

Testing - Oracles and Adequacy

Advanced Software Engineering
Spring 2023

© 2023 Jonathan Bell, [CC BY-SA](#)

A When survey is active, respond at **pollev.com/jbell**

2-7 Testing Oracles Overview

0 done

⟳ **0 underway**

Powered by  **Poll Everywhere**

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Have you written tests for software besides a course project before?

Yes

No

Total Results: 0

Which (if any) of these testing technologies/tools have you used?

xUnit (JUnit, PyUnit, etc)

Behavioral testing tools (Cucumber, jest, mocha, etc)

Selenium

QuickCheck

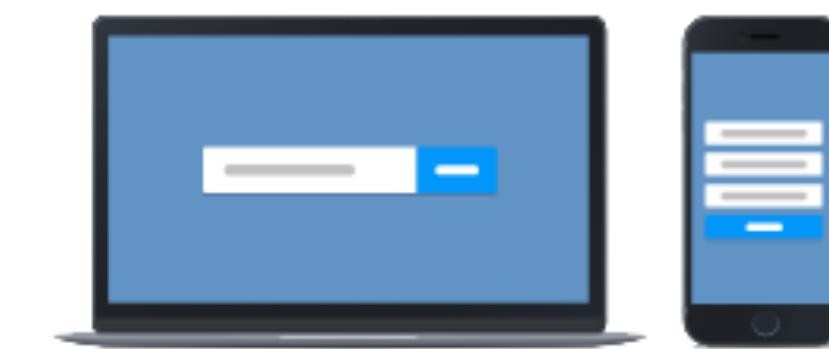
Fuzzers

Manual testing

Total Results: 0

Do you have a software project that you are working on and interested in creating automated tests for? If so, are there any particular aspects to it that are making it hard to test?

Join by Web



- 1 Go to [PollEv.com](#)
 - 2 Enter **JBELL**
 - 3 Respond to activity
- i** Instructions not active. [Log in](#) to activate

Total Results: 0

Powered by  **Poll Everywhere**

Start the presentation to see live content. For screen share software, share the entire screen. Get help at [pollev.com/app](#)

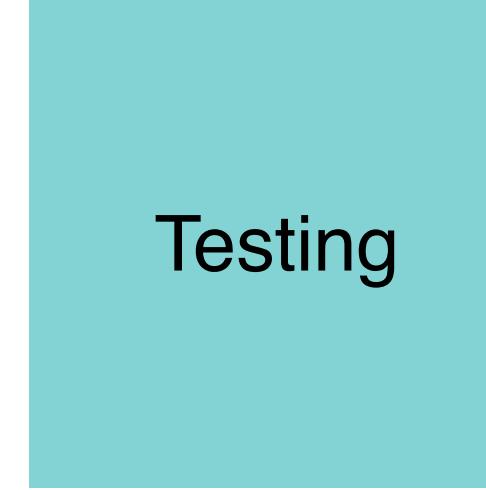
Big question: Quality Assurance

“Proving” that “the program” is “correct”



Verification

Prove properties about a program
High assurance



Testing

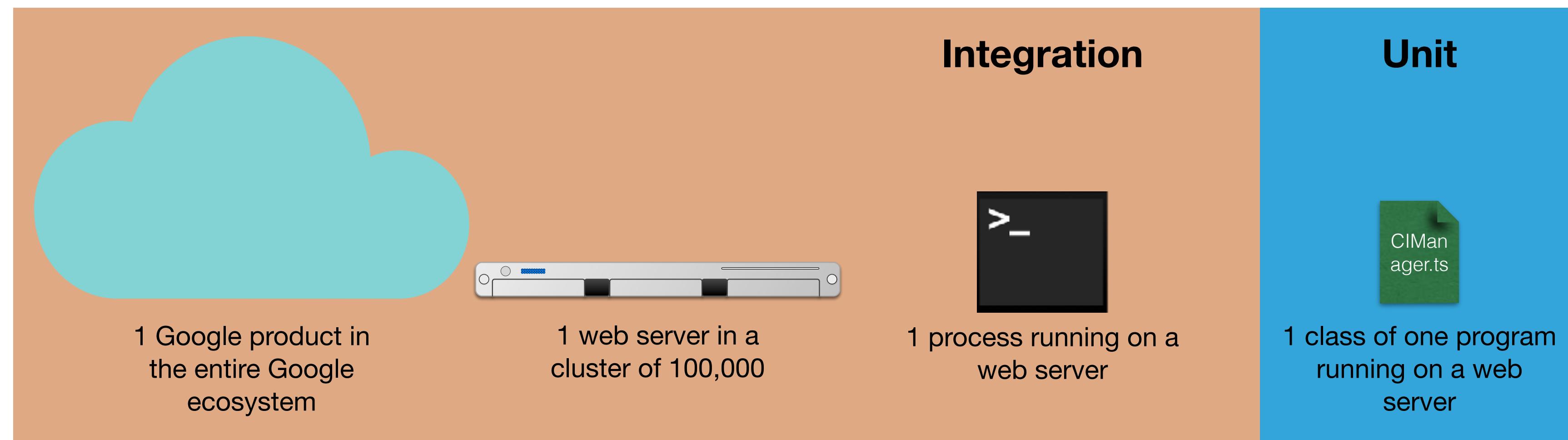
Check behaviors of a program
Assurance?

What is a test?

