An Empirical Analysis of Blind Tests

Kesina Baral and Jeff Offutt
Department of Computer Science
George Mason University
{kbaral4,offutt}@gmu.edu

***Abstract*—Modern software engineers automate as many tests as possible. Test automation allows tests to be run hundreds or thousands of times: hourly, daily, and sometimes continuously. This saves time and money, ensures reproducibility, and ultimately leads to software that is better and cheaper. Automated tests must include code to check that the output of the program on the test matches expected behavior. This code is called the *test oracle* and is typically implemented in assertions that flag the test as passing if the assertion evaluates to true and failing if not. Since automated tests require programming, many problems can occur. Some lead to false positives, where incorrect behavior is marked as correct, and others to false negatives, where correct behavior is marked as incorrect. This paper identifies and studies a common problem where test assertions are written incorrectly, leading to incorrect behavior that is not recognized. We call these tests *blind* because the test does not see the incorrect behavior. Blind tests cause false positives, essentially wasting the tests. This paper presents results from several human-based studies to assess the frequency of blind tests with different software and different populations of users. In our studies, the percent of blind tests ranged from a low of 39% to a high of 95%.**

I. Introduction

Software engineers have been automating tests for decades. However, in the past 10 years, industry has been steadily increasing its use of test automation [1]. This is driven partly by efficiency advantages of automation [2], and partly by agile processes such as test driven development [3], which require tests to be automated. The underlying driver is, not surprisingly, economics. *Fail watch* [4] analyzed English language news articles for a year and found 606 recorded software failures that impacted half the world's population and cost a combined $1.7 trillion US dollars.

An *automated test* is a software component that includes test input values and test evaluation code, known as *test oracles* (*TO*) [5], usually written as assertions. Automated tests execute against the software under test (*SUT*) and report whether the execution passed (the actual output was the same as the expected output as encapsulated in the TO) or failed (the behavior was incorrect). Automated tests run in build systems that run daily, hourly, or continuously. Unfortunately, many automated tests are incorrect. Some result in false positives, where the test incorrectly reports the software passed. Others result in false negatives, where the test incorrectly reports the software failed. This paper reports on research investigating a particular problem with false positives, where incorrect TOs [6] do not notice a failure in the SUT.

A test oracle is *incorrect* if it sometimes reports the wrong result. TOs can be incorrect for many reasons. This paper focuses on a particular type of TO that is incorrect because it is incomplete. This is a widespread but little studied problem that we call *blind tests*. A *blind test* has an incorrect assertion such that a portion of the output was not observed, leading to an incorrect result. The incorrect result could be a *false positive*, that is, the software produced at least one output value that was incorrect, but the test assertion did not check that particular output. The incorrect result could also be a *false alarm*, that is, the test caused the program to behave correctly but the expected output in the assertion did not include everything that was in the actual output.

For example, consider the following Java statements:

```
String firstName = "Anita";
String lastName = "Borgg"; //faulty spelling
assertEquals("Anita Borg",
     firstName+" "+lastName); //correct
assertEquals("Anita", firstName); //blind
assertEquals("Anita",
     firstName+" "+lastName); //false alarm
```

The first assertion is correct and will reveal the misspelling in `lastName`. The second does not check the last name, thus is blind to the mistake. The third checks both `firstName` and `lastName` in the actual output portion of the assertion, but only includes the first name in the expected output, so raises a false alarm even if the value for `lastName` is correct.

Assertions are simple and easy to implement for trivial computations, but are more problematic even for modest procedures. Consider sorting a simple numeric array, for example "`sort([5,3,4])`." Checking the entire output array ("`assertEquals([3,4,5], result);`") does not scale to arrays with thousands of elements. The expense is prohibitive, thus the automated test may only check part of the output. Spot checking the first or last element will miss many potential mistakes that may be buried in the middle of the array. Even checking that the result is in correct order will not find failures such as "`[3,3,3]`" and "`[0,0,0]`". Simple assertions are tempting to write and common even for experienced testers, but always have the potential to be at least partially blind. This paper focuses on blind tests that lead to false positives.

This paper starts with background in test automation and motivates the problem in section II. We then describe several empirical evaluations of test oracles in section III, followed by observations and results in section IV. Threats to validity are discussed in section V, and related work in section VI. This work opens the door to numerous future work directions as

discussed in section VII.

II. Background in Test Automation

This section introduces theoretical and practical concepts in test automation. More details can be found in textbooks [3], [7]. In test automation, *controllability* [8] refers to how hard it is to get the right inputs to the desired location in the program. When testing a Java method in isolation, most inputs are through parameters, which can be controlled directly. Thus, the method has fairly high controllability. When testing a specific *if*-statement inside a large program through its external UI, however, the tester must find input values that eventually cause the program to reach that *if*-statement, and the variables referenced in the test must have the appropriate values. In this situation, controllability is usually low (making test automation harder). *Observability* [8] refers to how hard it is to see the results of a test. Again, when unit testing a method, the return value and any values printed are easy to observe. However, if the software writes to a database, controls an external sensor, or changes a shared memory object in a distributed web application, observability is harder. Many theoretical and practical challenges of test automation are due to low controllability, low observability, or both.

