Helping Student Programmers Through Industrial-Strength Static Analysis: A Replication Study

Allyson Senger
asenger@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Stephen H. Edwards
edwards@cs.vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Margaret Ellis
maellis1@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

ABSTRACT

Static analysis tools evaluate source code to identify problems beyond typical compiler errors. Prior work has shown a statistically significant relationship between correctness and static analysis results. This paper replicates and extends a prior study on FindBugs, a static analysis tool aimed at professional Java programmers. The prior study showed a strong link between certain FindBugs issues and problems with program correctness. It also showed they were significantly associated with struggling, as indicated by taking more time, making more submissions, and receiving lower scores. However, the study used small programming exercises involving only a handful of lines of code from one semester of a CS1 course. This replication study uses the same experimental approach, but applies it to full-scale programming assignments from hundreds to thousands of lines in length, across all sections of CS1, CS2, and CS3 at a large public university over a period of 4 academic semesters, and involving 4,244 students in 109 laboratory sections completing 255,222 submission attempts. The goal of this replication study is to confirm how prior results hold up, and to explore how the results apply to full-sized programming assignments. We find a set of FindBugs warnings that are inversely correlated with correctness and confirm that their presence is still significantly associated with struggling on larger programming assignments. However, a larger number of FindBugs issues were identified as valuable. We also discuss how student-friendly messages have been added to provided student-readable feedback for a tool where the native messages were written for professional programmers.

CCS CONCEPTS

• **Applied computing** → Interactive learning environments; • **Social and professional topics** → **Student assessment**; • **Software and its engineering** → Software maintenance tools.

KEYWORDS

CS Education, static analysis, errors, bugs, FindBugs, CS1, CS2

ACM Reference Format:
Allyson Senger, Stephen H. Edwards, and Margaret Ellis. 2022. Helping Student Programmers Through Industrial-Strength Static Analysis: A Replication Study. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499310

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03.
https://doi.org/10.1145/3478431.3499310

1 INTRODUCTION

Static analysis tools have seen more frequent use in classrooms to help grade assignments or inform students of the best programming practices. These tools allow students to receive feedback more quickly than if a human was determining problems with a submission, allowing for a larger number of feedback/edit/resubmit cycles before an assignment is completed. Instead of relying on a teaching assistant or the professor, a student can use feedback from static analysis tools to immediately begin fixing issues.

This paper describes a replication of a prior study on the use of the FindBugs static analysis tool to find issues that are likely to be useful to students in improving the correctness of their answers, further studying this issue in a different context. In 2019, Edwards, Hovemeyer, and Spacco [4] conducted a study of FindBugs warnings on short programming exercises showing a group of these warnings that correlated with incorrect solutions and with indicators of struggling. This replication study uses the same techniques to determine if the results are transferable to full-sized programming projects in a range of Java programming courses from CS 1 through advanced data structures.

As in the original study, indicators of student struggling are time spent, number of attempts, and lower correctness. Programming assignments may result in different work habits such as students fixing more than one problem between submissions. Even if a submission contains a specific warning, that warning may not be the focus of a student's attention a they try to address other issues or missing pieces of their answer. Further, longer program solutions present the opportunity for a greater number of bugs, and usually involve much longer time spans than those of short programming exercises with answers consisting of only dozen lines of code. Because of differences in working strategy, it is important to evaluate FindBugs in the context of larger programming assignments.

The research questions addressed here are the same as in the original study [4], adapted for the context of complete programming assignments:

- **RQ1:** Which FindBugs warnings are relevant to students completing full-size programming assignments?
- **RQ2:** Do the FindBugs warnings correlate with incorrectness in student solutions?
- **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 feedback?

We investigate these questions using four semesters of data from CS1, CS2, and CS3 at a large public university, encompassing 255,222 submissions from 4,244 students. We find that the prior study's results transfer to this new context, although a larger set of useful FindBugs warnings are identified and they occur more frequently, presenting greater opportunities to help students through feedback.

This paper is organized in the following manner. Section 2 covers the related work and an explanation of the static analysis tool, FindBugs, that we will be employing. Section 3 contains the method of study. Section 4 covers the context in which this work occurred. Section 5 gives the results of the study and answers the research questions given both here and in the original paper. Section 6 reports the results of a pilot of providing real time student feedback about FindBugs warnings. Finally, section 7 concludes our findings and explains future directions for the work.