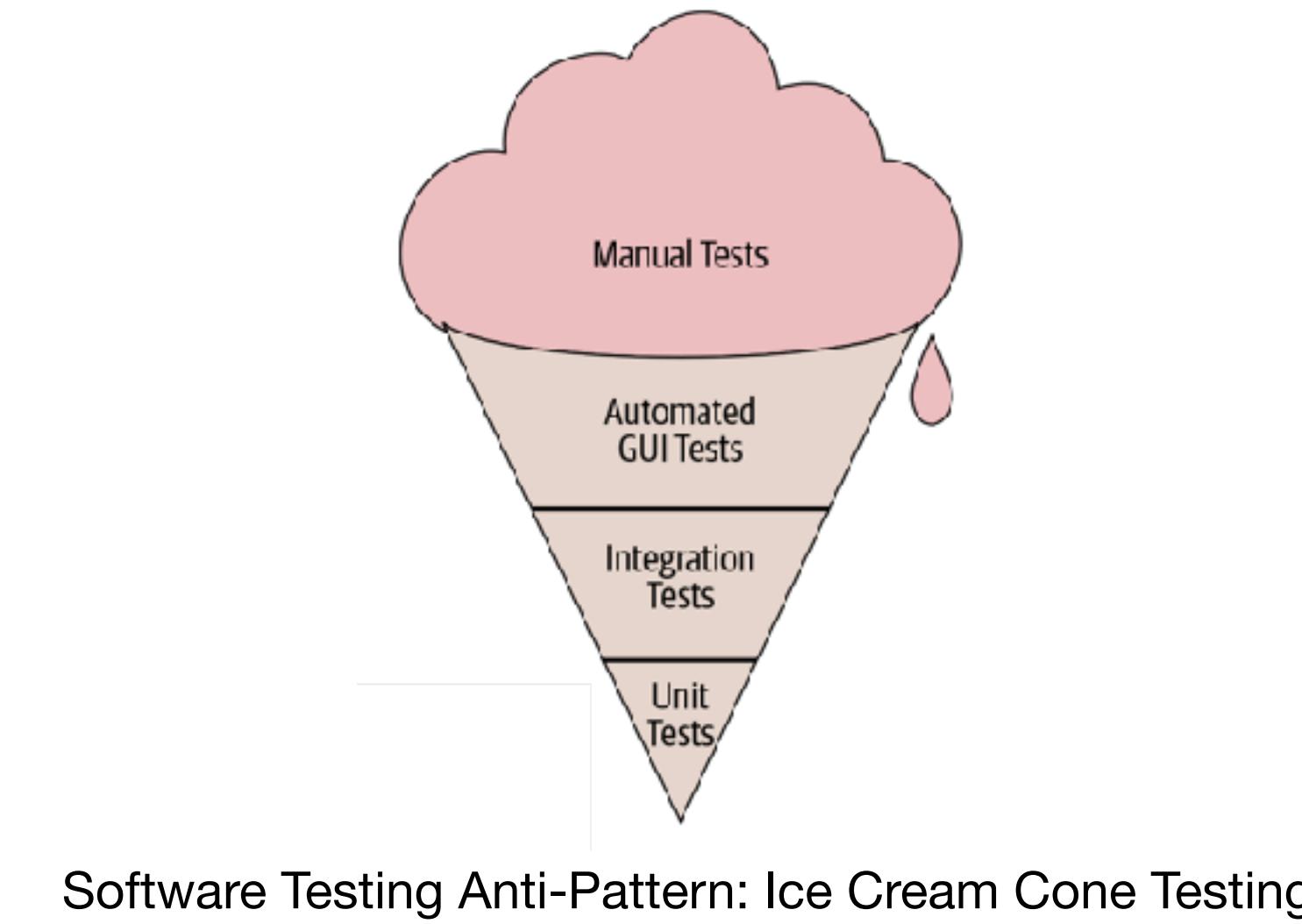
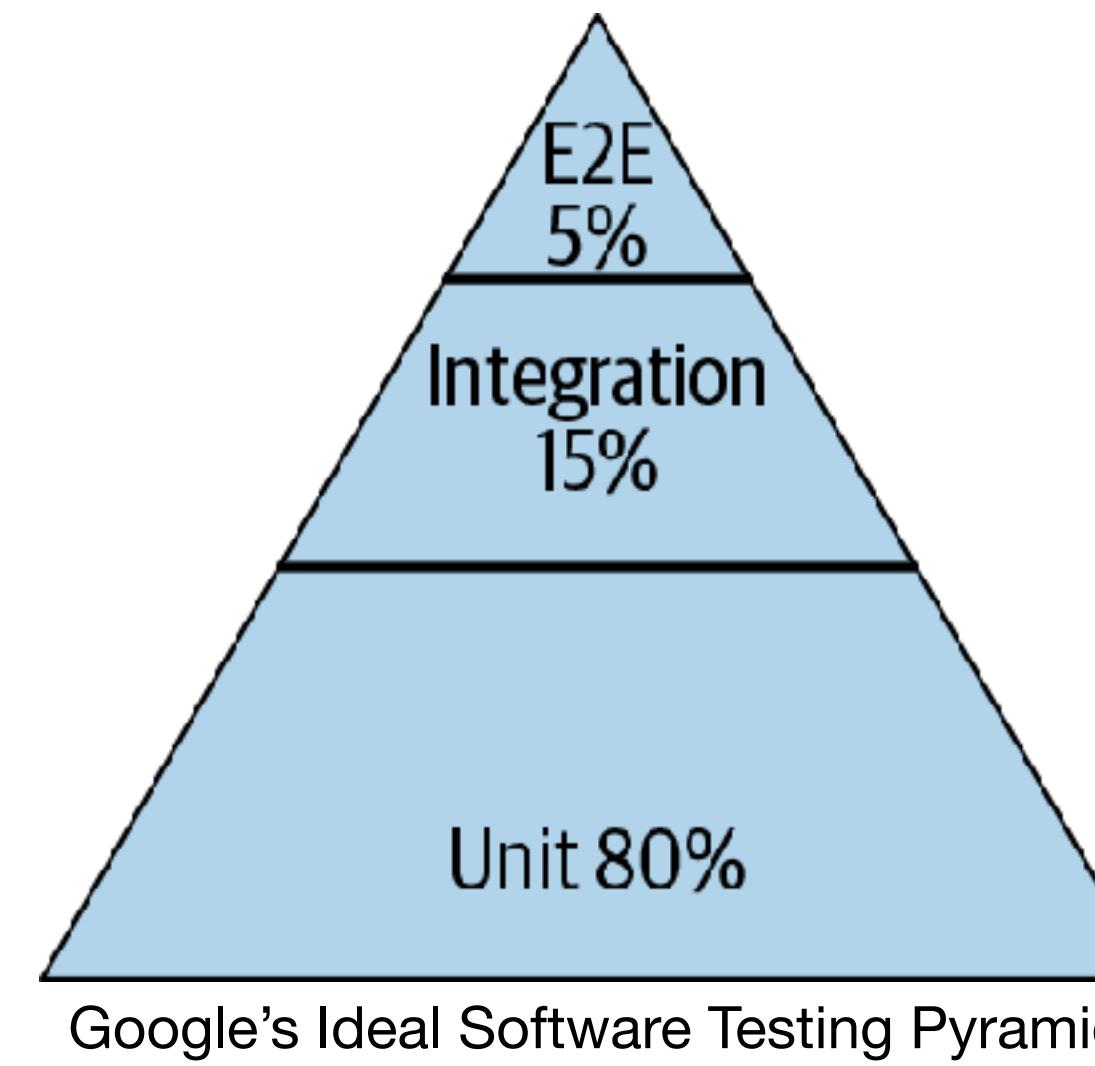
- “System under test”
- Inputs
- Oracles

Each test might target a different scope

- Unit tests: SUT = a single method/class/object
- Integration tests: SUT = combinations of units, a subsystem
- System tests: SUT = whole system being developed



Good test suites have a strong foundation of unit tests



Figures: “Software Engineering at Google: Lessons Learned from Programming Over Time,” Wright, Winters and Mansreck, 2020 (O’Reilly)

What makes a good test suite?

Brainstorming

- (The pyramid, not the ice cream cone)
- If a test fails, easy to debug
- Fast/cheap to run - including in a regression context where we know what code changed
- Minimum false positives (test fails when there is not a fault)
- Coverage of common (all?) behaviors/code
- Scalable and maintainable over time - readable
- Tests should be isolated (no contamination of external sources)
- Test should be repeatable/consistent/deterministic wrt SUT
- Test suites are optimized to detect crucial issues sooner, provide actionable output
- Minimum false negatives (it finds all of the bugs)
- Designed and organized in a way where a newcomer can understand; traceable from tests to their purpose
- Mimic real-world inputs/outputs/edge cases

What makes a good test?

- Desirable properties of test suites:
 - Find bugs
 - Run automatically
 - Are relatively cheap to run
- Desirable properties of individual tests:
 - Understandable and debuggable
 - No false alarms (not “flaky”)

What is “correct” behavior?