A. Components of a test case

Test cases are comprised of several pieces. *Test inputs* are inputs needed to complete an execution of the system under test (*SUT*) [7]. Test inputs are sometimes based on test criteria or some other strategy that yields specific test requirements. Test inputs might be sequences of method calls to an object or subsystem, including all necessary objects, parameters, and resources, or user-level inputs such as text values or UI selections. The specifics depend on the software under test, but the concepts are the same. Automated tests must also include *expected results*, which are the results the test produces if the program satisfies its intended behavior. A *test oracle* (TO) compares the expected results with the actual results to decide if a test passes. TOs are commonly implemented as assertions in frameworks such as JUnit [9]. Consider the following example JUnit test. The integers '2' and '3' are test inputs, and the expected result is '5'. The TO is the entire assertion, which directs JUnit to print the string if the assertion returns false.

```
@Test public void testAdd()
{
   assertTrue("testAdd incorrect",
              5 == Calc.add(2, 3));
}
```

One goal of a test is to expose *faults*. Test inputs can trigger a fault to result in an external *failure*, or incorrect behavior. That is, a SUT fails when actual results do not match expected results.

B. The RIPR model

The distinction between fault and failure led to the development of the reachability, infection, and propagation model in the 1980s [10], [11], [12], [13]. The use of assertions in automated tests meant that the tests did not always check the erroneous outputs, thus RIP was extended to include *reveal*, making the RIPR model [14].

Figure 1 illustrates the standard RIPR model from the testing field. TOs reveal faults by observing the *final program state*, that is, outputs and visible values after execution. To detect a fault, a test has to *reach* a faulty location, then the faulty code must *infect* the internal program state with incorrect values. An incorrect value must *propagate* to an **incorrect final state** (a failure). We cannot check the entire final output state, so TOs look at part of the final state (*observed*). If the **incorrect final state** and the **observed final state** intersect, then the test reveals the failure. If they do not, then the test does not reveal the failure. That is, the test is blind. Tests are expensive to design, create, automate, and run, effort that is largely wasted when tests can't see failures.

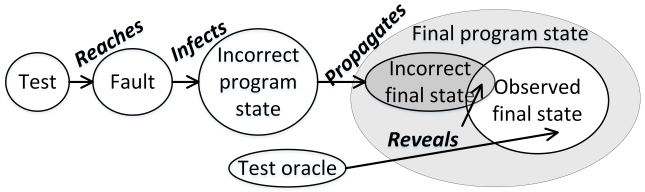

Fig. 1. The Reachability, Infection, Propagation, and Revealability Model

C. Testability

Freedman described testability as a property of an easily testable program [8]. He described these properties as the ability to generate finite numbers of non-redundant sets of test cases that can easily locate faults in the program and do not have input-output inconsistencies. He further defines domain testability as the lack of input output inconsistency using two concepts: controllability and observability.

A software component is said to be *controllable* if we can produce desired outputs from specified inputs [8]. Controllability measures how easy it is to provide inputs to a software component. Outputs that depend on environmental factors such as humidity, temperature, or motion sensors decrease controllability.

The ability to see the output is called *observability* [14], [8]. When software creates output that is hard to observe, for example, large data files, databases, signals to external hardware devices, and messages to remote computers, this is called an *observability* problem. Although widely known in practice, observability problems have been inadequately studied by researchers.

Writing correct oracles is hard for complicated software. In our education and consulting, we have observed that even experienced programmers and testers often make mistakes.

D. Challenges of creating test oracles

To write correct and complete TOs, programmers and testers need to understand program requirements and correct

behavior. It is challenging to write oracles for software that produce complicated outputs, and especially so if the output is non-deterministic. Consider an example SUT that accepts a collection of strings, then returns strings randomly, one at a time, when asked. Given the example partial test below:

```
@Test public void threeRandoStrings()
{
   rs.add("ICST");
   rs.add("ICSE");
   rs.add("FSE");
   String rando = rs.getRandomValue();
   // what assertion should be used?
}
```

What should be checked in an assertion? Correct behavior could be any of the three strings, so perhaps the assertion could check whether `rando` is equal to any of the three strings. However, that still would not ensure the strings are returned randomly. Checking for randomness requires some complex math—the test method would need to call `getRandomValue()` many times and check whether the distribution of strings returned is truly random. Such assertions are hard to think of, hard to design, and hard to code.

