BENOIT BAUDRY AND MARTIN MONPERRUS

Testing beyond coverage

Pseudo-tested methods can be a reliability risk. Here, the authors explain how they developed a methodology and tool to uncover them in Java applications.

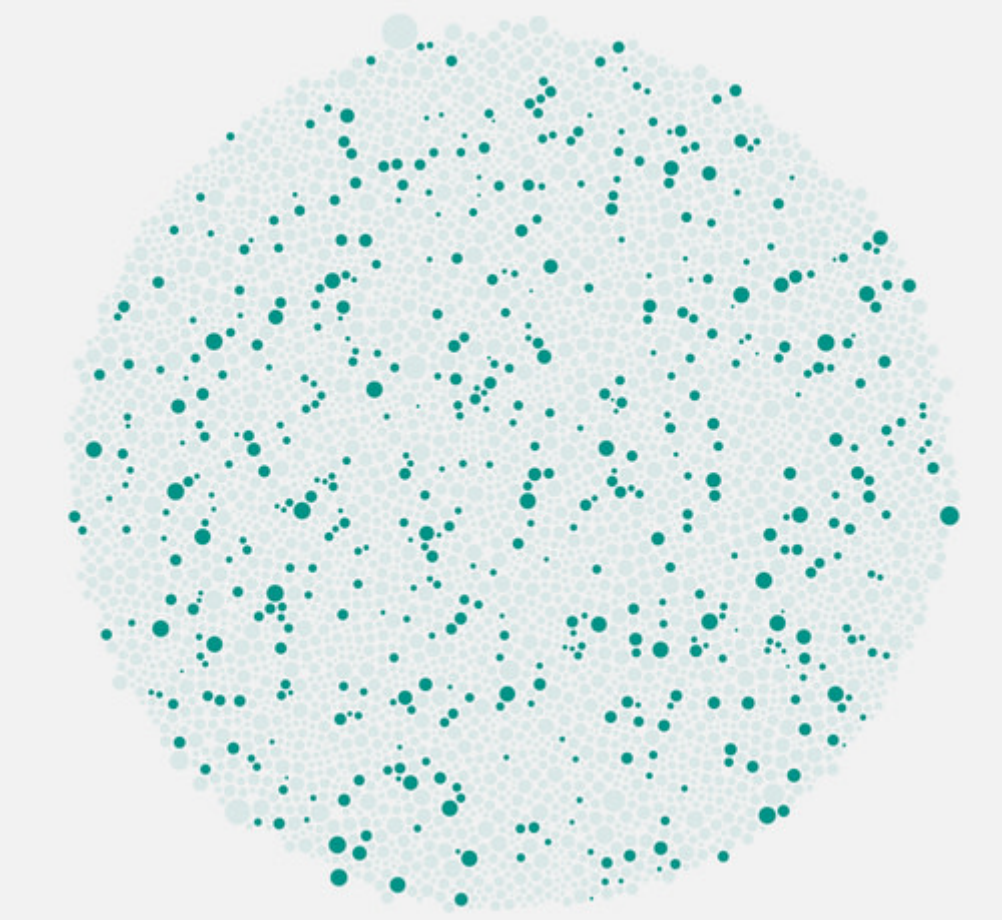
PART OF

ISSUE 16	Reliability
FEBRUARY 2021	

Development teams typically rely on code coverage to estimate how much they can trust their application's test suite. Covered code is considered more trustworthy than uncovered code, so the higher the code coverage, the stronger the trust.

But this sense of trust can be misleading. While tests can't detect bugs in unexecuted code, they can also miss bugs in code they do cover. For example, the following image highlights 925 test cases in the test suite of Apache's Commons Collections, which all cover part of a method called `ensureCapacity`. Yet not a single test fails when the `ensureCapacity` body is stripped out. (We'll return to the Commons Collections example later.) These frustrating scenarios can occur when test case assertions are too weak to fully assess the program's reliability. This represents a critical risk to the software: A developer could potentially introduce a change to `ensureCapacity` that breaks the application in production because the test suite wasn't able to catch it.

In this article, we'll evaluate the trust we as software developers place in automatic unit testing, and present a new method and tool to help make software test suites more trustworthy—which can make the software itself more reliable.



These 925 test cases cover the Apache Commons Collections method `ensureCapacity`, which ensures the map is large enough to contain the required number of elements.

