Can Industrial-Strength Static Analysis Be Used to Help Students Who Are Struggling to Complete Programming Activities?

Stephen H. Edwards
Virgina Tech
edwards@cs.vt.edu

David Hovemeyer
York College of Pennsylvania
dhovemey@ycp.edu

Jaime Spacco
Knox College
jspacco@knox.edu

Abstract

Static analysis tools evaluate source code to identify potential problems or issues beyond typical compiler errors. Prior work has shown a statistically significant relationship between the correctness of a student's work and statically identifiable flaws or "code smells" that are likely to indicate programming errors. This paper presents a comprehensive study of this relationship in the context of small programming exercises intended for use in student skill building. We use FindBugs, a static analysis tool that identifies program features that are likely to represent actual bugs in professional software. Our goal is to identify the extent to which FindBugs warnings might help novices struggling to solve short programming exercises. In this study, we ran FindBugs against 149,054 answers submitted by 516 students on 57 drill-and-practice coding exercises. We identify the specific FindBugs warnings that are inversely correlated with correctness. We confirm that presence of these warnings is significantly associated with struggling on an exercise, as indicated by taking more time, making more submissions, and receiving lower scores. Finally, every exercise exhibited answers that trigger these warnings, and 92.4% of students would experience these warnings over a full semester. Our results indicate that static analysis with tools designed for use in industry offers an untapped opportunity to provide hints or suggestions to students who are measurably struggling.

1. Introduction

As our understanding of how to teach computer science matures, many software engineering practices and tools have gradually worked their way into the classroom setting and have seen productive use in education. Integrated development environments, software testing frameworks, and software configuration management tools are all examples of tools originally developed for professionals and aimed at streamlining software development or engineering tasks, but that are now regularly used in many classrooms. Static analysis tools are another example; they have long been used to help ensure adherence to required coding standards, increase consistency to reduce maintenance costs, and identify suspicious or smelly code that might be improved. Static analysis tools have also been used in the classroom for similar reasons, having been employed in automated grading of student assignments to ensure adherence to coding standards, ensure presence of documentary comments, and so on.

Some static analysis issues have already been shown to be strongly correlated with lower scores on programming assignments [1], which suggests that static analysis could provide novices with "hints" or suggestions on where to focus their attention. However, providing more information to novices may not be helpful. We know that novices struggle to master the syntax of programming languages [2], and that providing enhanced compiler errors to novices does not necessarily help novices fix syntax errors [3, 4]. Thus, we need strong evidence that professionally-oriented static analysis tools, which can generate *false positives* are identifying true errors in novice code before we incorporating static checkers into introductory courses.

In this paper, we investigate the static analysis issues identified by FindBugs [5, 6, 7], a static analysis tool designed to identify potential programming bugs in Java programs. While prior work [1] has studied longer assignments, here we investigate static analysis issues in the context of short programming exercises where students practice basic programming skills. The overall aim of our research is to better understand the potential for "off the shelf" static analysis tools such as FindBugs to provide useful feedback to students who might be struggling with any programming activity. To explore this potential, we applied FindBugs to student work from a previously completed semester to determine how FindBugs warnings were related to student answers. While the current investigation applies in the context of small practice problems (where rich data is available for exploration), success here

URI: https://hdl.handle.net/10125/60221
ISBN: 978-0-9981331-2-6

combined with prior results on larger programming assignments [1] suggests continued exploration of the role of industry-standard static analysis tools in introductory programming courses. We aim to answer the following research questions:

RQ1: Which FindBugs warnings would be appropriate for generating hints?

RQ2: Are FindBugs warnings accurate? In other words, are warnings inversely correlated with correctness?

RQ3: Are FindBugs warnings associated with greater struggling on an exercise, in terms of time spent, submissions made, or final score?

RQ4: What is the potential for impacting students with FindBugs-based hints?

The paper is organized as follows. Section 2 discusses related work and how our work attempts to build on previous work. Section 3 describes FindBugs and presents examples of defects it identifies in student code. Section 4 describes the data set we used as the basis for our investigation and the context in which the data was collected. Section 5 presents the results of investigating our research questions. Section 6 presents conclusions and ideas for future work. The Appendix briefly describes all of the FindBugs warnings we found to be meaningful for the exercises in our data set.

2. Related work

Recently there has been increasing interest in using small programming exercises in CS 1 courses. Vahid et al. [8] described redesigning a traditional CS 1 course to use small programming exercises exclusively, with positive results. In our work, we seek to determine whether static analysis can be a useful way to provide feedback to students on small programming exercises.