2 RELATED WORK

There are several static analysis tools available that can assist programmers in finding problems, which makes them easier for students to fix [2]. PMD and CheckStyle are static analysis tools that focus heavily on formatting and coding style, while catching fewer coding flaws [5]. Different static analysis tools tend to find different bugs; even if they find the same ones, they tend to find them in different places. FindBugs focuses more on logical errors in programs, which may help students more easily arrive at a correct solution.

Rutar et al. examine common static analysis tools for Java programs. PMD, JLint, and FindBugs all use pattern detection with known errors to find problems in code. FindBugs only focuses on a single method at a time. ESC/Java uses theorem proving, which relies on preconditions and postconditions to find flaws [8].

Truong et al. [12] studied static analysis tools on small programming exercises. The tools compared a student's solution structurally to the instructor-proved answer and used software engineering metrics to check for logic errors. Edwards et al. [5] applied static analysis tools to investigate the most common errors in introductory programming courses. They found that many errors were cosmetic, but errors based on coding flaws were associated with lower scores.

Previous work has studied student understanding of compiler messages. Edwards et al. [4] summarize the work of others focused on understanding compiler messages, which are similar to the warnings provided by static analysis tools. Denny et al. [3], Pettit et al. [10], and Becker [1] examine how compiler messages could be improved in a teaching context because compiler errors are usually designed for professional developers. Prather et al. [11] studied how students interact with modified compiler error messages.

3 FINDBUGS

FindBugs analyzes Java bytecode to identify erroneous bug patterns, rather than depending on observing exceptions or incorrect behavior while the code runs. As in the original study [4], here we focus on the FindBugs warnings in the categories of Correctness, Bad Practice, and Style as being the most relevant. Figure 1 shows examples of problematic code from our data set. These figures illustrate the nature of issues that FindBugs can identify for students to correct.

```
--------------------------------
// ES_COMPARING_STRINGS_WITH_EQ
--------------------------------
String suffix = newHour.substring(...);
if (suffix == "pm") {
    hour = hour + 12;
}

--------------------------------
// EC_BAD_ARRAY_COMPARE
--------------------------------
byte[] idByteFromFile = new byte[idLength];
if (idByteFromFile.equals(sequenceIDToFind.getBytes()))
{
    ...
```

Figure 1: Examples of FindBugs warnings

4 METHOD

This paper replicates a prior study by Edwards, Hovemeyer, and Spacco [4] that focused on short-form programming exercises, where answers consisted of around a dozen lines of code or fewer. These short exercises were similar to what is in homework sets or drill-and-practice assignments. Their study covered 516 students in 57 short programming problems. It used FindBugs to identify the warnings most associated with submission problems. The prior study identified a *useful set* of 55 FindBugs warnings that occurred in student answers and presented good opportunities for providing feedback that will help students correct issues. Over the course of the prior study, 90% of students saw a useful FindBugs warning. In addition, the prior study identified a subset of 23 of these useful warnings as a *strong set* that occurred much often in incorrect submissions. Students spent significantly more time completing answers, using a greater number of submission attempts, and achieved lower correctness scores when FindBugs warnings in the strong set occurred.

This replication study applies the same experimental approach to full-scale programming assignments that are typical of out-of-class programming activities completed over one or several weeks of time, ranging in size from hundreds to a few thousand lines of source code. While the prior study only included CS1 exercises, this replication study includes all sections of CS1, CS2, and CS3 at a large public university over a period of 4 academic semesters. This replication study is much larger in scale, involving 4,244 students in 109 laboratory sections completing 255,222 submission attempts over the 4 academic semesters included in the data set. As in [4], we consider both individual submissions, or attempts electronically submitted by the student for one assignment, as well as the entire *history* of submissions or attempts made by one student on a single assignment. While some of these courses included weekly lab assignments, only out-of-class programming assignments were included in the study. All compiling submissions were run through FindBugs. Table 1 summarizes this data set.