III. Empirical Evaluations of Test Oracles

Our first question about blind test oracles is how common are they? Beyond their frequency, we seek to understand what causes tests to be blind. Answers to these questions will help find solutions to the problem. We have carried out several studies of test oracles to try to quantify, analyze, and categorize blind tests. Partners in industry and attendees of Google's Test Automation Conference [15] have frequently expressed that poor TOs are a major problem in test automation. In a 2010 keynote presentation, Patrick Copeland of Google claimed poor TOs was one of the most significant causes of wasted effort in testing [16]. During previous research with automated tests [14], we found that almost a third of the tests in our study were blind.

For this research, we ask the following research questions about blind tests:

- (RQ1) What percentage of tests are blind?
- (RQ2) Do some groups of testers write more blind tests than other groups?

A. Methodology

We collected three separate sets of data on this question, using different subject testers, different object programs, and with different guidance for creating test input values. In each, we started with a program module to test (the SUT) and asked engineers to create automated tests. The SUTs had a known collection of software faults.

We introduce the concept of a *complete test oracle* (CTO) to be a test oracle that checks the entire output space of the SUT. While a CTO may be too large to always be practical, this does give us a ground truth for the test—if the test caused a failure, the CTO is guaranteed to reveal that failure. In general, CTOs may need to include files on disk, databases, internal state variables, etc. In our studies, we narrowed the output space to focus on just the console output of the program. We replaced the engineer-written TO of each test with our own CTO. We then compared the result of executing the test with the original TO against the result with the CTO. If the original test did **not** reveal a failure, but the modified test did, then the TO in the original test was blind.

B. Study 1: Preliminary analysis

Table I shows data from a preliminary study that was primarily focused on automating the creation of test input values [14]. The tests were designed and implemented by hand from UML statecharts that described the behavior of the software. More details are in the previous paper. Out of a total of 93 automated tests that reached a fault, caused an infection in the program state, and propagated to the output state, only 65 of the TOs **revealed** the failure. That is, 28 tests were successful in that they caused a failure, but their TOs incorrectly reported that the test passed. Put another way, 30.11% of the failure-finding tests were ignored because the TOs were incorrect.

Not only is this a surprisingly high percentage of faulty TOs, but when broken out by groups, the success rate was surprising. Row UG in table I represents undergraduate students, FG represents full-time graduate students with no work experience, PG represents full-time software engineers who are also part-time graduate students, and FT represents full-time software testers. In our preliminary study, the testers who wrote the most successful TOs are the least experienced (undergraduate students), while the testers who wrote the most faulty TOs are the most experienced—full-time software testers.

These preliminary results, while based on only a small data set, were intriguing enough to convince us to investigate more thoroughly. Thus, we expanded this study with more subjects.

C. Study 2: Students in a fourth year testing class

The second study assigned students to write tests for a 100 line method they were already familiar with.

1) Subject and program selection: For this empirical study[1], we needed subjects who were familiar with the RIPR model and could write JUnit tests. Therefore, we collected tests written by 48 undergraduate students at our university who were taking a senior-level course in software testing. All were either Computer Science or Software Engineering majors and had studied the RIPR model in class. Approximately one-third of the students had some experience as software engineering interns in local companies.

The goal of the study was to compare and assess the quality of test assertions, so we needed all participants to write tests for the same program. The program was based on a quiz scheduling Java application used to schedule retakes of weekly quizzes in their class. Students enter the course number, then pick from a list of retake times available within the following

[1]All experimental object programs, including faults, and tests are available in our experimental repository: *https://github.com/Keshina/BlindTest*

TABLE I
STUDY ONE: TESTS THAT CAUSED FAILURE BUT DID NOT SEE THE FAILURE

	tests that caused failure	tests that revealed	% tests that revealed	caused failure, did not reveal	% did not reveal
Total	93	65	69.89	28	30.11
UG	30	26	86.67	4	13.33
FG	25	19	76.00	6	24.00
PG	21	12	57.14	9	42.86
FT	17	8	47.06	9	52.94

two weeks. The list includes the date, time, location, and quiz id for each available quiz. The list of retake times form part of the input space and is read from two XML files. The students are then prompted to enter their name and the quiz id. The software saves these inputs to schedule a retake. Figure 2 shows the example initial screen[2]. Students wrote tests for a 100 line method within this program that prints the list of available retakes.

2) Program requirements: The students used this program for the class, so were familiar with its behavior. The version we provided had known faults. We describe key requirements here, but omit some details for brevity. The course features weekly quizzes, and students are allowed to take an alternate version of a quiz within two weeks for 80% of the possible grade.

