Finding Bugs by Isolating Unit Tests

Kıvanç Muşlu Bilge Soran Jochen Wuttke
Computer Science and Engineering
University of Washington
Seattle, WA, USA
{kivanc,bilge,wuttke}@cs.washington.edu

ABSTRACT

Even in simple programs there are hidden assumptions and dependencies between units that are not immediately visible in each involved unit. These dependencies are generally hard to identify and locate, and can lead to subtle faults that are often missed, even by extensive test suites.

We propose to leverage existing test suites to identify faults due to hidden dependencies and to identify inadequate test suite design. Rather than just executing entire test suites within frameworks such as JUnit, we execute each test in isolation, thus removing masking effects that might be present in the test suites. We hypothesize that this can reveal previously hidden dependencies between program units or tests.

A preliminary study shows that this technique is capable of identifying subtle faults that have lived in a system for 120 revisions, despite failures being reported and despite attempts to fix the fault.

Categories and Subject Descriptors

D.2.5 [**Software**]: Testing and Debugging

General Terms

Reliability

Keywords

data dependency, testing, test isolation, test reuse

1. OVERVIEW

Developers often leave implicit their assumptions about how the code they write is intended to be executed. Implicit expectations about ordering constraints for method invocations or data dependencies, for example, can lead to subtle faults in the code and/or clients using the code. Although rare, such assumptions and faults occur in practice: for example, the initialization order of classes led to a fault in the Tomcat application server[1] that took several months to identify, diagnose and fix[2]. When testers are unaware of such assumptions by developers, the test suites they design are unlikely to systematically address this class of faults.

While it is rarely expressed explicitly, conventional wisdom states and incidental evidence from practice supports a fundamental requirement for test design: tests should be *independent*, that is running them isolated from other tests should not affect their outcome (e. g., [4]). The reason behind this requirement is that if tests are not independent, each test case is no longer a complete specification for what it tests. This premise is so deeply rooted in the minds of software testers that Michal Young, one of the authors of a standard textbook on software testing [6], mentioned in a recent email exchange that *"[in our book] for the most part we just assumed that test cases would be independent in the sense [we describe here], but didn't think to make that explicit."* [9]

Given that, on one side, most existing code contains implicit assumptions and hidden dependencies, and that on the other side test cases are usually assumed to be independent, we hypothesize that test cases for functionality that contains implicit assumptions are likely *not* independent. We further hypothesize that the outcome of test cases that are not independent is likely to change when they are executed in isolation. Thus, a test that changes its outcome when run in isolation likely signals a dependency that is worthy of thorough inspection. To assess the validity of these hypotheses, we propose to isolate test cases from their test suites and to execute them in a fresh environment to avoid "pollution" by other tests. Practically, in an environment like JUnit[3], this amounts to a new execution strategy that resets the environment before the execution of each test method.

We have tested these hypotheses in an experiment on the Apache Commons CLI[4] library, and our results show that isolating test cases can detect faults that are due to hidden dependencies. While this research is preliminary, we believe that it will result in a practical technique to identify dependencies among test cases and expose faults due to hidden assumptions.

In the following sections we briefly discuss the most related work in the area of testing and test adequacy, describe our initial experiments and results, and conclude with an

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

[1] `http://tomcat.apache.org/`
[2] Tomcat Bugzilla bug database: ID 40820, `https://issues.apache.org/bugzilla/show_bug.cgi?id=40820`
[3] `http://www.junit.org/`
[4] `http://commons.apache.org/cli/`

overview of the open questions and the research we are currently undertaking toward answering them.

2. RELATED WORK

Independence of test cases is usually assumed in any testing process. The technique we propose here detects dependence between test cases and, by proxy, identifies hidden dependencies between functional units of the program under test. While we are not aware of any other work that directly attacks this question, there is some research regarding requirements for good test design, including test independence and other measures of testability.