Test oracles

- Example-based: “For a given input, some assertions should be true”
- Properties: “All inputs in some class should satisfy some property”
 - “It doesn’t crash”
 - “Changing the input in some way should maintain the same output”
- Regression: “It provides the same output as it used to”
- Differential: “Two systems implementing the same spec should provide the same output”
- Human oracle: “For a given user, they should be satisfied”

Example-Based Testing

AKA “traditional” testing?

- Interpret the specifications for the program, encode those specifications into input/output/behavior examples
- Key problems: How to create the examples? How to encode them in an automated test?

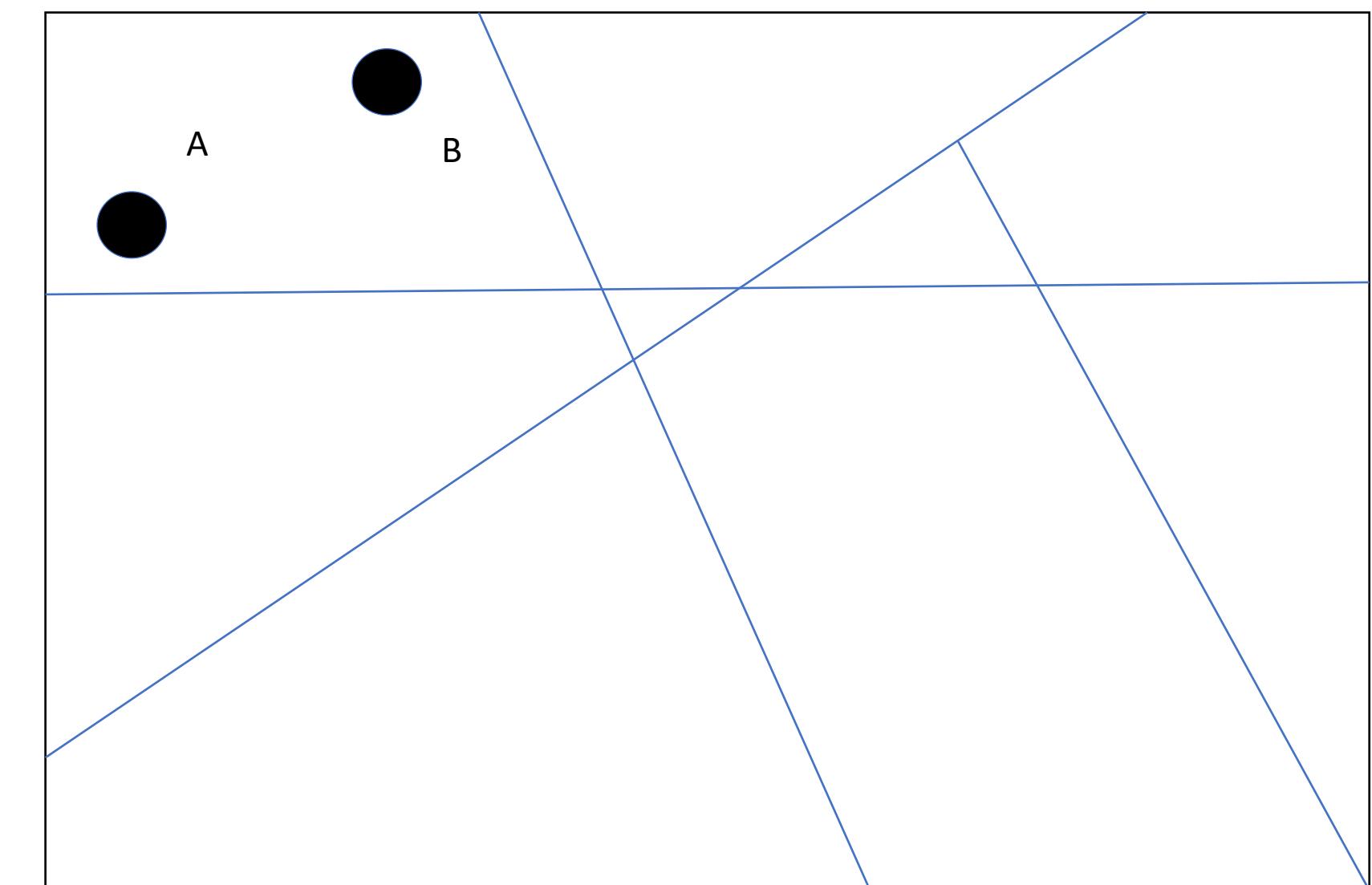
```
it('Removes the player from the list of occupants and emits an interactableUpdate event', () => {
  // Add another player so that we are not also testing what happens when the last player leaves
  const extraPlayer = new Player(nanoid(), mock<TownEmitter>());
  testArea.add(extraPlayer);
  testArea.remove(newPlayer);

  expect(testArea.occupantsByID).toEqual([extraPlayer.id]);
  const lastEmittedUpdate = getLastEmittedEvent(townEmitter, 'interactableUpdate');
  expect(lastEmittedUpdate).toEqual({ topic, id, occupantsByID: [extraPlayer.id] });
});
```

What is a “good” test suite?