In many ways, warnings produced by static analysis tools such as FindBugs are similar to compiler warnings and errors. There has been a significant amount of work aimed at improving compiler error messages and understanding how students use (or don't use) compiler diagnostics during programming tasks. Denny et al. [4], Pettit et al. [3], and Becker [9] investigated techniques for enhancing compiler error messages to allow students to understand and use them more effectively, with the different studies observing varying degrees of success. Prather et al. [10] investigated students' interactions with enhanced compiler error messages, finding evidence that students found them valuable. In our work, we are studying static analysis warnings on student code retrospectively. Future work is necessary to understand how novice programmers can use feedback generated by static analysis.

Previous work has investigated applying static analysis to student programs. Truong et al. [11] described a framework for using static analysis to identify poor design, poor coding practices, and overly-complex implementation in the context of small "fill in the gap" exercises. In our work, we are interested in applying a general-purpose static analysis tool commonly used in industry to short, free-form programming exercises. Edwards et al. [1] investigated applying static analysis to a broad collection of programming assignment submissions in several introductory courses to determine what types of static analysis errors students commonly make, and the extent to which they are associated with success. Our work focuses more narrowly on the potential for using static analysis as the basis for feedback generation on short programming exercises in CS1.

A significant amount of previous work has investigated techniques for automated feedback generation for programming exercises. Keuning et al. [12] conducted a systematic review of research on automated feedback techniques, including approaches featuring static analysis.

3. FindBugs

FindBugs [5, 6] is a static analysis tool that analyzes Java bytecode for hundreds of potential bug patterns. Its design philosophy is to use the simplest possible analysis that will identify real bugs while avoiding too many false positives, with a general goal of ensuring that at least 50% of FindBugs warnings identify real defects. Each error pattern in FindBugs was developed based on someone finding a bug in code, building the simplest possible analysis to detect the bug, and then tuning the analysis to ensure that the rate of false positives remained low. Thus, FindBugs tends to issue relatively few warnings when compared to static analysis tools that focus more on coding style (such as CheckStyle [13] and PMD [14]), but the warnings that are generated tend to identify real defects.

Figures 1, 2, and 3 illustrate the types of defects that FindBugs can detect using examples of authentic,

```
List<String> uc = null;
for (int i = 0;
    i < list.size() - 1; i++) {
  ...
  // NP_ALWAYS_NULL
  uc.add(list.get(i).toUpperCase());
}
```

Figure 1. Example of NP_ALWAYS_NULL warning

```
String repeat = "";
int i = 0;
while (i < n) {
  // IL_INFINITE_LOOP
  repeat = repeat + str.substring(n);
}
```

Figure 2. Example of IL_INFINITE_LOOP warning

```
int sum = 0;
int count = 0;
while (input.hasNextInt()) {
  ...
}
// ICAST_IDIV_CAST_TO_DOUBLE
return (sum/count);
```

Figure 3. Example of ICAST_IDIV_CAST_TO_DOUBLE warning

student-written code. These examples come from the data used in this paper. While not every FindBugs warning identifies a real defect, in these three cases, FindBugs identified real correctness issues. Our aim is to explore whether the warnings reported by FindBugs could, in general, be a useful form of feedback for students working on programming exercises.

4. Context

The data we used in our investigation consisted of student work on short programming exercises in *CS 1114: Introduction to Software Design* at Virginia Tech in Fall 2017, a CS1 course in Java. We performed a post hoc analysis on all of the student answer attempts submitted that semester, running FindBugs on each. We obtained IRB approval to analyze this work from a previous semester since it involved only coursework students submitted, the course had already finished, and the only risks to participants were confidentiality concerns. Because the course had already concluded before the study began, there was no opportunity to employ consent forms, and IRB approval included examining the entire set of submitted answers after anonymization through study codes. Because this analysis was performed after the course concluded, students did not receive feedback from FindBugs or any other static analysis tool (beyond the compiler) while developing their answers.

A total of 516 students participated in these exercises. Students consisted of freshmen wanting to be computer science majors, students from other majors working on a computer science minor, and a smaller population of students who are considering studying computer science in some capacity. Virginia Tech offers separate courses for non-majors, so the student population here consists of those who are more serious about pursuing computer science. Students in the course have a very wide range of prior experience, but approximately 60–70% of students already have some programming experience prior to taking this course.

During Fall 2017, students engaged in 57 short programming exercises. Students used CodeWorkout [15] to do the exercises. CodeWorkout is a web-based system where students work on short, drill-and-practice style coding exercises and receive immediate feedback. Students worked on 49 of these exercises as homework outside of class, where they could retry their answers repeatedly until they were satisfied with their score. Students received feedback on how their answers performed on reference tests so they would know immediately whether they had successfully completed each exercise. The expectation was that sufficiently motivated students would be able to achieve 100% correct answers. The remaining 8 questions were given as part of in-class, timed exams where students had more limited feedback, seeing only compiler errors and results on a couple of tests shown as examples in the question. Students also had unlimited opportunities to practice additional exercises at any time throughout the semester, although those optional practice activities are not included in this study. A total of 149,054 submissions were made to these 57 exercises by the students in this course during the period of this study.