There is a considerable amount of research on *software testability* [1, 8, 3]. The fundamental idea behind *testability* is that to test a piece of code, one must be able to control the inputs to the program, and must be able to observe the outputs. Together, this allows the definition of input/output pairs that serve as test case specifications and oracles. Certain features supported by most imperative and object-oriented programming languages, such as global and static variables, obscure both controllability and observability, since their use is often opaque. While the work on testability addresses requirements that test design should strive to meet, we detect violations of these requirements that lead to test cases that have unexpected dependencies.

Test case prioritization and selection for regression testing are well researched topics that assume independence of test cases [7, 5]. However, to the best of our knowledge, there is no work in this area that checks to what extent this assumption holds. With our technique, we acknowledge that test independence is desirable, but is not always easy to achieve. Thus, it is worthwhile to detect undesired dependencies, and to classify code patterns that lead to such dependencies. Better understanding how well the assumption of test case independence holds in general will also provide a more solid foundation to the practicality of techniques building on this assumption, and highlight the need for further research in case the assumption does not hold.

3. TEST CASE ISOLATION

We propose a new execution strategy for test cases that isolates the effects of each test case and can thus detect changes in test outcomes that are due to hidden dependencies. In principle, this requires to execute test cases in an environment that does not retain generated files, global shared program state and other means of sharing information in programs. The current focus of our work is on unit testing frameworks, in particular JUnit. Here, isolating test cases requires to run individual test methods together with the scaffolding provided, but without having other test methods interfere. Since the available test executors for JUnit do not allow for such separation, we define a simple workaround for that.

Figure 1 shows pseudocode for our test isolation algorithm. First, we run the entire test suite with the standard JUnit runner, and store the results for later comparison. Then, we extract test methods from every test class in the test suite and execute them one by one together with the framework scaffolding methods included in the test classes. To assure isolation, we execute each test method in a fresh copy of the execution environment (directories and files), and start a new JVM to avoid pollution of static and global

```
void runTestSuite(suite) {
  results = runSuiteNormal();
  for (testClass in suite) {
    tests = getTestMethods(testClass);
    for (test in tests) {
      res = runIsolated(test);
      if (res not in results )
        warn(test, res);
    }
  }
}
```

Figure 1: Test isolation algorithm

variables. If the outcome of a test run in isolation differs from the outcome when run together with the test suite, we record a warning and the test output for later analysis.

At present, further analysis has to be done manually. It appears that it is possible to guide automatic analyses based, for example, on the heuristics implemented in Testability Explorer[5], with our results.

4. EXPERIMENT AND PRELIMINARY RESULTS

We tested our hypotheses in an experiment on the Apache Commons CLI library. CLI is a small library, written purely in Java, and has no dependencies to other libraries. We chose this example because having no external dependencies ensures that all effects we observe in our experiment are due to the program itself, rather than external influences. Controlling the variables that may influence the outcome of our analysis more tightly allows us to draw conclusions with higher confidence.

To allow us to determine how long detected faults lived in the systems before they were identified and fixed, we ran our simple prototype on 153 recent and consecutive revisions of CLI from the Apache Subversion repository[6]. Our specific focus was on tests that succeed when executed with the entire test suite, but fail when run in isolation. While this is a limitation of our experiment, we believe that the results are representative, because across all revisions we studied there were only few tests that failed when run with the entire test suite. To be precise, seven of the studied revisions contain at least one test that fails when executed with the entire test suite[7]. We removed these revisions from the initial analysis, so that our analysis would only report those test executions that fail in isolation. The remaining 146 revisions contain on average 214 tests (min = 113, max = 361).