- Requirement 1: Students can only retake a quiz within 14 days from the date of first attempt. The 14-day period is counted from the date and time of the in-class quiz. The two week retake period can be extended if there is a break week during that period.
- Requirement 2: The application shows retake opportunities available within the next two weeks from the current date and time. The application also prints a message indicating the last day of the period (for example, "*Today is THURSDAY, FEBRUARY 28. Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 14*").
- Requirement 3: A *skip week* is a week with no classes, such as spring break. There are no retake opportunities or quizzes during skip weeks.
- Requirement 4: If there is a skip week within the next two weeks, the software extends the "currently scheduling" period by an additional week.
- Requirement 5: If there is a skip week within the next two weeks, the software extends the retake period to three weeks. For example, for a quiz originally given on February 19, if February 20 through February 27 is a skip week, the last retake date is extended from March 5 to March 12.
- Requirement 6: If a skip week is coming up, the application informs the student there will be no retakes or quizzes during that week. This message is shown between the last retake option before the break and the first retake option after the break.

[2]The actual scheduler is a web application, but students were given a command line version to allow for students who had not programmed Java servlets.

3) Known faults: The program had five known faults, all of which occurred naturally when the program was developed. The subjects had access to the program requirements, but were not told the software had faults. The faults are as follows:

1) The faulty method only checked if the last day of the two week scheduling period is in the skipped week, but does not check if the current day is in the skipped week or if there is a skip week between the current day and the last day of the period. This fault violates requirements 1, 2, and 4.
2) The last retake day for a quiz is not extended when there is a skip week between the in-class quiz and the last retake day for that quiz. This fault violates requirements 1 and 5.
3) A retake is available on the 14th day after the in-class quiz, but **after** the quiz time. That is, the retake is available for 14 days plus several hours. This violates requirement 1.
4) The message about skipped week is printed between the wrong retake options. This violates requirement 6.
5) The application prints retake opportunities during the skip week. This violates requirement 3.

4) Data collection: The Java program was given to the students and they submitted their JUnit tests electronically. The students wrote their tests in teams of three or four students; we consider each team to be one subject. We collected 137 automated tests from 23 subjects. The subjects were unaware of the study being conducted and the collected data were de-identified before analysis to ensure anonymity. The quality of the tests, including the test oracles, was **not** measured as part of their grades.

5) Test evaluation: To ensure we could associate each failure with its triggering fault, each fault was embedded into a separate version of the program. Thus we had six versions of the program. We also created complete test oracles (CTOs) that checked the entire output state of the program. We used the CTOs to create second versions of each test. The CTO versions of the tests have the property that if the test resulted in a failure they are guaranteed to reveal the failure, even if the original test's oracle did not.

We ran each test, and its alternate, on each faulty version and the original program (137*2*6 = 1644 executions). We instrumented the program to track whether each test reached, infected, propagated, and revealed the failure. Comparing

```
************************************************************
University quiz retake scheduler for class Software Testing
************************************************************

You can sign up for quiz retakes within the next two weeks.
Enter your name (as it appears on the class roster),
then select which date, time, and quiz you wish to retake
from the following list.

Today is THURSDAY, FEBRUARY 28.
Currently scheduling quizzes for the next two weeks,
until THURSDAY, MARCH 14
RETAKE: THURSDAY, FEBRUARY 28, at 10:00 in Mason Hall
   1) Quiz 4 from TUESDAY, FEBRUARY 19
   2) Quiz 5 from TUESDAY, FEBRUARY 26
RETAKE: TUESDAY, MARCH 5, at 15:00 in EB 5321
   3) Quiz 4 from TUESDAY, FEBRUARY 19
   4) Quiz 5 from TUESDAY, FEBRUARY 26
   5) Quiz 6 from TUESDAY, MARCH 5
RETAKE: WEDNESDAY, MARCH 6, at 15:30 in EB 4430
   6) Quiz 5 from TUESDAY, FEBRUARY 26
   7) Quiz 6 from TUESDAY, MARCH 5
RETAKE: THURSDAY, MARCH 7, at 10:00 in Mason Hall
   8) Quiz 5 from TUESDAY, FEBRUARY 26
   9) Quiz 6 from TUESDAY, MARCH 5
```

Fig. 2. Quiz retake scheduler screen

results of an original test with its CTO version let us identify tests that caused failure but did not reveal.

D. Study 3: More undergraduate students

Our third study used a different group of undergraduate students who took the same senior-level testing course in a different semester. The subjects were similar and the procedure the same as in study 2, however this study used a different object program.