A tale from the pseudo-tested trenches

Pseudo-tested methods can lure development teams into a false sense of security.

Pseudo-tested methods are methods that are covered by the test suite but whose behavior is not properly assessed by any test case. They can lure development teams into a false sense of security, and they represent a serious threat to constructing reliable applications. We've found them in a variety of popular software projects,[1] including Apache Commons Collections (in our estimation, the coolest collection library out there), PDFBox (the Swiss Army knife of PDF generating and editing tools), and the SDK for Amazon Web Services.

The COVID-19 pandemic dramatically altered the way many people around the world work and attend school. The demand for reliable videoconferencing systems has soared as a result. Along with Zoom and Microsoft Teams, the videoconferencing solution Jitsi Meet gained a significant number of users, who appreciated the system's end-to-end encryption, open-source codebase, and concise security and privacy policies, among other key features. With so many people depending on Jitsi Meet to work and learn, a strong test suite is essential to keep its application reliable and secure.

After reporting these **pseudo-tested methods** to Jitsi on GitHub, Jitsi's developers refactored the code to remove the pseudo-tested methods. See pull requests `#638` to `#644`.

Yet in October 2020, we analyzed Jitsi Meet's test suite and identified several **pseudo-tested methods**. For instance, we found one called `getEnforcedVideobridge` in a module responsible for managing media sessions between participants. This method's code is covered by 11 test cases spread over five test classes, yet no test case calls the method directly. This suggests the test suite is not accurately assessing the method's behavior.

```
1   public Jid getEnforcedVideobridge(){
```

```
    try {
      String enforcedBridge = properties.get(PNAME_ENFORCED_BRID
      if (isBlank(enforcedBridge)) {
        return null;
      }
      return JidCreate.from(enforcedBridge);
    }
    catch (XmppStringprepException e) {
      logger.error("Invalid JID for enforced videobridge", e);
      return null;
    }
  }
```

Code for the pseudo-tested method `getEnforcedVideobridge` in Jitsi Meet

Analyzing `getEnforcedVideobridge` further, we found that when its body is entirely removed (replaced by `return null`), none of the 11 test cases failed, making it a pseudo-tested method. If a commit breaks `getEnforcedVideobridge`'s behavior, no test case would capture the problem before integration testing—or, worse, production.

We were able to identify these problems in Jitsi Meet because its code is open source, but we suspect similar problems may exist in the test suites of proprietary systems such as Zoom. (We'd happily invite the test engineers who prove us wrong to visit us in Stockholm!)

Introducing the Descartes method

Mutation testing is a technique that measures an application test suite's quality. It generates different buggy variants of the application and then, by running the test suite on each variant, assesses the test suite's ability to detect them. While the test coverage metric can determine which parts of the code the test suite is able to reach, mutation testing can also determine which bugs the assertions in the test suite catch. The Descartes method can be considered an extreme form of mutation testing, since each buggy variant consists of

removing a method's whole body.

We maintain a tool for finding pseudo-tested methods in Java called Descartes. It's integrated into the Maven toolchain, which makes it easy to use for most Java projects. It's open source, and we provide support for it on GitHub.

Descartes is built on a methodology we've developed to identify pseudo-tested code. The idea is simple: Remove a method's body with automated code transformation and run the tests again. Think of it as a type of **mutation testing** with an extreme mutation operator (remove the full body of a method).

Here's how it works:

1. First, Descartes finds all methods in a program that are covered by the test suite.
2. Next, it eliminates uninteresting methods such as trivial setters and getters.
3. Then, one method body at a time, it empties each method and runs the test suite again. If a method returns no value, its body is basically stripped out. If a method is expected to return a value, Descartes generates a few variants that return predefined values, as shown in the table below.

METHOD TYPE	RETURNED VALUES
boolean	`true`, `false`
byte, short, long, int	`0`, `1`
float, double	`0.0`, `0.1`

METHOD TYPE	RETURNED VALUES
char	`''`, `'A'`
string	`""`, `"A"`
`T[]`	`new T[]`
reference types	`null`
void	-

When the test suite runs on a version of the program that includes an empty, covered method (the transformed one), there are two possible outcomes:

1. If no test case fails, then the method is pseudo-tested. This represents a reliability risk for the software system in question, and Descartes reports it to the developer.
2. If at least one test case fails, that's good news. It means the method is specified by at least one assertion and is protected against the introduction of unintentional errors. No action needs to be taken.