Interpretation: Coverage of Input Space

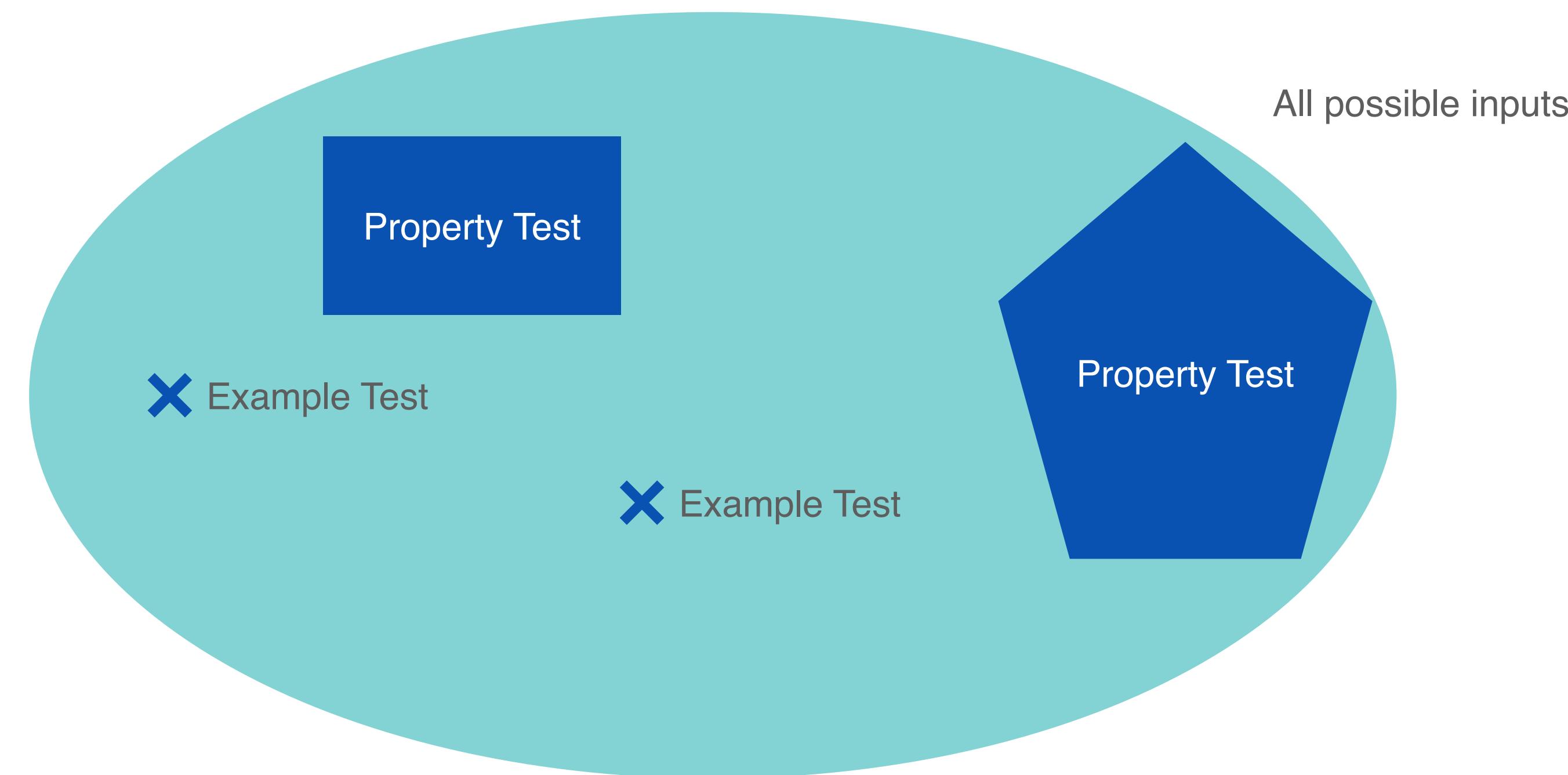
- (Manually) enumerate possible “equivalence classes” of inputs
- Ensure that each equivalence class is covered by a test
- Pay extra attention to boundary cases



If the program works for input A, it will probably work for input B

Property Testing

- Key idea: Instead of enumerating examples of inputs in an equivalence class, specify the equivalence class, and allow the testing framework to generate inputs



Property Testing: “Hello World” Example

- for all (a, b, c) strings
 - the concatenation of a, b and c always contains b
 - What other properties can we define for concatenation, and do you think that they will be likely to reveal bugs?

Property Testing: No Crash

“Don't do bad things”

- Not a signal that good things eventually happen, but still shockingly effective at finding defects

```
public void testBZip2CompressorStream(byte @Size(min=100, max=100)[ ] bytes){  
    OutputStream o = new ByteArrayOutputStream();  
    try {  
        BZip2CompressorOutputStream bo = new BZip2CompressorOutputStream(o);  
        bo.write(bytes);  
        bo.finish();  
    } catch (IOException e){  
        Assume.assumeNoException(e);  
    }  
}
```

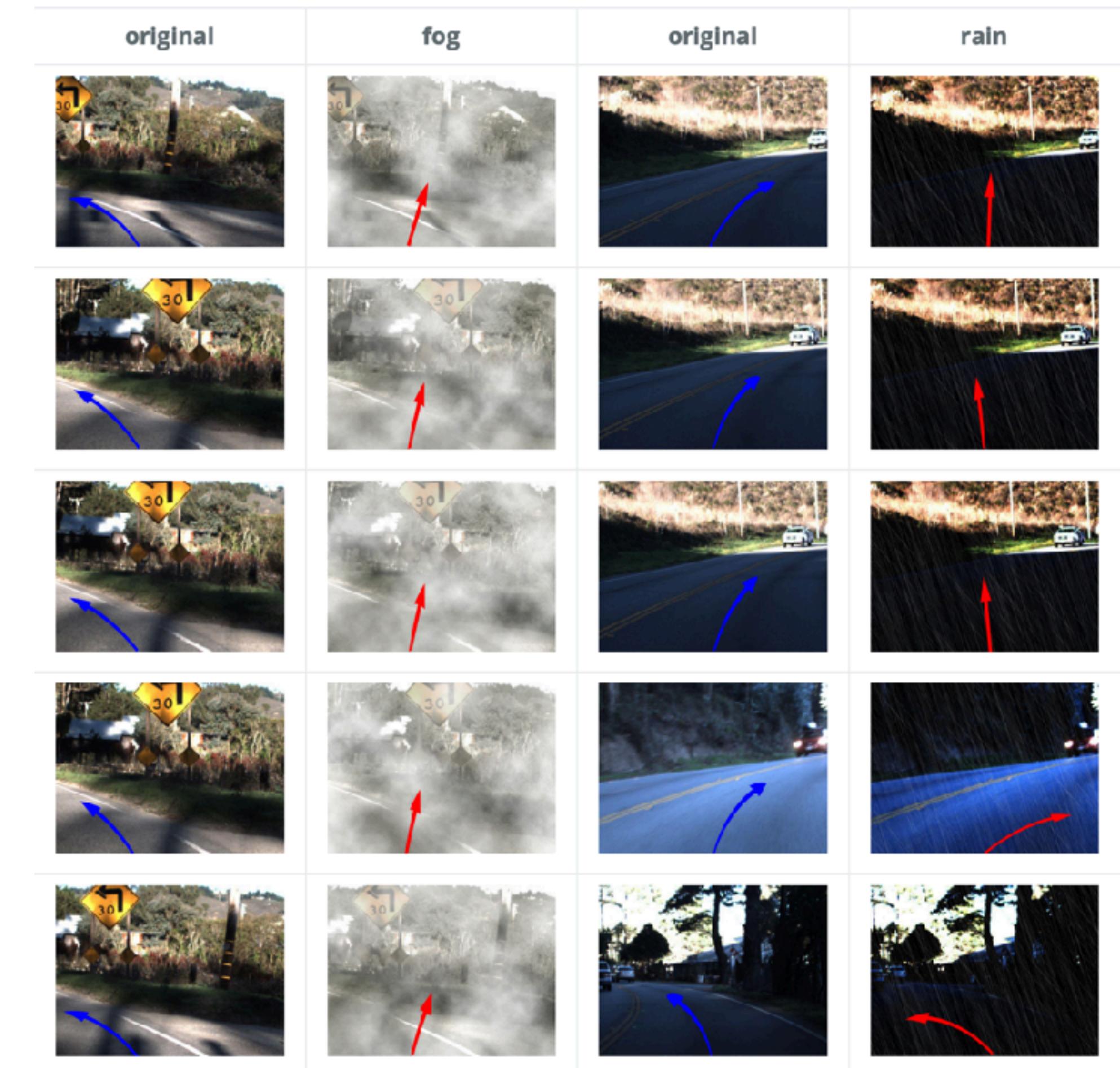
Property Testing: “Round Trip Conversion” Example