The object was a small calendar program that prints a month in calendar format given two integers representing a month and a year. The output created from the inputs "10 2019" is shown in figure 3.

```
     October 2019
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Fig. 3. Calendar output

We hand-seeded five faults, as described below. The program requirements and the implementation are much simpler than the quiz retake scheduler. As such, we expected students to infect, propagate, and reveal the faults with greater frequency.

1) Seeded faults: We seeded the following five faults into the program.

1) Fault 1: The program had no input validation, and threw run-time exceptions if non-integer values were entered, or if a month value less than 1 or greater than 12 was entered. Negative values for the year return reasonable results, although the program does not switch to a Julian calendar if a year before 1582 is entered. (This fault was naturally occurring.)
2) Fault 2: In the leap year calculation, the predicate
   `"((year%4 == 0) && (year%100 != 0))"`
   was changed to
   `"((year%4 == 0) || (year%100 != 0))"`
3) Fault 3: We changed the number of days in August from 31 to 30.
4) Fault 4: In the computation for which day of the week a particular date falls on, we changed the number of months from 12 to 10.
5) Fault 5: We changed the number of days in February in leap years from 29 to 30.

2) Data collection and test evaluation: In this study, we used the same process as in study 2 in section III-C. We collected 184 tests from 30 subjects and embedded each fault into a separate program. This required 184*2*6 = 2208 executions.

TABLE II
STUDY TWO (STUDENTS): NUMBER OF TESTS THAT REACHED EACH CONDITION IN THE RIPR MODEL FOR THE FIVE FAULTS–137 TOTAL TESTS

	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	Total
Reachability	83	78	78	80	78	397
Infection	5	4	4	3	11	27
Propagation	5	4	4	3	11	27
Reveal	2	2	0	1	3	8

TABLE III
STUDY TWO (STUDENTS): FREQUENCY OF PROPAGATING FAILURES THAT WERE REVEALED

	Propagated	Revealed	% of tests that revealed after propagation	% of tests that did **not** reveal after propagation
Fault 1	5	2	40.00%	60.00%
Fault 2	4	2	50.00%	50.00%
Fault 3	4	0	0.00%	100.00%
Fault 4	3	1	33.33%	66.67%
Fault 5	11	3	27.27%	72.73%
Average	5.4	1.6	29.62%	70.38%

TABLE IV
STUDY THREE (STUDENTS): NUMBER OF TESTS THAT REACHED EACH CONDITION IN THE RIPR MODEL FOR THE FIVE FAULTS–184 TOTAL TESTS

	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	Total
Reachability	175	113	113	113	113	627
Infection	63	28	6	13	54	164
Propagation	63	27	6	13	54	163
Reveal	63	18	3	5	27	116

TABLE V
STUDY THREE (STUDENTS): FREQUENCY OF PROPAGATING FAILURES THAT WERE REVEALED

	Propagated	Revealed	% of tests that revealed after propagation	% of tests that did not reveal after propagation
Fault 1	63	63	100.00%	0.00%
Fault 2	27	18	66.67%	33.33%
Fault 3	6	3	50.00%	50.00%
Fault 4	13	5	38.46%	61.54%
Fault 5	54	27	50.00%	50.00%
Average	32.6	23.2	61.03%	38.97%

E. Study 4: Professionals at a software engineering company

Our third study used engineers at a local software company. They included a mix of developers and full time testers, who volunteered their time without compensation. Most were early-career professionals 5 to 7 years out of college. All had college degrees in computer science or software engineering. We used the same Java program as in study 2 and shared the same requirements with the testers. The professionals wrote the JUnit tests in small teams of 1 to 5. We collected 14 automated tests from 5 subjects. Unlike the previous students study, the subjects were aware a study was being conducted. The data were anonymized before being analyzed. The rest of the study was the same as study two.

IV. OBSERVATIONS AND RESULTS

The quality of the test oracles from our student subjects in study 2, as shown in tables II and III, was even lower than in the preliminary study. Out of the 137 tests, an average of 79.4 reached the faulty location, between 3 and 11 tests caused the fault to infect the program state and propagate to output, and fewer than 4 tests revealed the faulty behavior. Fault #3 was not revealed by any test. It's also interesting to note that all infections propagated to output, that is, no faults were masked.

Table III breaks out the percentage of tests that caused a failure (propagated an infected state) and that also revealed the failure. At the high end was fault #2, for which failures were revealed half the time, and at the low end was fault #3, for which none of the four failures were revealed.

In testing, a test is considered to be successful if it causes the software to fail. Yet, overall in this study, 70% of the "successful" tests did not reveal the failures that they found—that is, they were blind.

Study 3 used a simpler program with more straightforward behavior. It was based on an assignment from an introductory programming course. Unlike the quiz retake scheduler, calendar only reads two integer inputs, and does not read or write to an external file. The logic is also less complicated and the faults were less "subtle," that is, a higher percentage of input values would result in failure.

Thus it is not surprising that more tests caused a failure, and that a higher percentage of tests revealed the failures (61%).

The results from professional software engineers were less encouraging. Table VI shows that a higher percentage of tests reached the faults (86%), and a higher percentage created an infected program state and propagated to incorrect output (20%). Yet only one test for one fault revealed the failure that it caused. As with Table III, Table VII shows the percentage of failure-causing tests that revealed. In this study 95% of the tests that caused failure were blind. As in the preliminary study, the students in studies 2 and 3 created better test oracles than the professionals in study 4.

Taken together, these studies are clear and convincing that

TABLE VI
STUDY FOUR (PROFESSIONALS): NUMBER OF TESTS THAT REACHED EACH CONDITION IN THE RIPR MODEL FOR THE FIVE FAULTS–14 TOTAL TESTS

	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	Total
Reachability	14	12	11	11	12	60
Infection	4	2	2	2	4	14
Propagation	4	2	2	2	4	14
Reveal	0	0	0	0	1	1

TABLE VII
STUDY FOUR (PROFESSIONALS): FREQUENCY OF PROPAGATING FAILURES THAT WERE REVEALED

	Propagated	Revealed	% of tests that revealed after propagation	% of tests that did not reveal after propagation
Fault 1	4	0	0%	100%
Fault 2	2	0	0%	100%
Fault 3	2	0	0%	100%
Fault 4	2	0	0%	100%
Fault 5	4	1	25%	75%
Average	2.8	0.2	5%	95%

blind tests are a major problem for test automation. Many tests are wasted because their test oracles are incorrect. Not only does this waste valuable resources, but we lose the ability to improve our software by correcting the faults. That is, our software is less reliable.

We cannot be certain why students' tests consistently performed better than professionals' tests. Our working theory is that test automation has only recently been widely taught at universities (often in early programming courses). Thus current students learned test automation with both theory and practice, while many professionals learned JUnit syntax on the job without deep study. This is pure speculation, however, and we hope that further work can shed more light on this question.

A. Root causes of blind tests

After identifying blind tests, we analyzed the root cause for why each test missed its failure. We then grouped them into four categories. We describe them, with examples, below.

1) *Code reuse*: We found one cause of blind tests to be code reuse. Some testers reused test assertions from a previous test, but without appropriate changing the assertion for the new test. Other testers reused the implemented source code when putting expected outputs into the test oracle. If testers reuse assertion without carefully analyzing the new test inputs, it is easy to create blind tests.
   For example, one fault in the Calendar application incorrectly assigned 30 days to August instead of 31.