Table 1 summarizes the data set. The terminology used here is as follows. A *submission* is a request by a student working on one exercise for CodeWorkout to compile the student's code, execute unit tests, and use the unit test results to score the submission. Because CodeWorkout does not attempt to compile the student's code until a submission is made, the submission may fail to compile. Thus, *compiling submissions* is the subset of all submissions that compile successfully and for

Exercises	57
Students	516
Submissions	149,054
Compiling submissions	78,855
Histories	24,133
Successful histories	20,617
Submissions per history	
Min	1
Median	3
Mean	6.2
Max	133

Table 1. Descriptive statistics for the data set

which unit tests are run. When a student's submission passes all of the exercise's unit tests, it is considered to be correct. A *history* is a sequence of one or more submissions made by one student working on one exercise. A *successful history* is one where the final submission passed all of the exercise's unit tests.

5. Results

In this section we discuss our investigation and findings. We have broken down our four primary research questions into more specific subgoals that are delineated in the following subsections. These results were obtained by applying FindBugs to all of the student answers after the course had completed.

5.1. Which FindBugs Warnings Are Relevant?

> **RQ1**: Which FindBugs warnings would be appropriate for generating hints?

Because FindBugs is designed to identify bugs in code written by experienced programmers, not all of its warning types are necessarily meaningful in the context of student-written code. Thus, we break this RQ into two related subquestions and consider each in turn.

> **RQ1a**: Of the FindBugs warnings naturally occurring in the dataset, which indicate potential coding issues? (We will refer to this set of warning types as the *useful* set.)

FindBugs looks for 424 potential issues partitioned into 9 categories. The 76 FindBugs warnings detected in this dataset belonged to just 5 categories: Performance, Dodgy Code, Bad Practice, Correctness, and Malicious Code. The Correctness category contains warnings with a high probability of identifying a real defect, so we felt that all observed warnings from that category were useful. We determined that warnings in the Performance and Malicious Code categories were not relevant for student programming exercises, so we excluded them. This left the warnings in the Dodgy Code and Bad Practice categories, which we evaluated individually. After inspection we omitted the following:

- IM_BAD_CHECK_FOR_ODD
- DMI_USELESS_SUBSTRING
- RCN_REDUNDANT_NULLCHECK_OF_NONNULL-_VALUE
- NM_METHOD_NAMING_CONVENTION
- BC_EQUALS_METHOD_SHOULD_WORK_FOR-_ALL_OBJECTS
- HE_EQUALS_USE_HASHCODE
- NP_EQUALS_SHOULD_HANDLE_NULL-_ARGUMENT

We found that all of the warning types listed above were either likely to be a component of a correct solution (IM, DMI), were likely to be harmless (RCN, NM), or pertained to APIs not used in any of the programming exercises in the data set (BC, HE, NP).

Eliminating the warnings in the Performance and Malicious Code categories along with the 7 warning types mentioned above, we are left with a *useful* set which includes 55 warning types in three categories (Correctness, Dodgy Code, and Bad Practice). A complete list of the warnings in the *useful* set can be found in the Appendix in Table 5. This set is useful because it shows promise for providing feedback to students as they work, since warnings in the *useful* set naturally occur among student answers and have a reasonable probability to identify programming faults.

> **RQ1.b**: Of the *useful* set, which are most important for a student to correct? (We will refer to this set of warning types as the *strong* set.)

Of the 55 warnings in the *useful* set, we examined which were present in final submissions made by students, and whether those final submissions successfully answered the exercise. This allowed investigating which warnings were associated with students failing to successfully complete an exercise.

15 warnings from the *useful* set were *only* present in incorrect final answers, never in correct final answers:

- ICAST_IDIV_CAST_TO_DOUBLE
- EC_NULL_ARG
- GC_UNRELATED_TYPES
- QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT
- INT_BAD_COMPARISON_WITH_NONNEGATIVE-_VALUE
- IL_INFINITE_LOOP
- NP_ALWAYS_NULL
- UCF_USELESS_CONTROL_FLOW_NEXT_LINE
- DB_DUPLICATE_BRANCHES
- EC_BAD_ARRAY_COMPARE
- EC_UNRELATED_CLASS_AND_INTERFACE
- EC_UNRELATED_TYPES
- DMI_COLLECTIONS_SHOULD_NOT_CONTAIN-_THEMSELVES
- NP_LOAD_OF_KNOWN_NULL_VALUE
- RANGE_STRING_INDEX

These 15 warnings are extremely likely to represent faults, since all final answers containing at least one of these failed one or more reference tests. This suggests that student answers that contain any of these warnings are extremely likely to require corrective changes before they can successfully answer the exercise.