Descartes' final output is a list of improperly tested methods. These are concrete targets around which developers can concentrate their next testing efforts, or good leverage for convincing managers to allocate some budget for testing and reliability.

When pseudo-testing meets observability

Our group of software engineers and PhD students has used Descartes on open-source and commercial projects, including the Amazon Web Services SDK and the enterprise wiki system XWiki. We've found that

remediating pseudo-tested methods can sometimes be an observability challenge for engineers. Compensating for the weaknesses of test oracles is not straightforward: There are parts of software systems that simply cannot be observed, due to visibility and encapsulation mechanisms as well as nonfunctional properties.

VISIBILITY AND ENCAPSULATION

Some pseudo-tested methods, for instance, are private. In these cases, although the developer may be able to write an assertion to specify the behavior, they have to substitute a direct call (which is forbidden by the runtime) with an indirect call via the public API. This is not always easy, though it's possible in programming languages with little visibility enforcement, or in popular dynamic languages such as JavaScript.

METHODS THAT IMPLEMENT NONFUNCTIONAL REQUIREMENTS

We've also noticed that many pseudo-tested methods are related to nonfunctional requirements such as optimization (e.g., handling a cache) and security (e.g., checking input data's validity). Discussing these cases with lead developers, we've learned that these methods, which are not central to the application's functional features, are hard to test.

Consider Apache's Commons Collections, one of the leading Java libraries for handling collections. It includes an abstract class for HashMap, with code to ensure the map is large enough to contain the required number of elements. The method is aptly named `ensureCapacity`. Commons Collections is tested in accordance with the Apache Software Foundation's notably high standards: When running the whole test suite on the library, `ensureCapacity` is invoked over 11,000 times and covered by more than 900 test cases. Running Descartes on it, however, we found that the body of `ensureCapacity` can

be completely deleted without breaking any test case. We found a total of 40 pseudo-tested methods out of 2,729, or approximately 3 percent. What a blast!

We reached out to Commons Collections' developers on GitHub to let them know about this weakly tested method. They acknowledged the issue and added a stronger assertion to the test suite, which makes `ensureCapacity` better tested.

After analyzing it further, we discovered that `ensureCapacity` contains purely nonfunctional code, which would be catastrophic to break. With only coverage information to go on, **Commons Collections' developers** weren't able to detect this problem. This suggests projects with high reliability requirements, which include ensuring the quality of nonfunctional properties, should implement tooling to detect pseudo-tested methods.

```
1   protected void ensureCapacity(final int newCapacity) {
2       final int oldCapacity = data.length;
3       if (newCapacity <= oldCapacity) {
4             return;
5       }
6       if (size == 0) {
7             threshold = calculateThreshold(newCapacity, loadFact
8             data = new HashEntry[newCapacity];
9       }
10      else {
11            //Rehash code
12        ...
13            threshold = calculateThreshold(newCapacity, loadFact
14            data = newEntries;
15      }
16  }
```

Code for the pseudo-tested method `AbstractHashedMap#ensureCapacity` in

Take Descartes for a test drive

We believe the Descartes methodology is unique in the way it emphasizes methods as a unit of feedback for developers. The granularity of a method makes it well-suited to providing actionable feedback: Descartes essentially tells the developer, "Look! There's a testing problem with this method," and they can act accordingly. At the same time, it's not as overwhelming as reporting problems at the line level. Overall, developers and testers can easily understand the targeted feedback Descartes provides, and can take action to consolidate assertions or add tests to cover edge cases.

We hope this discussion of pseudo-tested methods will motivate you to investigate how your own testing processes can be improved and made more trustworthy. If you're working on a Java project, perhaps you'll find Descartes a helpful place to start.

[1] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, "A Comprehensive Study of Pseudo-tested Methods," *Empirical Software Engineering* (2018): 1–33.

ABOUT THE AUTHOR

Benoit Baudry and **Martin Monperrus** are professors of software engineering at KTH Royal Institute of Technology in Stockholm. They work with their group of graduate students to develop novel tools to

build reliable software.

TOPICS

Learn Something New

 @incrementmag

---

 incrementmag

 RSS Feed

ABOUT

Increment is a print and digital magazine about how teams build and operate software systems at scale. Learn more

WORK WITH US

Interested in joining the team at Stripe? View job openings

 Published by Stripe Privacy policy