```
int[] days = { 0, 31, 28, 31, 30, 31, 30,
31, 30, 30, 31, 30, 31 };
```

   Some testers also used 30 instead of 31 while creating tests, essentially copying the expected output from the actual output. Thus they had the expected output as:

```
expectedOutput = " S  M Tu  W Th  F  S\n" +
                 "             1  2  3\n" +
                 " 4  5  6  7  8  9 10 \n" +
                 "11 12 13 14 15 16 17 \n" +
                 "18 19 20 21 22 23 24 \n" +
                 "25 26 27 28 29 30 \n";
```

2) *Misunderstood program requirements*: Some tests were blind because the tester misunderstood what the correct program behavior should be. Although these test oracles may have observed the correct part of the output space, the assertions were written with incorrect behaviors.
   For example: Requirement 4 in study 2 states that if there is a skip week within the next two weeks, the software should display an extended retake period. A tester coded the following into a test:
   - `startSkip = "2019-3-1";`
   - `endSkip = "2019-3-7";`
   - `expectedOutput = "Today is THURSDAY, FEBRUARY 28.  Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 14."`
   - `actualOutput = "Today is THURSDAY, FEBRUARY 28.  Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 14";`
   - `assertEquals(expectedOutput,actualOutput);`

   Since the program's output matched the expected output, the test was considered to have passed. Unfortunately, the tester misunderstood the point of the *skip week*. The correct output should have been:
   - `correctOutput = "Today is THURSDAY, FEBRUARY 28. Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 21";`

   As a result, the test was incorrectly marked as passing.
3) *Technical inexperience*: We also observed that testers sometimes were not familiar enough with the programming language or the testing tool. Programmers who had difficulty understanding the code and did not understand how the test assertions worked had difficulty setting up test inputs and writing test assertions. This led to the creation of test oracles that either did not observe the output space correctly or observed an unimportant section of the output space. This problem was **not** unique to students; but also happened with professional developers and testers.
   The subject programs used in studies 2, 3, and 4 print to standard output (using the Java *out.println* method). Printed output can be captured using the `System Rules` API or the `ByteArrayOutputStream` class, but some testers did not know how, did not capture outputs correctly, and thus did not see failures. For example, one tester rewrote the Calendar program so that instead of printing the output, it returned a string as an output. Another tester simply wrote no assertions at all, so missed **all failures**.
4) *Lack of testing knowledge*: We found that some subjects were confused about how to design a test oracle, and thus created blind tests. For example, some tried to validate the implemented code instead of verifying its correctness through testing. In fact, they assumed the implemented code was correct and the goal of testing is to make sure that it works without crashing.
   For example: one fault in Calendar was that it assigned the number of days in February to be 30 instead of 29 in leap years.