In addition, 8 other warnings from the *useful* set were frequently associated with incorrect final answers, appearing more than 3 times as frequently among incorrect final answers than in correct final answers:

- ES_COMPARING_STRINGS_WITH_EQ
- RCN_REDUNDANT_NULLCHECK_WOULD-_HAVE_BEEN_A_NPE
- IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN
- RV_RETURN_VALUE_IGNORED
- UCF_USELESS_CONTROL_FLOW
- UC_USELESS_OBJECT
- FE_FLOATING_POINT_EQUALITY
- NP_NULL_ON_SOME_PATH

While some student answers with these warnings were able to pass all reference tests, the majority (greater than 75%) of final answers with these warnings were unsuccessful. Together, the 23 warnings listed here form the *strong* set—those warnings that are very important to fix if a student is to produce a successful final answer.

5.2. Are FindBugs Warnings Accurate?

> **RQ2**: Are FindBugs warnings accurate? In other words, are warnings inversely correlated with correctness?

A major challenge when analyzing submission or snapshot data sets is that each submission is based on the previous submission, and is therefore not an independent event. For example, a submission containing an accurate bug warning may be followed by 10 more submissions that also contain the same warning, but the student was in fact working on some other part of the code and never tried to to fix the issue the warning denoted. Thus, we cannot simply count the number of submissions that contain FindBugs warnings.

Instead, we focus on *transition* submissions, which are submission events where either the number of *useful* FindBugs warnings changes, or the number of successful unit tests passed changes. If FindBugs were mainly issuing false positives that did not identify true errors, we would not expect to find a correlation between, for example, submissions that both decrease the number of FindBugs warnings and also pass more test cases, or submissions that increase the number of FindBugs warnings and also pass fewer test cases.

	+FB	0FB	-FB
+TC	508	20,362	1,894
0TC	1,057	21,757	946
-TC	670	7,776	332

Table 2. Transitions analysis. +FB means a snapshot adds an FB warning, -FB removes a warning, and 0FB indicates no changes in warnings. +TC increases the number of passing test cases, -TC decreases passing test cases, and 0TC means no change.

We present the results of this transition submission analysis in Table 2. We can see that submissions that remove FindBugs warnings are five times more likely to pass more test cases than fewer test cases, while submissions that add more FindBugs warnings are about as likely to pass more test cases than fewer. The differences in these observations is significant ($\tilde{\chi}^2$ = 934.25, df = 4, p-value < 0.0001). We also see that the vast majority of the snapshots in the data set neither add nor remove FindBugs warnings, which is consistent with FindBugs' design goal of a low rate of false positives.

The transition submissions analysis suggests that *useful* FindBugs warnings are likely finding real errors in the code, because adding more FindBugs warnings is correlated with less correctness, while removing FindBugs warnings is correlated with more correctness. However, these data and results are not sufficient to know if FindBugs warnings are identifying *important* problems. It could be the case that FindBugs identifies many likely errors, but students fix these errors quickly, or complete the exercise regardless of the presence of the error. Thus, we need to explore whether the warnings are correlated with higher measures of student struggle.

5.3. Are Warnings Associated with Struggling?

> **RQ3**: Are FindBugs warnings associated with greater struggling on an exercise, in terms of time spent, submissions made, or final score?

Part of our underlying hypothesis is that FindBugs warnings might be useful in providing hints to students when they are struggling. However, investigating this aspect requires that we operationalize what it means for a student to "struggle." Here, we have considered three basic measures as indicators of student struggle. First, we believe that if a student is struggling with an exercise, it will require more time than on exercises when they are not struggling. Second, a student who is struggling might also be expected to make more submissions on the exercise. Third, if a student is struggling, they may

finish an exercise with a lower correctness score. These three measurable outcomes are used in the analysis here as indicators of whether a student might be struggling, or at least working harder, on a particular exercise.

Since CodeWorkout only tracks students' explicit attempts to submit a solution, we must estimate work time based on the time between submissions. We assume that any gap between submissions of 10 minutes or more represents two distinct work sessions.

Based on these indicators, we investigate the question of whether FindBugs warnings are associated with greater struggle by testing for differences in the indicators. We first studied the relationship between warnings in the *useful* set and the three outcomes of interest. We then extended our examination to the *strong* subset, to see how it was related to student struggling.

> **RQ3a**: Are *useful* warnings associated with increased struggling?

It is possible that stronger students are more likely to succeed on exercises, spend less time, make fewer submissions, and are also less likely to write exhibiting FindBugs warnings. To control for the strength of the student, we performed a within-subjects comparison using a mixed model repeated measures ANOVA. In this analysis, students represent the subjects, the student's history on each exercise represent the repeated observations, the observations were split into two groups (work histories that contained either no FindBugs warnings, or at least one FindBugs warning), and the time spent on the exercise was the dependent variable. Thus, differences between the students should not be the underlying cause of observed differences, since students are being compared with themselves.