Our setup directly applies the technique described in the previous section to each revision. For each individual test method from the test suite we

1. create a new, clean execution environment by copying all non-test files of the original execution environment,

[5] `http://www.testabilityexplorer.org/`

[6] `http://svn.apache.org/viewvc/commons/proper/cli/trunk/`, revisions 661513–1091538

[7] revisions: 695410, 695672, 742845, 744070, 1091539, 1091550, and 1091575

Revision	Class	Test	Fixed
661513	BugTest	test13666	956302
		test27635	712642
712642	HelpFormatter-Test	testOptionWith-outShortFormat2	956302

Table 1: Suspicious tests found. The table shows the *revision* where we detect the fault, which *class* and *test* method detect it, and in which revision it was *fixed* or moved.

```
1  public final class OptionBuilder {
2    private static String argName;
3
4    private static void reset() {
5      ...
6      argName = "arg";
7      ...
8    }
9
10   public static Option create(String opt){
11     Option option =
12       new Option(opt, description);
13     ...
14     option.setArgName(argName);
15     OptionBuilder.reset();
16     return option;
17   }
18 }
```

Figure 2: Fault-related code from `OptionBuilder.java` (rev. 661513)

2. create a new JUnit TestCase that contains only scaffolding and the selected test method,

3. compile, build, and run this single test through the Maven build system and record the outcome.

We discovered three test cases that fail when run in isolation (summarized in Table 1). The first revision we analyzed (661513) has two failing tests, `test13666` and `test27635` in test class `BugsTest.java`. In revision 712642 (54 revisions later), the method `test27635` is moved to test class `HelpFormatterTest.java` and its name is changed to `testOptionWithoutShortFormat2`. All these tests continue to fail until revision 956302 (120 revisions in total), from which on all tests pass.

A detailed study of the code under test revealed that both tests fail due to the same hidden dependency fault. The fault is located in `OptionBuilder.java` and is due to the improper use of static variables. Figure 2 shows code that illustrates the fault. By default, `argName` is initialized to `null` (line 2), and only set to its intended default value `"arg"` by the `create()` method via calling `reset()` (line 15). Consequently, if clients of CLI do not explicitly initialize the value of `argName`, the first option created will have `null` rather than `"arg"` as its argument name. In CLI, there are two types of options: options with and without argument names. If an option without argument is created first, this fault will not lead to a failure, because the `null` value will be ignored. Consecutive calls to `create()` can rely on `reset()` to establish the desired default value.

Both `test13666` and `test27635` (or `testOptionWithoutShortFormat2`) can reveal this fault, since they create an option with the default argument as the first thing in their execution. However, the test classes `BugTest` and `HelpFormatterTest` both contain other tests that create options *before* `test13666` and `test27635` respectively. Thus, when the tests in these classes are executed in order, the tests executed before `test13666` and `test27635` call `create()` at least once, which sets the default `argName` value, thus masking the fault.

This fault is reported in the bug database several times[8], starting on March 13, 2004 (CLI-26). The report is marked as resolved *three years* later on March 15, 2007, but is then reopened as CLI-186 on July 31, 2009. On this report, one of the developers commented:

> "I reproduced the issue, it requires a dedicated test case since it is tied to the initialization of a static field in OptionBuilder"

Despite the realization that a dedicated test is required, no such test was ever created. About one month later, the bug is duplicated as CLI-187, and the actual fix happens one year later on June 19, 2010, after a total "awareness" of the fault of four years. The fix consists of adding the following code to `OptionBuilder.java`:

```
static {
//ensure consistency of initial values
   reset();
}
```

Since test isolation can identify the dependency fault in the first revision with two different test cases, we strongly believe that the developers could have fixed this long lasting fault much earlier, if they had used our method. After identifying the failing tests, following the dynamic execution of these tests and fixing the fault should be straightforward. Moreover, since our technique is relatively lightweight, it can be integrated into the normal development process and can thus aid developers while they write code, rather than in a separate test and analysis phase.

5. OPEN QUESTIONS AND CURRENT RESEARCH EFFORTS

The results presented in the previous section show that our technique can detect faults due to hidden dependencies. While the cases we report here concern incorrect and inefficient use of static variables, we believe that our technique can also detect similar problems that are due to other forms of shared state, including shared global variables and inappropriate reuse of injected dependencies [2].