```
if (month == 2 && isLeapYear(year))
   days[month] = 30;
```

   Some of our testers copied the assertions from the output of the incorrect program:

```
expectedOutput = " S  M Tu  W Th  F  S\n" +
                 "             1  2 \n" +
                 " 3  4  5  6  7  8  9 \n" +
                 "10 11 12 13 14 15 16 \n" +
                 "17 18 19 20 21 22 23 \n" +
                 "24 25 26 27 28 29 30 \n";
```

V. Threats to Validity

This section summarizes threats to validity and what we did to mitigate them. Some subjects might not have known, or not

fully understood the RIPR model, leading to incomplete tests. Since this study relies on the model to assess test quality, we mitigated this threat by recruiting subjects who were familiar with the RIPR model. On average, just 71% of the total test cases reached each fault. The remaining 29% did not reach a fault, thus could not be analyzed for test oracle quality.

The overall number of subjects and test cases is another threat to validity. The study required quite a bit of hand analysis, including creating the complete test oracles and determining why some test oracles did not succeed. This limited the total number of subjects and test cases that could be used. A follow-up study that took a simpler approach would be less precise, but might be able to analyze more tests. Another potential threat is the programming language. We used Java, and it is possible that the results might be different with other programming languages.

Although we gave subjects detailed specifications of the application under test, it is possible that some did not understand the expected behavior.

Finally, we observed that some of the students did not put out as much effort as might be hoped. This probably affected the quality of the test inputs more than the quality of the test oracles, the true target of the study.

VI. Related Work

Bertolino et al. acknowledged test oracle efficiency and effectiveness as one of the challenges in software testing and urged further research in the topic [17]. More recently, Ma'ayan studied 112 Java repositories and documented problems with their unit tests [18]. Tests that mask the existence of faults are unreliable and often useless. A *flaky test* is one type of unreliable test [19], [20], [21]. In this paper, we identify blind tests as another type of unreliable test. Vahabzadeh et al. conducted an empirical study of faults in open source test code [22]. They studied faults that were reported and fixed in the code repository, but did not identify additional faults. They used the term "silent horror test bugs" to describe tests that incorrectly passed (a subset of our blind tests). However, since these are the hardest faults to identify (why look at a passing test?), and Vahabzadeh et al. only looked at faults that were found and fixed, they were not able to measure the prevalence of blind tests.

Several tools and metrics have been introduced to help testers develop better test assertions. Xie et al. developed a tool, *Orstra*, to improve automatically generated unit test suites by automatically generating test assertions through regression [23]. Song et al. proposed an eclipse plugin, UnitPlus, which recommends relevant methods to check variable state while writing test assertions [24]. Staats et al. proposed an oracle creation method that ranks variables based on their fault finding capability and monitors those through tests [25]. Loyola et al. proposed a similar approach for automatic oracle creation by selecting a set of variable to be monitored based on the interactions and dependencies observed among the variables [26]. Schuler et al. demonstrated that test oracle quality is an important factor in gauging test quality [27]. They introduced the concept of *checked coverage* as a measure of test oracle quality. Checked coverage looks at the statement that contributes to results assessed by the test oracle.

Zhi et al. reported a case study on the inadequacy of test assertions [28] with results that generally agree with ours. Our study differs from their study in several ways: (1) They analyzed tests found in three open source projects, whereas this study is done on tests written by developers. (2) They used mutation operators to generate faults whereas we used both real-life faults that existed naturally in the program and hand-seeded faults. (3) They used mutation location and statement coverage to determine if the fault was found. We used a more precise method—whether the test reported failure on the fault.

Xie and Memon [29] analyzed GUI tests, finding that oracles that check the entire state can detect more faults. They used manually seeded faults.

Staats et al. [30] define *oracle soundness* for a program, test case, and oracle as: if an oracle is true for a program with a test case, the specification holds for the program when running the test case. However, this is not always true, as the oracle might observe a subset of the program output or final program state and miss some failures.

Similar to test adequacy criterion proposed by Koster et al. [31], our study judges the quality of a test based on its *soundness*. A key difference is that Koster et al. looked at whether tests checked all the affected variables or not; whereas this study checks whether a test hits all four steps in the RIPR model to determine if a test is good enough or not. In our study, if a test reaches the fault, creates an error state, and propagates it to output, yet the test assertion cannot reveal it, then the quality of test is considered to be poor.

VII. Conclusions and Future Work

The most important conclusion about this study is that blind tests are very common. This leads to waste and inefficiency, and contributes to the billions of dollars lost every year on faulty software [4]. Although not as widely studied as flaky tests [32], blind tests may be a more common problem.