The time taken on an exercise was significantly different ($F(1, 18459) = 4178$, $p < 0.0001$) between the two groups. Histories with at least one *useful* warning took more time (mean 828 seconds, s.d. 886, median 580 seconds) compared to histories without warnings (mean 234 seconds, s.d. 384, median 70 seconds). Figure 4 depicts the difference in the distribution of time spent. The presence of at least one FindBugs warning on any submission in one student's history on one exercise increases the median time to complete that exercise by a factor of more than 8, an enormous difference in effort on relatively short exercises. The effect size for the difference between the means is characterized by a Cohen's d value of 1.14. These results suggest that FindBugs warnings are associated with students taking significantly more time to complete an exercise.

We also examined the number of submissions made using the same model setup. Histories containing *useful* warnings had more submissions (mean 14.9, s.d. 14.0, median 11) than histories without warnings (mean 4.7, s.d. 6.2, median 2), which was significant ($F(1, 24130) = 4641$, $p < 0.0001$) with an effect size (Cohen's d) of 1.21. Figure 5 depicts the difference in the distribution.

Finally, we examined the correctness scores received on the final answers in the histories. Histories where *useful* warnings were present scored lower (mean 85.0%, s.d. 27.9%) than histories without warnings (mean 92.4%, s.d. 23.3%). These differences were significant ($F(1, 4431) = 132$, $p < 0.0001$) with an effect size (Cohen's d) of 0.31. Further, only 69.4% of histories where warnings were present finished successfully with a correct solution, while 87.0% of histories without warnings were successful.

Thus, when students experienced at least one *useful* warning on any submission, this was associated with increased time spent, a greater number of submissions, and a lower score, compared to other histories by the same student without warnings. This suggests that warnings in the *useful* set are associated with student struggle measured with these three outcome indicators.

> **RQ3b**: Are *strong* warnings associated with increased struggling?

We performed the same analysis with respect to the *strong* subset of warnings. We again used a mixed model repeated measures ANOVA, where students were the subjects and exercise histories were the repeated observations. The presence or absence of at least one warning in the *strong* subset was the independent variable representing the primary fixed effect of interest, and time spent was the dependent variable.

The time taken on an exercise was significantly different ($F(1, 12600) = 3086$, $p < 0.0001$) between the two groups of histories. Histories with at least one *strong* warning took more time (mean 1034 seconds, s.d. 1041, median 753 seconds) compared to histories without *strong* warnings (mean 272 seconds, s.d. 448, median 83 seconds). The effect size (Cohen's d) is 1.43. Figure 4 shows the distribution of time spent for histories with *strong* warnings compared to those with no warnings or with *useful* warnings.

As expected, histories containing *strong* warnings had more submissions (mean 18.6, s.d. 16.3, median 14) than histories without (mean 5.4, s.d. 7.2, median 3), which was significant ($F(1, 24476) = 2387$, $p < 0.0001$) with an effect size of 1.53. Figure 5 compares the number of submissions in the two conditions.

Finally, histories with *strong* warnings scored lower (mean 77.8%, s.d. 31.3%) than histories without (mean 92.9%, s.d. 22.4%). These differences were significant ($F(1, 7578) = 332$, $p < 0.0001$) with an effect size (Cohen's d) of 0.65. Only 55.6% of histories where

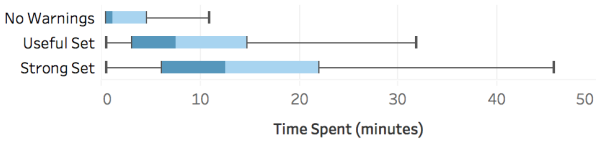

Figure 4. The distribution of time spent when no warnings are present, vs. at least one *useful* warning, vs. at least one *strong* warning.

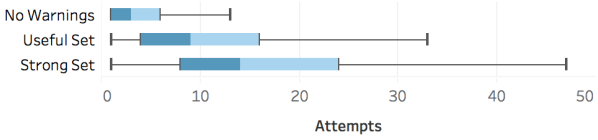

Figure 5. The distribution of submissions made when no warnings are present, vs. at least one *useful* warning, vs. at least one *strong* warning.

warnings were present ended with a correct solution, compared to 87.2% of histories without warnings.

Overall, the differences in terms of time taken, submissions made, and score earned were all more extreme (with larger effect sizes) than with the larger *useful* set of warnings. This corroborates the treatment of the *strong* warnings as being more critical for students to fix, since they are associated with measurably larger amounts of struggling. Based on these results, on exercises where students experienced at least one *strong* warning on any submission, this was associated with increased time spent, a greater number of submissions, and a lower score compared to other histories by the same student without any *strong* warnings.

5.4. Potential Impact

> **RQ4**: What is the potential for impacting students with FindBugs-based hints?

