analysis of the Quality of Novice Student Programs," Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99), pp. 78-82, [Online], Available: https://doi.org/10.1145/299649.299689, 1999.

[26] S. Nutbrown and C. Higgins, "Static Analysis of Programming Exercises: Fairness, Usefulness and a Method for Application," Computer Science Education, vol. 26, pp. 104–128, 2016.

[27] N. Truong, P. Roe and P. Bancroft, "Static Analysis of Students' Java Programs," Proceedings of the 6th Australasian Conference on Computing Education (ACE '04), vol. 30, pp. 317-325, 2004.

[28] D. Liu and A. Petersen, "Static Analyses in Python Programming Courses," Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), pp. 666-671, [Online], Available: https://doi.org/10.1145/3287324.3287503, 2019.

[29] T. Flowers, C. A. Carver and J. Jackson, "Empowering Students and Building Confidence in Novice Programmers through Gauntlet," Proceedings of the 34th Annual Frontiers in Education (FIE 2004), vol. 1, pp. T3H/10-T3H/13, Savannah, GA, 2004.

[30] M. Hristova, A. Misra, M. Rutter and R. Mercuri, "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students," ACM SIGCSE Bulletin, vol. 35, no. 1, pp. 153–156, 2003.

[31] M. Striewe and M. Goedicke, "A Review of Static Analysis Approaches for Programming Exercises," Proc. of the International Computer-assisted Assessment Conference, Research into E-Assessment (CAA 2014), Communications in Computer and Information Science, vol. 439, pp. 100-113, 2014.

[32] K. Abd Rahman and M. J. Nordin, "A Review on the Static Analysis Approach in the Automated Programming Assessment Systems," Proc. of the National Conference on Programming, [Online], Available:http://www.academia.edu/download/30480008/khirulnizam-reviewonstaticanalysisapproach.

pdf, 2007.

[33] S. H. Edwards, N. Kandru and M. Rajagopal, "Investigating Static Analysis Errors in Student Java Programs," Proceedings of the 2017 ACM Conference on International Computing Education Research, pp. 65-73, [Online], Available: https://doi.org/10.1145/3105726.3106182, 2017.

[34] H. Keuning, B. Heeren and J. Jeuring, "Code Quality Issues in Student Programs," Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17), pp. 110-115, [Online], Available: https://doi.org/10.1145/3059009.3059061, 2017.

[35] T. Delev and D. Gjorgjevikj, "Static Analysis of Source Code Written by Novice Programmers," Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON), DOI: 10.1109/EDUCON.2017.7942942, Athens, Greece, 2017.

[36] T. Schorsch, "CAP: An Automated Self-assessment Tool to Check Pascal Programs for Syntax, Logic and Style Errors," Proceedings of the 26th SIGCSE technical symposium on Computer science education (SIGCSE '95), pp. 168–172, [Online], Available: https://doi.org/10.1145/199688.199769, March 1995.

[37] B. Hanks, S. Fitzgerald, R. McCauley, L. Murphy and C. Zander, "Pair Programming in Education: A Literature Review," Computer Science Education, vol. 21, pp. 135–173, 2011.

[38] L. Murphy, B. Richards, R. McCauley, B. B. Morrison, S. Westbrook and T. Fossum, "Women Catch up: Gender Differences in Learning Programming Concepts," Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE '06), pp 17–21, [Online], Available: https://doi.org/10.1145/1121341.1121350, 2006.

[39] W. W. F. Lau and A. H. K. Yuen, "Gender Differences in Learning Styles: Nurturing a Gender and Style Sensitive Computer Science Classroom," Australasian Journal of Educational Technology, vol. 26, no. 7, [Online], available: https://doi.org/10.14742/ajet.1036, 2010.

[40] N. Subrahmaniyan, L. Beckwith, V. Grigoreanu, M. Burnett, S. Wiedenbeck, V. Narayanan, K. Bucht, R. Drummond and X. Fern, "Testing *vs.* Code Inspection *vs.* What Else?: Male and Female End-users' Debugging Strategies," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08), pp. 617-626, [Online], Available: https://doi.org/10.1145/1357054.1357153, 2008.

[41] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell and C. Cook, "Tinkering and Gender in End-user Programmers' Debugging," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06), pp. 231-240, [Online], Available: https://doi.org/10.1145/1124772.1124808, 2006.

[42] M. Burnett, S. D. Fleming, S. Iqbal, G. Venolia, V. Rajaram, U. Farooq, V. Grigoreanu and M. Czerwinski, "Gender Differences and Programming Environments: Across Programming Populations," Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10), Article no. 28, pp. 1–10, [Online], Available: https://doi.org/10.1145/1852786.1852824, 2010.

[43] I. Wagner, "Gender and Performance in Computer Science," ACM Transactions on Computing Education (TOCE), vol. 16, no. 3, pp. 1-16, 2016.

[44] A. Lishinski, A. Yadav, J. Good and R. Enbody, "Learning to Program: Gender Differences and Interactive Effects of Students' Motivation, Goals and Self-efficacy on Performance," Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16), pp. 211-220, [Online], Available: https://doi.org/10.1145/2960310.2960329, 2016.