- Consider writing this kind of property test for any code that transforms an input in a way that is also reversible

```
public void roundTripBzip2Test(byte @Size(min = 100, max = 100) [] bytes) throws IOException {  
    ByteArrayOutputStream o = new ByteArrayOutputStream();  
    BZip2CompressorOutputStream bo = new BZip2CompressorOutputStream(o);  
    bo.write(bytes);  
    bo.finish();  
    byte[] compressedBytes = o.toByteArray();  
    byte[] decompressedBytes = new byte[1024];  
    new BZip2CompressorInputStream(new ByteArrayInputStream(compressedBytes))  
        .read(decompressedBytes, 0, decompressedBytes.length);  
    Assert.assertArrayEquals(bytes, decompressedBytes);  
}
```

Property Testing: Self Driving Cars

- Problem: ML application learns from traffic images, determines how to steer car safely
- How do we exhaustively generate inputs?
- Approach: apply image transformations to known cases



Property Testing: Graphics Drivers

- Problem: Graphics shader compilers are hard to implement correctly. What is “correct?”
- One idea: Given a shader that creates an image, transform the shader in a way that shouldn’t cause a change in the generated image

Automated Testing of Graphics Shader Compilers

ALASTAIR F. DONALDSON, Imperial College London, UK

HUGUES EVRARD, Imperial College London, UK

ANDREI LASCU, Imperial College London, UK

PAUL THOMSON, Imperial College London, UK

We present an automated technique for finding defects in compilers for graphics shading languages. A key challenge in compiler testing is the lack of an oracle that classifies an output as correct or incorrect; this is particularly pertinent in graphics shader compilers where the output is a rendered image that is typically underspecified. Our method builds on recent successful techniques for compiler validation based on *metamorphic testing*, and leverages existing high-value graphics shaders to create sets of transformed shaders that should be semantically equivalent. Rendering mismatches are then indicative of shader compilation bugs. Deviant shaders are automatically minimized to identify, in each case, a minimal change to an original high-value shader that induces a shader compiler bug. We have implemented the approach as a tool, GLFuzz, targeting the OpenGL shading language, GLSL. Our experiments over a set of 17 GPU and driver configurations, spanning the main 7 GPU designers, have led to us finding and reporting more than 60 distinct bugs, covering all tested configurations. As well as defective rendering, these issues identify security-critical vulnerabilities that affect WebGL, including a significant remote information leak security bug where a malicious web page can capture the contents of other browser tabs, and a bug whereby visiting a malicious web page can lead to a “blue screen of death” under Windows 10. Our findings show that shader compiler defects are prevalent, and that metamorphic testing provides an effective means for detecting them automatically.

CCS Concepts: • Software and its engineering → Software testing and debugging; • Computing methodologies → Rasterization;

Additional Key Words and Phrases: GPUs, OpenGL, GLSL, testing, shaders, compilers

ACM Reference Format:

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (October 2017), 29 pages.

<https://doi.org/10.1145/3133917>

1 INTRODUCTION

Real-time 2D and 3D graphics, powered by technologies such as OpenGL [Kessenich et al. 2016c], Direct3D [Microsoft 2017a] and Vulkan [Khronos Group 2016], are at the heart of application domains including gaming and virtual reality, and are employed in rendering the graphical interfaces of operating systems, interactive web pages, and safety-related products such as automated driver

This work was supported by EPSRC Early Career Fellowship (EP/N026314/1), an EPSRC-funded studentship via the Centre for Doctoral Training in High Performance Embedded and Distributed Systems (EP/L016796/1), and Imperial College’s EPSRC Impact Acceleration Account.

Authors’ addresses: A. F. Donaldson, A. Lascu, H. Evrard, P. Thomson, Department of Computing, Imperial College London, London, SW7 2AZ, UK.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2017 Copyright held by the owner/author(s).

2475-1421/2017/10-ART93

<https://doi.org/10.1145/3133917>

Property Testing: Graphics Drivers

Config.	Original image	Injection	Variant image
1 (AMD)		<pre>if(injSwitch.x > injSwitch.y) { if(injSwitch.x > injSwitch.y) { return; } int f = 1; } [...] if(injSwitch.x > injSwitch.y) { return; }</pre>	
10 (ARM)		<pre>if(injSwitch.x > injSwitch.y) { for(int i = 0; i < 1; i++) { k = 0.0; } }</pre>	
14 (ARM)		<pre>uniform float GLF_13time; float GLF_13map() { [...] } [...] float GLF_13t = 1.0; for(int GLF_13i = 0; GLF_13i < 1; GLF_13i++) { if(GLF_13t > 1.0) { continue; } GLF_13t += GLF_13map(); }</pre>	
9 (Apple)		<pre>for(int c = 0; c < 1; c++) { if(j == 0) { return vec4(0.8, 0.5, 0.5, 1.0); } } return vec4(0.8 * injSwitch.y, 0.7, 0.4, 1.0);</pre>	
15 (ImgTech)		<pre>if(injSwitch.x > injSwitch.y) { for(int i = 0; i < 10; i++) { continue; } } [...] if(injSwitch.x > injSwitch.y) { if((p.z > 60.)) { break; } }</pre>	
3 (Intel)		<pre>[...] vec2 uvs = [...]; vec4 uvs_vec = vec4(0.0, uvs, 0.0); /* Replace uvs with uvs_vec.yz after */</pre>	
16 (NVIDIA)		<pre>vec3 hsbToRGB(float h, float s, float b) { return b * ((false ? (-s) : 1.0) - s) + (b - (false ? (-s) : b * (1.0 - s))) * clamp(abs((false ? (-s) : 6.0)) * [...]); }</pre>	
17 (Qualcomm)		<pre>/* Multiple instances of the following, where v is a literal value */ if(injSwitch.x > injSwitch.y) return v;</pre>	

Table 2. Wrong image examples from different configurations. The image produced by the original shader is on the left, the image produced by the variant shader is on the right, and the code injection (highlighted in yellow) that induces the bug is shown in the middle. Recall that injSwitch is set to (0.0, 1.0) at runtime.

Automated Testing of Graphics Shader Compilers

ALASTAIR F. DONALDSON, Imperial College London, UK

HUGUES EVRARD, Imperial College London, UK

ANDREI LASCU, Imperial College London, UK

PAUL THOMSON, Imperial College London, UK

We present an automated technique for finding defects in compilers for graphics shading languages. A key challenge in compiler testing is the lack of an oracle that classifies an output as correct or incorrect; this is particularly pertinent in graphics shader compilers where the output is a rendered image that is typically underspecified. Our method builds on recent successful techniques for compiler validation based on *metamorphic testing*, and leverages existing high-value graphics shaders to create sets of transformed shaders that should be semantically equivalent. Rendering mismatches are then indicative of shader compilation bugs. Deviant shaders are automatically minimized to identify, in each case, a minimal change to an original high-value shader that induces a shader compiler bug. We have implemented the approach as a tool, GLFuzz, targeting the OpenGL shading language, GLSL. Our experiments over a set of 17 GPU and driver configurations, spanning the main 7 GPU designers, have led to us finding and reporting more than 60 distinct bugs, covering all tested configurations. As well as defective rendering, these issues identify security-critical vulnerabilities that affect WebGL, including a significant remote information leak security bug where a malicious web page can capture the contents of other browser tabs, and a bug whereby visiting a malicious web page can lead to a “blue screen of death” under Windows 10. Our findings show that shader compiler defects are prevalent, and that metamorphic testing provides an effective means for detecting them automatically.