Web-CAT was the automated grader used to collect student submissions in this study. Web-CAT evaluates student programming submissions on coding style, unit testing, and correctness. When

Projects	39
Students	4,244
Submissions	255,222
Compiling submissions	245,160
Histories	20,968
Successful histories	15,005

Table 1: Descriptive statistics for the data set

students submit to Web-CAT, they receive a report containing the code coverage of their own test cases, the results from running those tests, and an estimation of the problem coverage of their solution. The grader gives hints on where problem coverage can be increased and marks lines of student code that contain problems identified by static analysis. Prior to this study, Web-CAT supported the use of both PMD and Checkstyle for static analysis checks on student code. As part of this study, we added support for running FindBugs on submissions as well, and integrated messages resulting from FindBugs warnings into Web-CAT's existing feedback results.

5 RESULTS

5.1 Which FindBugs Warnings Are Relevant?

RQ1: Which FindBugs warnings are relevant to students completing full-size programming assignments?

FindBugs can identify 424 specific issues in 9 categories [6]. The original study found 76 of these FindBugs warnings naturally occurred in its data set, and by examining them, decided to only include FindBugs warnings from the three categories of Correctness, Bad Practice, and Dodgy Code. The original study listed 7 additional warnings that were also excluded as either being likely to be part of a correct solution, as addressing APIs not covered in the exercises, or as being harmless. This produced a total of 55 distinct FindBugs warnings in the original study, termed the *useful set*.

In this replication, we adopt the same exclusions as the original study. However, we observed a larger range of FindBugs warnings. We found 185 distinct FindBugs warnings across all student submissions. This larger number is likely the result of the programs being longer and involving more combinations of and interactions between programming language features. While this superset of the original paper is broader in scope, it was not surprising in this new context. As in the prior study, we examined the additional warnings and excluded 37 as being part of APIs not covered in the courses, likely to be part of a correct solution, or harmless. This produced a new *useful set* of 148 FindBugs warnings that formed the basis of our analysis. The full list of useful warnings identified here is available online at http://anonymous.url/findbugs-useful-set.csv.

Which of the useful warnings are most important for a student to correct? For each warning in the useful set, we examined all submissions exhibiting a warning to determine which were behaviorally correct or not—passing all instructor reference tests for the assignment. The original study identified a subset of 23 useful warnings that were more strongly associated with incorrect submissions based on the proportion of submissions with the warning that were incorrect, using a threshold of 75% to identify warnings that were 3 or more times as likely to appear in incorrect submissions than in correct ones. In our study, 128 out of the 148 useful errors exceeded this threshold, which is close to the complete useful set. By examining the data, it became clear that on larger programming assignments, students make a larger number of submissions, and the majority of these (in excess of 70%) are not yet completely correct. This makes sense due to the scale of the assignments. In the prior study, students made fewer submissions, and their submissions were shorter, leaving fewer opportunities to observe warnings.

To address this issue, we also looked beyond the submissions containing a given warning to consider all submissions that were incorrect. By viewing the warnings as indicators of incorrect submissions, and by considering the full population of incorrect submissions in the entire data set, it is possible to calculate precision, recall, F1 scores [7] for each warning. F1 scores serve as a fitness measure that balances both the interests of precision (minimizing false positives) and recall (minimizing false negatives). Sorting the warnings by F1 scores allows us to prioritize the warnings that are present in the largest numbers of incorrect submissions.

As in the previous study, we also found a number of warnings that *only* occurred in incorrect submissions. We found 27 such warnings, compared to 15 in the prior study. However, many of these warnings occurred in very few submissions, with a quarter only arising in a single submission each. In fact, only 4 such warnings appeared in 25 or more submissions, with the others having extremely low recall and F1 scores. Although the prior study retained all such warnings, here we only consider the 4 that were present in at least 25 submissions each (out of 245,160, or at least 0.01%). Among the remainder of the useful set, we chose to focus on those that were present in at least 500 incorrect submissions (0.2% of all compiling submissions), resulting in a *strong set* of 59 FindBugs warnings.

Table 2 lists the FindBugs warnings in the *strong* set. The "Subs" column shows the number of incorrect submissions containing each warning. A total of 315,074 submissions contained at least one strong set warning, with 11,149 containing only other warnings in the useful set. Thus, the strong set represents 96.6% of the submissions with any warnings of interest.