By investigating **RQ1**, we have identified the set of warnings that are appropriate to use for hints, and through **RQ2** we have confirmed that these warnings are associated with correctness of solutions, and by investigating **RQ3** we have evidence that presence of these warnings is significantly associated with student struggling, as indicated by greater time taken, more submissions made, and lower scores earned. However, it is also important to consider the scope of effect that any hints generated based on FindBugs warnings may have.

One concern is that FindBugs warnings are relatively rare across all submissions. Of the 149,054 submissions in this data set, only 7.4% (11,059) generated *useful* warnings. This raises an immediate concern that, while

	Count	%
All submissions	149,054	100.00%
Compiling submissions	78,855	52.90%
With any useful warning	11,059	7.42%
With any strong warning	6,194	4.16%
With Correctness warning	2,790	1.87%
With Dodgy code warning	6,515	4.37%
With Bad practice warning	2,554	1.71%

Table 3. Occurrences of *useful* warning types

	Count	%
All students	516	100.00%
Saw any useful warning	477	92.44%
Saw any strong warning	405	78.49%
Saw any Correctness warning	361	69.96%
Saw any Dodgy code warning	461	89.34%
Saw any Bad practice warning	215	41.67%

Table 4. Student exposure to *useful* warning types

helpful when available, FindBugs-based hints may be too infrequent to help. On the other hand, 85.3% of histories end in successful solutions, and 84.0% of histories involved less than 10 minutes of time taken to reach a solution. In other words, the data suggest that in most cases, students were not struggling, in terms of time taken or lack of success (that is, not achieving a 100% correctness score on the final submission).

Still, the concern that FindBugs warnings are too rare to be useful is important to consider. Thus, we examined how many students, exercises, and histories could potentially benefit from these hints.

> **RQ4.a**: How many students can benefit from FindBugs hints?

Table 3 summarizes the number of occurrences of warnings in the *useful* set. A warning from the *useful* set was only reported on about 7.4% of all submissions, which corresponds to 14.0% of compiling submissions. At first glance, this does not seem like a lot of warnings. However, Table 4 reveals that, of the 516 students in this study, 92.4% would have received a warning from the *useful* set at some point on some exercise. While these warnings are rare across all total submissions, individual students make enough submissions over all of the exercises they attempt that most students encounter at least one *useful* warning at some point.

Furthermore, each student would have seen a *useful* warning on an average of 5.9 exercises. We find 12.8% of submission histories produce at least one *useful* warning. But 84.0% of histories require less than ten minutes of time spent, which suggests that most students

are not struggling most of the time. Indeed, from Section 5.3, we know that histories with *useful* warnings have a median time spent of 580 seconds, or nearly ten minutes, while histories without such warnings have a median time spent of just 70 seconds. There are also significant differences in the number of submissions made and the final score achieved. In other words, there appears to be an overlap between the 12.8% of histories with warnings and those that represent student struggling, which appear to be similarly infrequent.

Additionally, note that the warning types identified by FindBugs were created largely through studying defects that occurred in large codebases written by experienced programmers. As such, FindBugs is not necessarily tuned to detect the types of mistakes made by novice programmers, and the addition of detectors for new novice-code-specific warning types could potentially find a larger set of errors.

> **RQ4.b**: How many exercises could potentially benefit from FindBugs hints?

Finally, we also examined which exercises had at least one submission with *useful* warnings. In this data set, it turned out that *every* exercise had at least one student history that produced a *useful* warning, and that each exercise had on average 54.4 (out of 516) students who would have seen a warning from useful set. As a result, it appears that the FindBugs warnings are general enough that they can identify issues across a wide set of exercises, instead of only certain types.

5.5. Threats to Validity

It is important to consider the factors that may limit the generality of this study's results or may raise questions regarding the validity of the conclusions. One significant issue is the role that individual programmer traits may play in all dimensions of performance, including the likelihood they will trigger certain FindBugs warnings. As stated earlier, it is not only possible, but also even likely, that stronger students are more likely to succeed on exercises, spend less time, make fewer submissions, and are also less likely to write code containing errors identifiable by FindBugs. However, in this study, the primary reason for using the repeated measures mixed model instead of a simpler ANOVA was precisely to perform within-subjects comparisons in order to control for individual differences. That is, our comparisons here are based on comparing one student's exercise histories containing appropriate FindBugs warnings again that *same* student's exercise histories that were warning free when looking for differences in time taken, number of submission attempts, and final score. This analysis strategy ensures the conclusions are not influenced by individual differences between programmers.