We are continuing to identify and classify root causes of blind tests. The list in section IV-A is a strong starting point, but as yet incomplete.

As we continue to grow our understanding of blind tests, we will address the problem in three directions. First will be to educate software engineers, including developers and testers, as to how to properly write effective test oracles. We expect to develop educational materials and evaluate their value on students as well as professional software engineers.

Next, we plan to develop techniques to automatically detect TO problems. This will be done with both static and dynamic analysis. Static analysis techniques such as slicing can be used to identify parts of the output domain that can be modified by specific test inputs. If those output values are not checked, the TO could be blind. CTOs can be generated dynamically and used sporadically. CTOs are normally too expensive to use every time a test is run, but they could be used occasionally

to compare with the existing TO. If the outputs differ, the TO probably has blind spots.

Finally, we hope to adapt automatic program repair (APR) techniques to test oracles. This context is smaller than general APR and much of what a test oracle should check can be determined through analysis techniques such as slicing and data flow.

VIII. Acknowledgements

We thank our industrial partner for providing access to corporate resources.

References

[1] J. Dunn, "Career trends of software test automation engineering in 2019," Online, February 2019, https://dzone.com/articles/career-trends-of-software-test-automation-engineer, last access August 2019.

[2] S. Palamarchuk, "The true ROI of test automation," Online, 2019, https://abstracta.us/blog/test-automation/true-roi-test-automation/, last access August 2019.

[3] L. Koskela, *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Greenwich, CT: Manning Publications Company, 2008.

[4] W. Platz, "Software fail watch: 5th edition," Online white paper, 2017, https://www.tricentis.com/resources/software-fail-watch-5th-edition/, last access: August 2019.

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[6] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Better testing through oracle selection," in *Proceedings of the 33rd International Conference on Software Engineering (NIER Track)*, ser. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, May 2011, pp. 892–895.

[7] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, UK: Cambridge University Press, 2017, ISBN 978-1107172012.

[8] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, June 1991.

[9] E. Beck and K. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 37–50, July 1998.

[10] L. J. Morell, "A theory of error-based testing," Ph.D. dissertation, University of Maryland, College Park, MD, USA, 1984, Technical report TR-1395.

[11] ——, "A theory of error-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, August 1990.

[12] J. Offutt, "Automatic test data generation," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1988, Technical report GIT-ICS 88/28.

[13] R. A. DeMillo and J. Offutt, "Constraint-based automatic test data generation," *IEEE Transaction on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[14] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, April 2017.

[15] Google, "Google test automation conference," Online, https://developers.google.com/google-test-automation-conference/, last access: September 2019.

[16] P. Copeland, "Google's innovation factory (keynote address)," in *Proceedings of the Third International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society Press, March 2010.

[17] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[18] D. Ma'ayan, "The quality of JUnit tests: An empirical study report," in *ACM/IEEE 1st International Workshop on Software Qualities and their Dependencies (SQUADE)*, 2018.

[19] M. Fowler, "Eradicating non-determinism in tests," Online, 2011, https://martinfowler.com/articles/nonDeterminism.html, last accessed October 2019.

[20] "Flakiness dashboard HOWTO," Online, http://www.chromium.org/developers/testing/flakiness-dashboard, last access October 2019.

[21] Anonymous, "TotT: Avoiding flakey tests," Online, 2008, https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html, last accessed October 2019.

[22] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, September 2015, pp. 101–110.

[23] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *20th European Conference on Object-Oriented Programming*, vol. 4067, April 2006, pp. 380–403.

[24] Y. Song, S. Thummalapenta, and T. Xie, "Unitplus: Assisting developer testing in eclipse," in *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*. ACM, 2007, pp. 26–30.

[25] M. Staats, G. Gay, and M. P. E. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 870–880.

[26] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: Automated oracle data set selection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 193–203.

[27] D. Schuler and A. Zeller, "Checked coverage: An indicator for oracle quality," *Wiley's Software Testing, Verification, and Reliability*, vol. 23, no. 7, pp. 531–551, 2013.

[28] J. Zhi and V. Garousi, "On adequacy of assertions in automated test suites: An empirical investigation," in *Third IEEE International Workshop on Regression Testing*, March 2013, pp. 382–391.

[29] Q. Xie and A. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transaction on Software Engineering and Methodology*, vol. 16, no. 1, February 2007.

[30] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *33rd International Conference on Software Engineering*. ACM, 2011, pp. 391–400.

[31] K. Koster and D. C. Kao, "State coverage: A structural test adequacy criterion for behavior checking," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, September 2007, pp. 541–544.

[32] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Foundations of Software Engineering*, Hong Kong, China, November 2014, pp. 643–653.