5.2 Are FindBugs Warnings Accurate?

RQ2: Do the FindBugs warnings correlate with incorrectness in student solutions?

To examine whether FindBugs warnings are related to correctness, we performed a chi-squared test with both course and presence or absence of warnings in the strong set as factors, with the dependent variable being whether or not a submission passes all reference tests. The presence of strong warnings was significantly associated with a decrease in correct submissions (chi square = 584.5, $p < 0.0001$), with 39.8% of submissions without strong warnings being correct, and just 23.0% of submissions having strong warnings being correct. Course level was also significant (chi square = 647.2, $p < 0.0001$), with only 14.5% of CS3 submissions being correct, compared to 40-42% of CS1 and CS2 submissions.

Subs	Warning type	Description
341	*Warnings only present in incorrect submissions (at least 25)*	
196	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	regular expression not correct
56	EQ_ALWAYS_TRUE	equals() method always returns true, non-symmetric
46	VA_FORMAT_STRING_BAD_CONVERSION	argument does not match format string specifier
43	MF_CLASS_MASKS_FIELD	field has same name as visible field in superclass
314733	*Warnings present in* > 500 *incorrect submissions*	
60504	DLS_DEAD_LOCAL_STORE	variable assigned but not read or used
31380	ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	instance method writes to static
19546	RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT	method has no effect other than ignored return
17957	EQ_COMPARETO_USE_OBJECT_EQUALS	class defining compareTo() uses Object.equals()
17780	ES_COMPARING_STRINGS_WITH_EQ	string comparison with ==
14584	EC_BAD_ARRAY_COMPARE	equals(Object o) used on array (compare mem. address)
13929	UWF_UNWRITTEN_FIELD	field is never assigned a value
9666	RR_NOT_CHECKED	ignored return of InputStream.read()
9403	UWF_NULL_FIELD	all writes to field are null
8503	NP_NONNULL_PARAM_VIOLATION	null param. not allowed or always deref.
7989	NP_UNWRITTEN_FIELD	dereference of uninitialized field
6952	UC_USELESS_OBJECT	object is created and modified, but never used
5990	NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT	equals(Object) should check for null argument
5677	NP_NULL_ON_SOME_PATH_EXCEPTION	exception path guarantees deref. of null
5345	ES_COMPARING_PARAMETER_STRING_WITH_EQ	param. string comparison with ==
4939	RV_ABSOLUTE_VALUE_OF_RANDOM_INT	absolute value of Integer.MIN_VALUE is negative
4903	SF_SWITCH_NO_DEFAULT	switch statement has no default case
4723	BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS	equals(Object o) should not assume type for o
4647	EQ_GETCLASS_AND_CLASS_CONSTANT	equals() breaks in subclass due to class literal
4207	EC_UNRELATED_TYPES	equals(Object) compares two different classes
3864	NP_NULL_ON_SOME_PATH	branch guarantees deref. of null value
3527	RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE	deref. of value previously compared to null
3525	UC_USELESS_CONDITION	condition always yields the same result
3507	UR_UNINIT_READ	read to uninitialized field
3175	ICAST_IDIV_CAST_TO_DOUBLE	division args. not cast to double
3118	BC_VACUOUS_INSTANCEOF	instanceof test always returns true
3000	RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	return value not checked
2133	FE_FLOATING_POINT_EQUALITY	bad comparison of floating points
1904	ICAST_INTEGER_MULTIPLY_CAST_TO_LONG	int multiplication could overflow
1798	EQ_SELF_USE_OBJECT	defines covariant version of equals()
1794	DMI_INVOKING_TOSTRING_ON_ARRAY	toString() method on array is not useful
1784	UC_USELESS_VOID_METHOD	void method appears to have no side effects
1665	DB_DUPLICATE_BRANCHES	conditional branches with identical code
1655	NP_NULL_PARAM_DEREF_NONVIRTUAL	possible null value passed to non-null method param.
1615	OS_OPEN_STREAM	stream not closed, file descriptor leak
1509	IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN	param. ignored and overwritten
1423	SA_FIELD_DOUBLE_ASSIGNMENT	e.g., field_x = field_x = 12; (probable typo)
1353	DLS_DEAD_LOCAL_STORE_SHADOWS_FIELD	unused variable with same name as field
1325	BC_UNCONFIRMED_CAST	cast not checked, could fail
1190	EQ_DOESNT_OVERRIDE_EQUALS	subclass does not override equals() from superclass
1147	NP_LOAD_OF_KNOWN_NULL_VALUE	reference to previously null value
1103	RV_RETURN_VALUE_IGNORED	method return value ignored, e.g., String.trim()
952	URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD	public or protected field is never read
925	SF_SWITCH_FALLTHROUGH	switch needs breaks or returns
823	UUF_UNUSED_PUBLIC_OR_PROTECTED_FIELD	public or protected field is never used
743	RpC_REPEATED_CONDITIONAL_TEST	e.g., if (x == 0 || x == 0)...(probable typo)
701	CO_SELF_NO_OBJECT	likely unintentional covariant compareTo() defined
691	NS_DANGERGOUS_NON_SHORT_CIRCUIT	non short-circuit logic in conditional
666	DMI_HARDCODED_ABSOLUTE_FILENAME	File object uses absolute pathname
657	NP_ALWAYS_NULL	deref. of known null value
652	NP_NULL_PARAM_DEREF	deref. of null param. or null param. not allowed
601	NP_TOSTRING_COULD_RETURN_NULL	toString() can return null
554	EC_NULL_ARG	passing null to equals() method is not useful
545	IL_INFINITE_RECURSIVE_LOOP	infinite recursive loop
525	SA_FIELD_SELF_ASSIGNMENT	e.g., field_x = field_x; (probable typo)