However, there are additional issues when interpreting the study's conclusions. First, this work is based on small programming exercises, and may not generalize to other kinds of programming activities. Prior work suggests similar effects may be present in longer programs, which is promising but not conclusive. Broader study in other contexts is needed to confirm these results. Also, this experiment only uses FindBugs' static checks—other static analysis tools or other static checks may produce different results. In addition, there may even be exercise-specific causes, where some exercises simply take more time and also cause students to write more warning-prone code. Because we performed a within-subjects comparison and had only a single history for each exercise by each student, it was not possible to control for problem-specific effects of this nature. A different study would be necessary to assess whether specific problems are more error-prone. Finally, this study was performed using introductory students. While FindBugs was developed for use by professional developers (experts), it does appear to have value in an educational setting with beginners. However, it is unclear what role experience plays in the tendency to write warning-prone code, so the results here cannot be easily generalized to other groups.

6. Conclusions

In this paper, we applied FindBugs to student short exercise answers from a previous semester to determine whether FindBugs warnings are fruitful sources of feedback to help students struggling with incorrect answers. We found a *useful* set of FindBugs warnings with clear potential to serve in identifying hints for students. A subset of these (the *strong* warnings) were even more predictive of failing to complete an exercise if they were not repaired. By using mixed model repeated measures ANOVA tests, we were able to show that the *useful* and the *strong* warnings were significantly associated with longer work times, more submissions, and lower scores.

Even though *useful* warnings were only reported for a small percentage of submissions (around 7.4% of total submissions and 14% of compilable submissions), most students (92.4%) would have encountered such a warning at some point in the course, with each student averaging 6 exercises where at least one submission would have triggered a *useful* FindBugs warning. Furthermore, each exercise had an average of 54.4 students with at least one *useful* warning, or about

10% of the class. The rate at which students would experience these warnings is comparable to the rate at which students appear to struggle on exercises, as measured by time taken, number of submissions, and lower scores. We believe this is good evidence in support of our underlying hypothesis that static analysis may be useful in generating hints to students about problems to fix, and that such hints may nudge students in a more productive direction when they are struggling with a specific practice exercise.

While these results were obtained with small programming exercises by beginners, earlier research on static analysis using PMD and Checkstyle, two other professionally-oriented static analysis tools, found statistically significant relationships between checks that were aimed at detecting behavioral flaws and the correctness of student answers on larger programming projects [1]. As future work, this approach should also be investigated in the context of other programming activities, to establish its degree of generality.

Finally, the next step is to implement a hinting system that will point out likely programming flaws as students work on their answers. Integrating such a hinting system into CodeWorkout offers significant potential to help students, although further study involving live experiments with students using such hints is necessary to confirm effectiveness. The next research goal should be confirming whether students show improved outcomes when receiving such hints.

Acknowledgments

This work is supported in part by the National Science Foundation under grants DUE-1625425 and DRL-1740765. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] S. H. Edwards, N. Kandru, and M. B. Rajagopal, "Investigating static analysis errors in student java programs," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, (New York, NY, USA), pp. 65–73, ACM, 2017.

[2] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, (New York, NY, USA), pp. 208–212, ACM, 2011.

[3] R. S. Pettit, J. Homer, and R. Gee, "Do enhanced compiler error messages help students?: Results inconclusive.," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, (New York, NY, USA), pp. 465–470, ACM, 2017.

[4] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, (New York, NY, USA), pp. 273–278, ACM, 2014.

[5] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, Dec. 2004.

[6] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, (New York, NY, USA), pp. 13–19, ACM, 2005.

[7] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, pp. 22–29, Sept. 2008.

[8] F. Vahid, J. M. Allen, K. Downey, and A. Edgcomb, "Many-small programs in cs 1 using an auto-grading lab system," tech. rep., University of California, Riverside, November 2017.

[9] B. A. Becker, "An effective approach to enhancing compiler error messages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, (New York, NY, USA), pp. 126–131, ACM, 2016.

[10] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. Cohen, "On novices' interaction with compiler error messages: A human factors approach," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, (New York, NY, USA), pp. 74–82, ACM, 2017.

[11] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ACE '04, (Darlinghurst, Australia, Australia), pp. 317–325, Australian Computer Society, Inc., 2004.

[12] H. Keuning, J. Jeuring, and B. Heeren, "Towards a systematic review of automated feedback generation for programming exercises – extended version," Tech. Rep. UU-CS-2016-001, Department of Information and Computing Sciences, Utrecht University, 2016.

[13] "Checkstyle website." `http://checkstyle.sourceforge.net/`, 2018.

[14] "Pmd website." `https://pmd.github.io/`, 2018.

[15] S. H. Edwards and K. P. Murali, "Codeworkout: Short programming exercises with built-in data collection," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, (New York, NY, USA), pp. 188–193, ACM, 2017.

[16] "Findbugs bug descriptions." `http://findbugs.sourceforge.net/bugDescriptions.html`, 2018.