CCS Concepts: • Software and its engineering → Software testing and debugging; • Computing methodologies → Rasterization;

Additional Key Words and Phrases: GPUs, OpenGL, GLSL, testing, shaders, compilers

ACM Reference Format:

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (October 2017), 29 pages.

<https://doi.org/10.1145/3133917>

1 INTRODUCTION

Real-time 2D and 3D graphics, powered by technologies such as OpenGL [Kessenich et al. 2016c], Direct3D [Microsoft 2017a] and Vulkan [Khronos Group 2016], are at the heart of application domains including gaming and virtual reality, and are employed in rendering the graphical interfaces of operating systems, interactive web pages, and safety-related products such as automated driver

This work was supported by EPSRC Early Career Fellowship (EP/N026314/1), an EPSRC-funded studentship via the Centre for Doctoral Training in High Performance Embedded and Distributed Systems (EP/L016796/1), and Imperial College’s EPSRC Impact Acceleration Account.

Authors’ addresses: A. F. Donaldson, A. Lascu, H. Evrard, P. Thomson, Department of Computing, Imperial College London, SW7 2AZ, UK.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2017 Copyright held by the owner/author(s).

2475-1421/2017/10-ART93

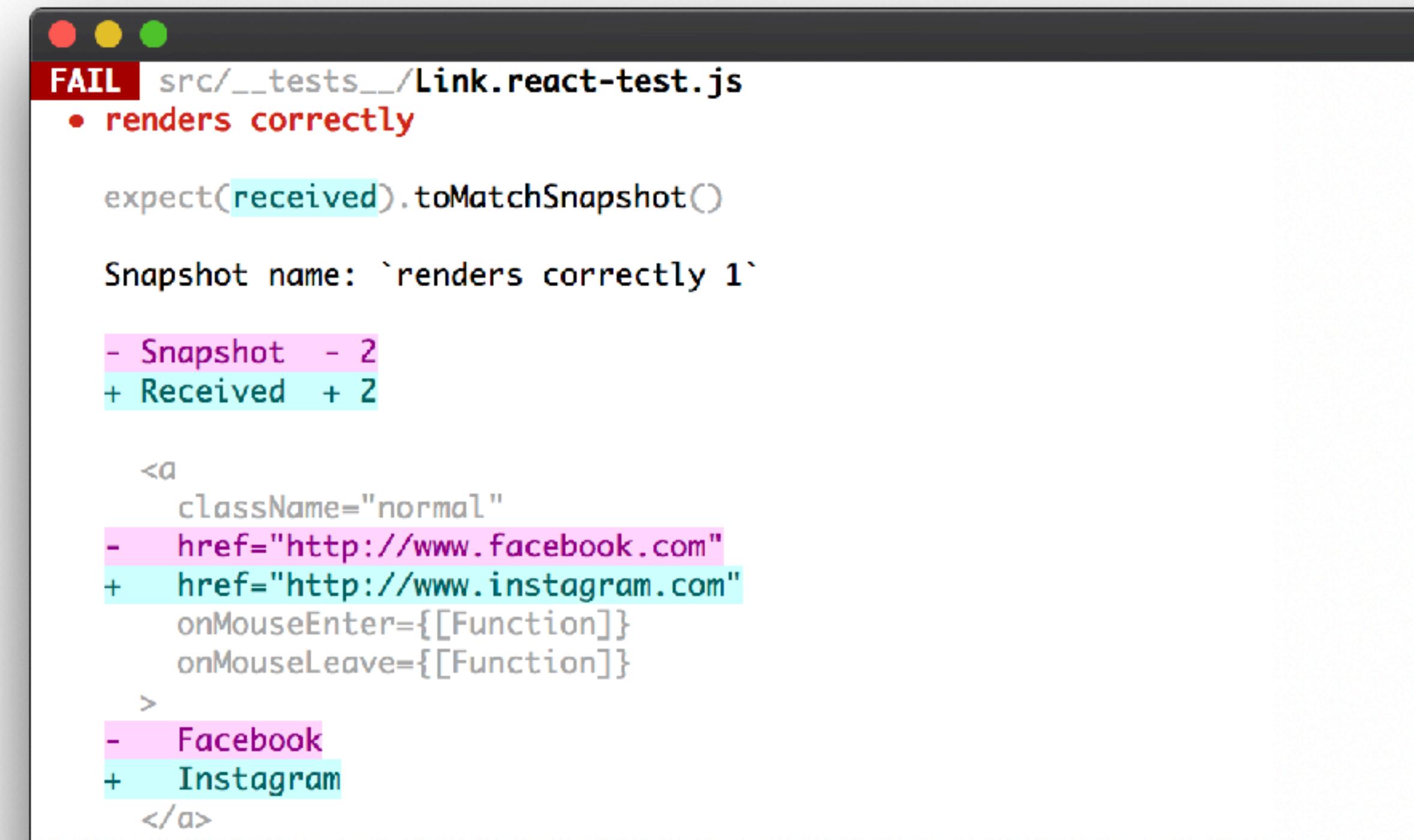
<https://doi.org/10.1145/3133917>

Regression Testing: “Snapshot” Testing

- The first time the test runs, it saves a “snapshot” of the rendered GUI
- Subsequent runs will fail if the snapshot changes

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link page="http://
www.facebook.com">Facebook</Link>)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```



A screenshot of a terminal window showing a failed regression test. The title bar says "FAIL src/_tests_/Link.react-test.js". The message "• renders correctly" is in red. Below it, the command `expect(received).toMatchSnapshot()` is shown. A snapshot comparison follows, with "Snapshot" and "Received" columns. The "Snapshot" column shows a link to "http://www.facebook.com" with the text "Facebook". The "Received" column shows a link to "http://www.instagram.com" with the text "Instagram". The terminal uses color-coded syntax highlighting for code and log output.

```
FAIL src/_tests_/Link.react-test.js
• renders correctly

expect(received).toMatchSnapshot()

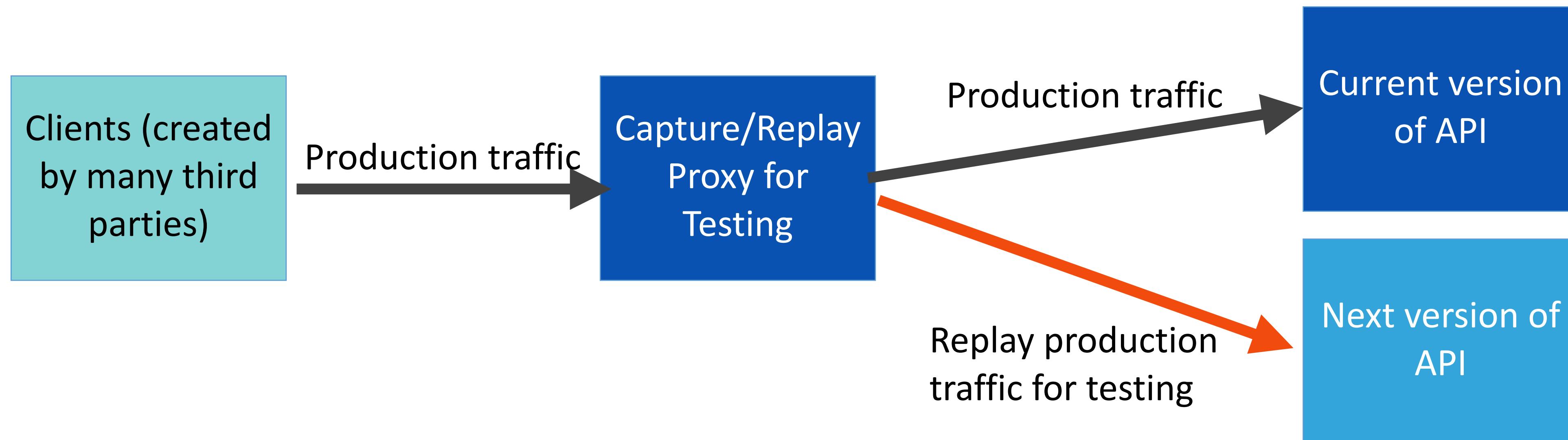
Snapshot name: `renders correctly 1` 

- Snapshot - 2
+ Received + 2

<a
  className="normal"
- href="http://www.facebook.com"
+ href="http://www.instagram.com"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
- Facebook
+ Instagram
</a>
```