Table 2: FindBugs warnings

5.3 Are Warnings Associated with Struggling?

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

We use the same indicators of struggling as the original study [4]. There, while there is no direct measure of struggle, three separate measures are treated as indicators of struggle because they are expected as consequences of struggle. If students are having difficulty, they may take more time on a project, make more submissions to the automated grader, or have lower correctness scores when they finish the assignment. As a result, we investigate relationships between warnings and all three.

Students work on assignments across multiple work sessions or sittings. As in the original study, we estimate the full work time on a project by adding the elapsed times between successive submissions. To account for extended breaks between separate work sessions, we discount gaps that are excessively long. We use 3 hours between submissions to indicate a break between work sessions, an inter-session delay used in prior research studies in similar contexts [9]. 86% of submissions occur within less than 3 hours. Any gap between submissions greater than 3 hours was removed and replaced with the average interval between submissions for that student on that assignment to get a more accurate estimate of total time taken.

Final submissions containing a warning from the *strong* set had an average working time of 5.83 hours across the entire assignment while those without a *strong* set warning only took 4.59 hours. This indicates that students who still had *strong* warnings in their final submissions took more time on the assignment. Since this is one of the indicators used in the original study for struggling, longer times may indicate possible struggle. A mixed model repeated measures ANOVA was used to perform a within-subjects comparison on students, with different assignments as repeated measures. Both the course and the presence or absence of strong warnings were used a factors. All three factors (course, student, and presence of at least one strong warning anywhere in the submission history) were significant, with the presence of a strong warning anywhere in the history being associated with longer time spent ($F(1, 15354) = 334.4$, $p < 0.0001$). Results for the full useful set are similar, since the strong set accounts for 96.6% of warnings within the useful set.

Students who had *strong* warnings in the final submissions also required more submissions. Those without warnings in the final submission had 10.18 submissions on average with those with warnings had 14.89 on average. Strong warnings in a final submission were a statistically significant factor in total number of submissions a student made to that assignment, using the same mixed model ANOVA approach ($F(1, 20181) = 57.7$, $p < 0.0001$). Students may have been struggling to fix code which in turn required more submissions to complete the assignment. Adding in FindBugs warnings to feedback could help students diagnose problems more quickly and require fewer submissions.

Lastly, students with *strong* warnings upon assignment completion passed 89.77% of reference tests while those without passed an average of 92.75% ($F(1, 20047) = 72.5$, $p < 0.0001$, Cohen's $d = 0.14$). While this difference is significant, it is very small.