Appendix: *Useful* warning types

Table 5 lists the warning types in the *useful* set along with very brief descriptions of the kind of defect identified by each warning. Warnings in the *strong* subset are also indicated in the table. For more detailed explanations of these warning types, see the FindBugs bug description web page [16].

Strong	Warning type	Description
	Correctness Warnings	
	BC_IMPOSSIBLE_DOWNCAST_OF_TOARRAY	Bad cast of array converted from collection
	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN	return x++; (increment has no effect)
	DLS_OVERWRITTEN_INCREMENT	x = x++; (increment has no effect)
•	DMI_COLLECTIONS_SHOULD_NOT_CONTAIN_THEMSELVES	attempt to add collection to itself
	DMI_INVOKING_TOSTRING_ON_ARRAY	toString() method on array is not useful
•	EC_BAD_ARRAY_COMPARE	equals() method on arrays is not useful
•	EC_NULL_ARG	passing null to equals() method is not useful
•	EC_UNRELATED_CLASS_AND_INTERFACE	equals() comparison on unrelated objects
•	EC_UNRELATED_TYPES	equals() comparison on unrelated objects
•	GC_UNRELATED_TYPES	arg. type vs. collection type param. mistmatch
	ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL	integer-valued double passed to Math.ceil()
	ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND	integer-valued float passed to Math.round()
•	IL_INFINITE_LOOP	infinite loop
	IL_INFINITE_RECURSIVE_LOOP	infinite recursive loop
•	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	int compared with out of range long constant
○	IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN	assignment to parameter without using its value
•	NP_ALWAYS_NULL	dereference of known null value
○	NP_NULL_ON_SOME_PATH	dereference of possibly null value
	NP_NULL_PARAM_DEREF_ALL_TARGETS_DANGEROUS	possibly null value where expecting non-null
	NP_UNWRITTEN_FIELD	dereference of uninitialized field
•	QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT	if (x = true) { ... } (probably meant ==)
	RANGE_ARRAY_INDEX	out of bounds array element reference
•	RANGE_STRING_INDEX	out of bounds string element reference
○	RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE	dereference of value previously compared to null
	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	regular expression uses invalid syntax
	RE_POSSIBLE_UNINTENDED_PATTERN	possibly unintended use of regex metacharacter
○	RV_RETURN_VALUE_IGNORED	method return value ignored, e.g., String.trim()
	RpC_REPEATED_CONDITIONAL_TEST	e.g., if (x == 0 || x == 0) ... (probable typo)
	SA_LOCAL_SELF_COMPARISON	if (x == x) ... (probable typo)
	SA_LOCAL_SELF_COMPUTATION	e.g., foo = x - x; (probable typo)
	UWF_UNWRITTEN_FIELD	field is never assigned a value
	VA_FORMAT_STRING_EXTRA_ARGUMENTS_PASSED	too many arguments passed to format() method
	Dodgy Code Warnings	
	BC_UNCONFIRMED_CAST	source type has variants not castable to dest. type
•	DB_DUPLICATE_BRANCHES	if/else blocks have identical code
	DLS_DEAD_LOCAL_STORE	value stored to local variable is never used
	DLS_DEAD_LOCAL_STORE_IN_RETURN	return statement has assignment to local variable
○	FE_FLOATING_POINT_EQUALITY	equality comparison of floating-point values
•	ICAST_IDIV_CAST_TO_DOUBLE	result of integer division is cast to double
	INT_BAD_REM_BY_1	x % 1, guaranteed to yield 0
•	NP_LOAD_OF_KNOWN_NULL_VALUE	reference to a variable known to contain null value
	NS_DANGEROUS_NON_SHORT_CIRCUIT	use of non-short-circuit operator (& or |) for logic
	NS_NON_SHORT_CIRCUIT	use of non-short-circuit operator (& or |) for logic
	QF_QUESTIONABLE_FOR_LOOP	mismatch from loop condition to updated variable
	RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT	ignored return value of method with no side effect
	SA_LOCAL_DOUBLE_ASSIGNMENT	x = x = 17; (possible typo)
	SA_LOCAL_SELF_ASSIGNMENT	x = x; (possible typo)
	ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	instance method writes to static field
○	UCF_USELESS_CONTROL_FLOW	control-flow has no effect
•	UCF_USELESS_CONTROL_FLOW_NEXT_LINE	control-flow has no effect, possible typo
	UC_USELESS_CONDITION	condition always yields the same result
○	UC_USELESS_OBJECT	object is created and modified, but never used
	UC_USELESS_VOID_METHOD	void method appears to have no side effects
	URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD	public or protected field is never read
	Bad Practice Warnings	
	ES_COMPARING_PARAMETER_STRING_WITH_EQ	comparing string objects with ==, not equals()
○	ES_COMPARING_STRINGS_WITH_EQ	comparing string objects with ==, not equals()

Table 5. Useful FindBugs warnings (marks indicate Strong warnings)