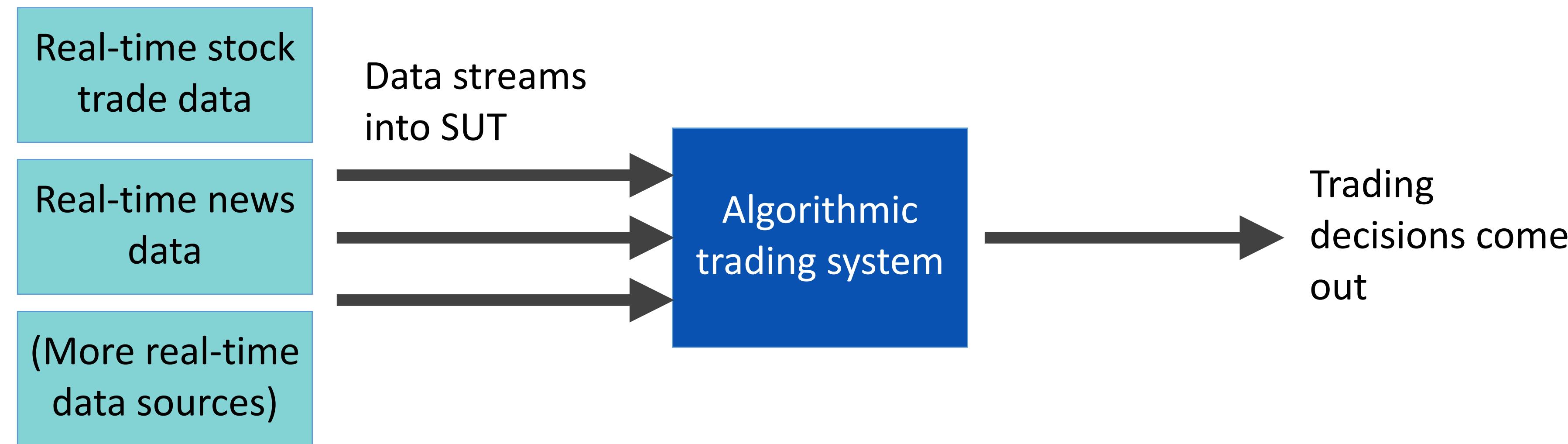
Regression Testing: API Evolution

- Record the API requests and responses that clients make
- Test new versions of the API by identifying requests that result in different responses (“breaking changes”)



Regression Testing: General Record/Replay

- Example: How to test an algorithmic trading system that consumes real time streaming data?



Differential Testing: JVMs

- Problem: How to test JVMs?
 - JVM is very well-specified, but there are nonetheless corner cases & missing tests
- Context: There are many different JVM implementations
- Solution: Compare behavior of each JVM on some class files
- Each discrepancy is a bug in at least one implementation (or the specification!)

Coverage-Directed Differential Testing of JVM Implementations

Yuting Chen

Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
chenyt@cs.sjtu.edu.cn

Ting Su

Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, China
tsuletgo@gmail.com

Chengnian Sun Zhendong Su

Department of Computer Science
University of California, Davis, USA
{cnsun, su}@cs.ucdavis.edu

Jianjun Zhao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
Department of Advanced Information Technology
Kyushu University, Japan
zhao-j@cs.sjtu.edu.cn

Abstract

Java virtual machine (JVM) is a core technology, whose reliability is critical. Testing JVM implementations requires painstaking effort in designing test classfiles (*.class) along with their test oracles. An alternative is to employ binary fuzzing to differentially test JVMs by blindly mutating seedling classfiles and then executing the resulting mutants on different JVM binaries for revealing inconsistent behaviors. However, this blind approach is not cost effective in practice because most of the mutants are invalid and redundant.

This paper tackles this challenge by introducing *classfuzz*, a *coverage-directed fuzzing approach* that focuses on representative classfiles for differential testing of JVMs' startup processes. Our core insight is to (1) mutate seedling classfiles using a set of predefined mutation operators (mutators) and employ *Markov Chain Monte Carlo (MCMC) sampling* to guide mutator selection, and (2) execute the mutants on a reference JVM implementation and use *coverage uniqueness* as a discipline for accepting representative ones. The accepted classfiles are used as inputs to differentially test different JVM implementations and find defects.

We have implemented classfuzz and conducted an extensive evaluation of it against existing fuzz testing algorithms.

Our evaluation results show that classfuzz can enhance the ratio of discrepancy-triggering classfiles from 1.7% to 11.9%. We have also reported 62 JVM discrepancies, along with the test classfiles, to JVM developers. Many of our reported issues have already been confirmed as JVM defects, and some even match recent clarifications and changes to the Java SE 8 edition of the JVM specification.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—Testing tools (e.g., data generators, coverage testing); D.3.3 [Programming Languages]: Language Constructs and Features—Classes and objects; D.3.4 [Programming Languages]: Processors—Code generation

General Terms Algorithms, Reliability, Languages

Keywords Differential testing, fuzz testing, Java virtual machine, MCMC sampling

1. Introduction

Java Virtual Machine (JVM) is a mature Java technology. A JVM is responsible for loading, linking, and executing Java classfiles (*.class) in the same way on any platform [29]. Various JVMs (i.e., JVM implementations), such as Oracle's HotSpot [3], IBM's J9 [4], Intel's RVM [5], Azul's Zulu [6], and GNU's GJ [2], are available and still evolving. They adopt different implementation techniques (such as just-in-time compilation, ahead-of-time compilation, and interpretation) and adapt to different operating systems and CPU architectures. To ensure their compatibility, they must consistently implement a single JVM specification [25].