Overall, submissions without *strong* warnings in the final submission take less time, require fewer submissions, and earn slightly higher scores. Because *strong* warnings are associated with all three indicators, as in the original study, this also suggests that strong warnings are more often present when students may be struggling.

	Count	%
All students	4,244	100%
Saw any useful warning	3,928	92.55%
Saw any strong warning	3,908	92.08%
Saw any Correctness warning	4,051	95.45%
Saw any Style warning	3,848	90.67%
Saw any Bad practice warning	2,976	70.12%

Table 3: Student exposure to useful warning types

5.4 Potential Impact

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

Like the original study [4], by investigating RQ1, we have identified the set of warnings that are appropriate to use in providing student feedback to help students, 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 how many students experience these issues.

Unlike the original study, FindBugs warnings appear more frequently in this study's larger programs. Of the 255,222 submissions, 56% generated at least one *useful* warning, compared to just 7.4% in the original study. On this basis, students could potentially receive helpful feedback on more than half of their submissions for any assignment. Additionally, while 71% (15,005) of the histories eventually end with a correct solution, the other 29% still include useful FindBugs warnings. These higher rates are most likely a consequence of the larger size of programming assignments compared to short practice exercises. Table 3 summarizes the number of students who experienced warnings in the useful or strong set. Providing feedback on these warning has the potential to help nearly all students at some point, across a very large number of assignments, regardless of course level or difficulty.

5.5 Threats to Validity

The validity of the conclusions found in this study could be threatened by several things. First, the calculation of the total time a student spent on an assignment relied on estimating the amount of time spent between submissions. While we opted to average the times under 3 hours for each student to estimate their "typical" submission rate, substituting this average for longer gaps to account for breaks between separate work sessions, it is possible that students spent more or less time working between submissions. We would also expect that students may work for more time before initial submissions than those toward the end of the history, which mainly constitute cosmetic PMD and CheckStyle warnings.

Second, unlike the original study in the CodeWorkout context, it is more difficult to tell which warnings students are trying to resolve between submissions. In the original paper, submissions often only had 0 or 1 warnings which made associations with correctness simpler. In the context of full-scale programming assignments, the larger pieces of code with a greater number of warnings on each submission makes it more challenging to determine whether the presence or absence of any one of the errors has clear consequences.

Third, while here we use three indicators of struggle to assess whether FindBugs warnings may be associated with student struggle, following the original study, there is no direct measure of struggle used. While FindBugs warnings do appear to be associated with the three indicators, and the combination of all three indicators clearly suggests a similar association with struggle, none of the statistical tests involve direct measures of struggle and so this association is only indirect through the three indicators of time taken, number of submissions, and final correctness score achieved.

6 FEEDBACK IMPLEMENTATION

Based on the original set of *useful* warnings from [4], we wrote custom FindBugs feedback messages to try to make them more easily understood by students with the aim of providing these automatically to students. Since this is just a portion of the *useful* set identified the current study, we used FindBugs' built-in messages for any warning without a custom message. Some of the warnings, such as UC_USELESS_CONDITION, had FindBugs messages we thought were well worded. Figure 2 show examples of rewritten warning messages that students could receive from FindBugs warnings. We then modified the grading plugin used for our courses on Web-CAT to run FindBugs on each student submission and include the corresponding messages in its output to students.

We collected data from a summer semester where students received these warnings. The data came from 4 projects also used in typical semesters, from both a CS1 class and a CS2 class. The average number of FindBugs warnings per submission was lower in all of the summer assignments, possibly indicating that the students were able to fix problems between submissions because of the notification that problems were present. The summer submissions had an average of 2.2 bugs still appearing in the final submission, while the data from past semesters for those projects have 3.8 on average. The summer assignments had an average work time (after removing down time as explained in Section 5.5) of 3.5 hours; the assignments from past semesters had an average work time of 4.6 hours. This could show that students in the summer had less trouble fixing warnings because they were shown and explained which led to faster completion. However, this would require additional research during regular semesters to reach conclusions. The summer histories had an average of 7.04 submissions while the past semester histories had an average of 11.39 submissions. Finally, summer had an average of 11.29 bugs per history while past semesters had an average of 91.48; this is likely because we were not alerting students of FindBugs issues in previous semesters.