[45] M. Fisher and A. Cox, "Gender and Programming Contests: Mitigating Exclusionary Practices," Informatics in Education, vol. 5, no. 1, pp. 47–62, 2006.

[46] J. Carter and T. Jenkins, "Spot the Difference: Are There Gender Differences in Coding Style?," Proceedings of the 3rd Annual LTSN-ICS Conference, [Online], Available: https://kar.kent.ac.uk/id/eprint/13757, 2002.

[47] L. Williams, R. R. Kessler, W. Cunningham and R. Jeffries, "Strengthening the Case for Pair Programming," IEEE Software, vol. 17, no. 4, pp. 19–25, 2000.

[48] C. McDowell, L. Werner, H. E. Bullock and J. Fernald, "The Impact of Pair Programming on Student Performance, Perception and Persistence," Proceedings of the 25th International Conference on Software Engineering (ICSE '03), pp. 602-607, 2003.

[49] N. Salleh, E. Mendes and J. Grundy, "Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review," IEEE Transactions on Software Engineering, vol. 37, no. 4, pp. 509–525, 2011.

[50] T. Dybå, E. Arisholm, D. I. K. Sjøberg, J. E. Hannay and F. Shull, "Are Two Heads Better Than One? On the Effectiveness of Pair Programming," IEEE Software, vol. 24, no. 6, pp. 12–15, 2007.

[51] S. H. Edwards., "Using Software Testing to Move Students from Trial-and-error to Reflection-in-action," Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04), pp. 26-30, [Online], Available: https://doi.org/10.1145/971300.971312, 2004.

[52] A. Hellas, P. Ihantola, A. Petersen, V. V. Ajanovski, M. Gutica, T. Hynninen, A. Knutas, J. Leinonen, C. Messom and S. N. Liao, "Predicting Academic Performance: A Systematic Literature Review," Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITICSE'18), pp. 175-199, [Online], Available: https://doi.org/10.1145/3293881.3295783, 2018.

[53] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian and R. Bowdidge, "Programmers' Build Errors: A Case Study (at Google)," Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pp 724–734, [Online], Available: https://doi.org/10.1145/2568225.2568255, 2014.

[54] J. Leinonen, L. Leppänen, P. Ihantola and A. Hellas, "Comparison of Time Metrics in Programming," Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER'17), pp. 200–208, [Online], Available: https://doi.org/10.1145/3105726.3106181, 2017.

[55] J. Bennedsen and M. E. Caspersen, "Failure Rates in Introductory Programming," ACM SIGCSE Bulletin, vol. 39, no. 2, pp. 32-36, [Online], Available: https://doi.org/10.1145/1272848.1272879, 2007.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 06, No. 03, September 2020.

ملخص البحث:

يمكن أن يُساعد تحليل البيانات المتعلقة بالبرامج التي يكتبها الطّلبة الباحثين في فهم الكيفية التي يتعلم بها المبرمجون المبتدئون، وأن يبين للمعلمين كيف يقومون بتعليم المبتدئين على النحو الافضل. يستكشف هذا البحث كيفية استخدام أدوات التحليل الإستاتيكية للبرامج التي يكتبها الطلبة في محاولةٍ للوقوف على سلوك الطلبة البرمجي وأدائهم في مساقات البرمجة.

تم في هذا البحث تحليل استخدام ثلاث أدوات للتحليل الإستاتيكي في واجبات مقرَّرٍ تمهيدي في البرمجة. وأكدت النتائج ما توصلت إليه دراسات سابقة من أنّ المشاكل المرتبطة بالتنسيق والتوثيق هي المشاكل الأكثر شيوعاً من بين تلك التي وجدت في برامج الطلبة من قبل أدوات التحليل الإستاتيكي، وأنّ هذا الأمر ثابت بصرف النظر عن التخصص والأداء في المقرّر، وأنّ هناك أنواعاً معينة من المشاكل أكثر ارتباطاً بالأداء من غيرها.

كذلك تبيّن أن التكرار الكلي للمشاكل التي تكتشفها أدوات التحليل في واجبات الطالب البرمجية يرتبط بعلامة الطالب النهائية في المقرّر، وأنّ وجود أي نوع من المشاكل في آخر محاولة في كل واجب برمجي يرتبط بضعف الأداء في الاختبارات الغير برمجية في المقرر. من جهة أخرى، اتّضح أنّ الطالبات يرتكبن عدداً أقل من أخطاء التوثيق والأسلوب البرمجي مقارنةً بأقرانهنّ من الذكور، وأنّ الطلبة الذين يعملون بالاشتراك مع غيرهم من الطلبة يرتكبون على وجه العموم أخطاءً أقل مقارنةً بالطلبة الذين يعملون منفردين. كذلك فإن نتائج هذا البحث تطرح أسئلة حول جدوى استخدام بعض المقاييس المستخدمة عادة مع أدوات التحليل البرمجي لتقدير صعوبة تصحيح الأخطاء من جانب الطلبة.