A dependency violation detected by our technique may represent a weakness of the test suite rather than a fault in the program. If, for example, the test suite reuses data objects across several tests, this may trigger a dependency warning, but does not imply that the program is faulty. We are considering a wider range of experiments intended to

[8] `https://issues.apache.org/jira/browse/CLI-26`
`https://issues.apache.org/jira/browse/CLI-186`
`https://issues.apache.org/jira/browse/CLI-187`

examine different kinds of dependency violations and to determine which are due to inadequate test suites rather than faults in the program.

We also believe that our technique is capable of finding other types of dependency violations, in particular those due to interactions with the file system or databases. However, there are many legitimate reasons why tests for databases and file handling may not be independent. Therefore, it remains an open question to what extent our technique can be applied in such domains, and to what extent we need to extend its heuristics to reduce the number of false positives reported.

An important issue to determine is the false positive rate. We have not yet conducted a formal study of false positives, since our prototype implementation is not yet complete and has some weaknesses that further work will remove. It appears, however, that the absolute number of false positives will be low, since dependency violations are relatively rare.

The fault we discussed in the previous section is due to inappropriate use of static variables in Java. From the code structure it is clear that the involved variable should be a field of an instantiated class, and the problem would not even occur. It is an interesting question to explore which other patterns of faulty code we can identify by using our technique, and if we can catalog them in enough detail to inform static analysis techniques.

Another open question that we intend to address is the value of our technique in a development process. Our experiment shows that the technique could find the dependency fault when it was introduced, however it took the developers a long time of trial and error before they could fix it. We do believe that the warnings produced by our technique alone would have helped developers to fix the detected problem much faster. Furthermore, we envision several extensions to our technique that will enhance its usefulness. Dynamic techniques, such as coverage computation, can establish a starting set of affected methods, and data-flow analysis and program slicing of these can help to reduce the amount of code developers would need to analyze to determine the location of a fault.

At present, our prototype implementation incurs significant overhead. However, a large part of it is caused by calling Maven to clean and build the system for every test method. A more advanced implementation will instead be a JUnit runner, and thus avoid the overhead due to Maven. The creation of a new JVM for each test method also creates considerable overhead, even though we believe that for most test suites performance will be acceptable. In cases where the performance penalty is too high, it appears that running our technique as part of a nightly build cycle rather than during development still has considerable benefits.

We are currently working on improving our results along two major lines of research. On the one side, we are applying our technique to a wider range of programs. Gathering more cases where our technique produces warnings will lead to a better understanding of when and how to apply our technique to obtain the best value. On the other side, we are working on identifying patterns in the code that has been flagged as faulty. If we can find typical patterns of what causes warnings in our technique, we can then build heuristics that will further improve the precision of the warnings produced and will guide developers more quickly to find the faulty code.

6. CONCLUSION

In this paper we present a new technique to detect hidden dependencies in programs and test suites. The experimental results show that the technique has the potential to detect faults due to hidden dependencies early, and can guide developers more quickly to the faulty code.

We are currently experimenting with more programs to establish cases where the technique applies well, how we can improve the quality of the reported results, and how we can extend the technique with heuristics to aid fault localization in addition to detecting faults.

7. ACKNOWLEDGMENTS

This work was partly funded by Swiss National Science Foundation grant no. PBTIP1-134569, and NSF grants CCF-1016490AM01, CNS-0855252, and CCF-0963757.

8. REFERENCES

[1] Robert V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

[2] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, Jan 2004. `http://martinfowler.com/articles/injection.html\#ServiceLocatorVsDependencyInjection`.

[3] What is testable? `https://code.google.com/p/testability-explorer/wiki/HowItWorks`, accessed June 09, 2011.

[4] `http://junit.sourceforge.net/doc/testinfected/testing.htm`, accessed June 09, 2011.

[5] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 119–129. ACM, 2002.

[6] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, 2007.

[7] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[8] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, 1995.

[9] Michal Young, June 9, 2011. Personal communication.