7 CONCLUSIONS

This paper reports on a replication of a prior study in a new, expanded context. The intent of the original study was to determine if

```
--------------------------------
// ES_COMPARING_STRINGS_WITH_EQ
--------------------------------
The == compares the memory addresses of two Strings.
Use equals() instead.

--------------------------------
// RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT
--------------------------------
Immutable objects like Strings are not updated when a
method is called on them. If you want the updated
value, you will have to assign the result of the method
to a variable.

--------------------------------
// EC_BAD_ARRAY_COMPARE
--------------------------------
The equals() method compares the memory addresses
of arrays instead of the contents of the arrays.
Consider using Arrays.equals() or write your own
loop comparison.
```

Figure 2: Examples of student-friendly messages

FindBugs could be a potential way to provide feedback to students in the context of small programming exercises; it determined that FindBugs was likely to be helpful in addressing and identifying problems in student code. In this paper, we have shown that the same applies in the context of full-scale programming assignments across CS1, CS2, and CS3. Students who were unable to resolve FindBugs warnings in our strong and useful sets made more submission attempts, took longer time to complete assignments, and achieved slightly lower correctness scores. These indicators suggest that FindBugs warnings may also be indicators of students struggling. Because so many students experience these warnings on 56% of all submissions, providing explicit feedback in an automated grader could be extremely useful, providing even more assistance than using this approach on small programming exercises. Our preliminary experiences adding these messages to the Web-CAT autograder by customizing the grading plugin indicate this feedback could be helpful for helping students in correcting coding problems so they struggle less.

In the future, we aim to write custom feedback messages for the warnings in the useful set identified in this paper, adding to the custom messages we produced from the results of the previous study. The new set of useful warnings is nearly three times the size of the set identified in the prior study. These messages will give students more help with the warnings most prevalent and troublesome in programming assignments.

It will also be essential to evaluate the affect FindBugs feedback has on students in a future study since as of now we only have small summer classes for reference. We would also like to investigate which FindBugs warnings most accurately predict struggling students. Hopefully, creating updated messages for warnings highly associated with struggling will prevent students from spending excessive time on specific coding problems.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DRL-1740765. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. ACM, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[2] Tomche Delev and Dejan Gjorgjevikj. 2017. Static analysis of source code written by novice programmers. In *2017 IEEE Global Engineering Education Conference (EDUCON)*. 825–830. https://doi.org/10.1109/EDUCON.2017.7942942

[3] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. ACM, New York, NY, USA, 208–212. https://doi.org/10.1145/1999747.1999807

[4] S. Edwards, Jaime Spacco, and David Hovemeyer. 2019. Can Industrial-Strength Static Analysis Be Used to Help Students Who Are Struggling to Complete Programming Activities?. In *HICSS*.

[5] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 65–73. https://doi.org/10.1145/3105726.3106182

[6] FindBugs bug descriptions 2021. FindBugs bug descriptions. http://findbugs.sourceforge.net/bugDescriptions.html.

[7] Peter A. Flach and Meelis Kull. 2015. Precision-Recall-Gain Curves: PR Analysis Done Right. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 838–846.

[8] J. S. Foster, C. B. Almazan, and N. Rutar. 2004. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Los Alamitos, CA, USA, 245–256. https://doi.org/10.1109/ISSRE.2004.1

[9] Michael S. Irwin and Stephen H. Edwards. 2019. Can Mobile Gaming Psychology Be Used to Improve Time Management on Programming Assignments?. In *Proceedings of the ACM Conference on Global Computing Education* (Chengdu,Sichuan, China) *(CompEd '19)*. Association for Computing Machinery, New York, NY, USA, 208–214. https://doi.org/10.1145/3300115.3309517

[10] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. ACM, New York, NY, USA, 465–470. https://doi.org/10.1145/3017680.3017768

[11] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 74–82. https://doi.org/10.1145/3105726.3106169

[12] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static Analysis of Students' Java Programs. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30* (Dunedin, New Zealand) *(ACE '04)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 317–325. http://dl.acm.org/citation.cfm?id=979968.980011