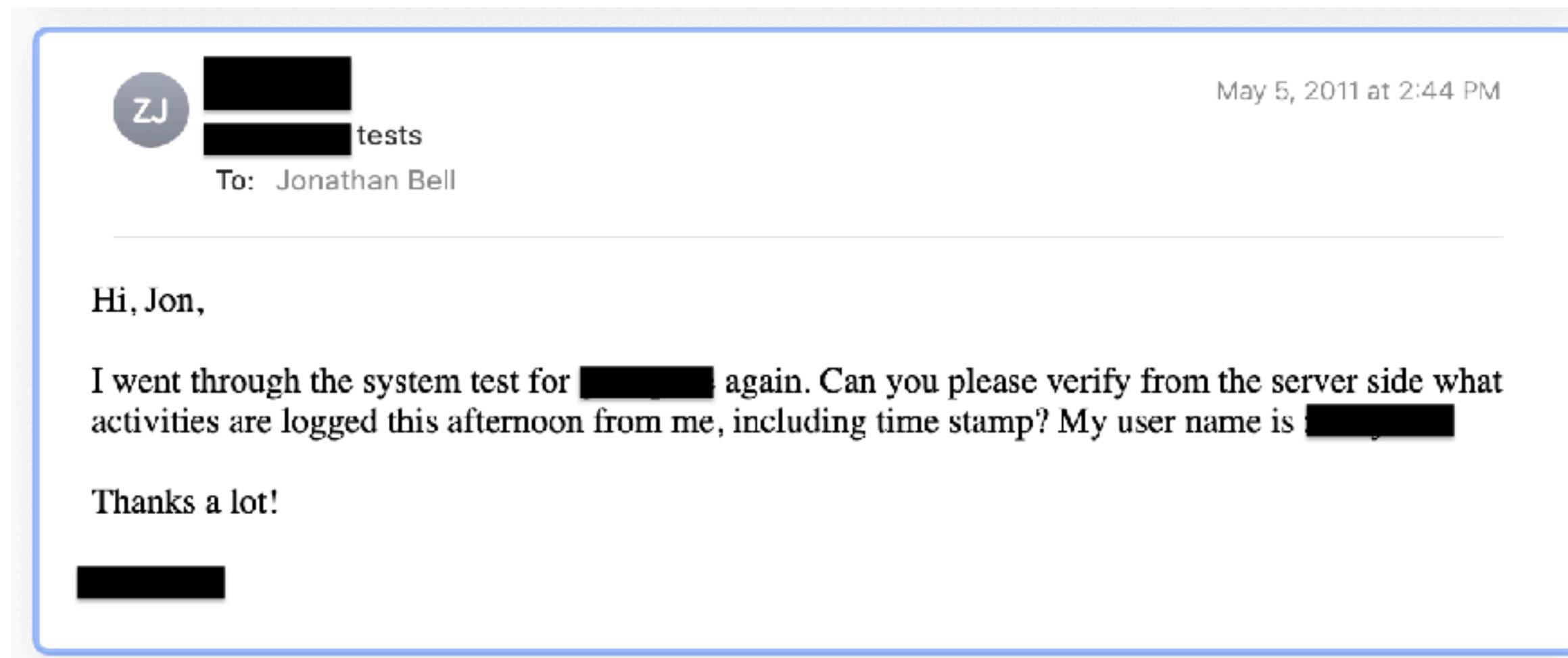
The reality is that no two JVMs are exactly alike, and they can behave disperately when encountering corner cases or invalid classfiles: a Java class can run on some JVMs but not

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...\$15.00
<http://dx.doi.org/10.1145/2908000.2918095>

Generalized Oracles: Human Evaluation

- Some oracles might be too difficult to encode in an automated test
- This is an anti-pattern



Generalized Oracles: Human Evaluation

Usability Testing

- Observe real users interacting with your software - provide each user with a task, monitor their progress towards completing that task
- Consider a diverse set of users that represent those who will use your software
- Validate problems (and fixes) that you identify in cognitive walkthroughs
- Example: usability testing for [Microsoft Academic](#)

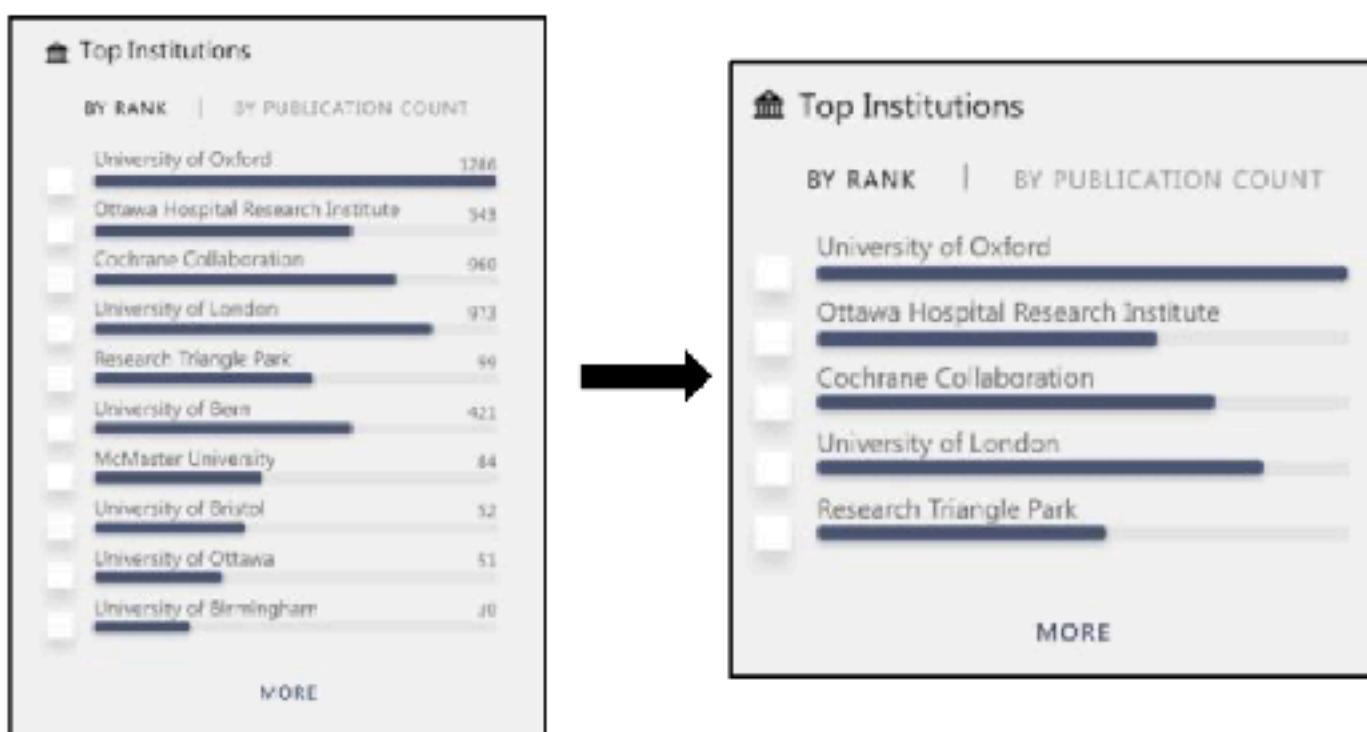


Figure 5. Issues 1 & 2 filtering redesign. (Left) Original: List of institutions with publication counts for each. (Right) post-GenderMag: Shorter list of institutions and removed the publication counts that drew attention away from the checkbox actionability.

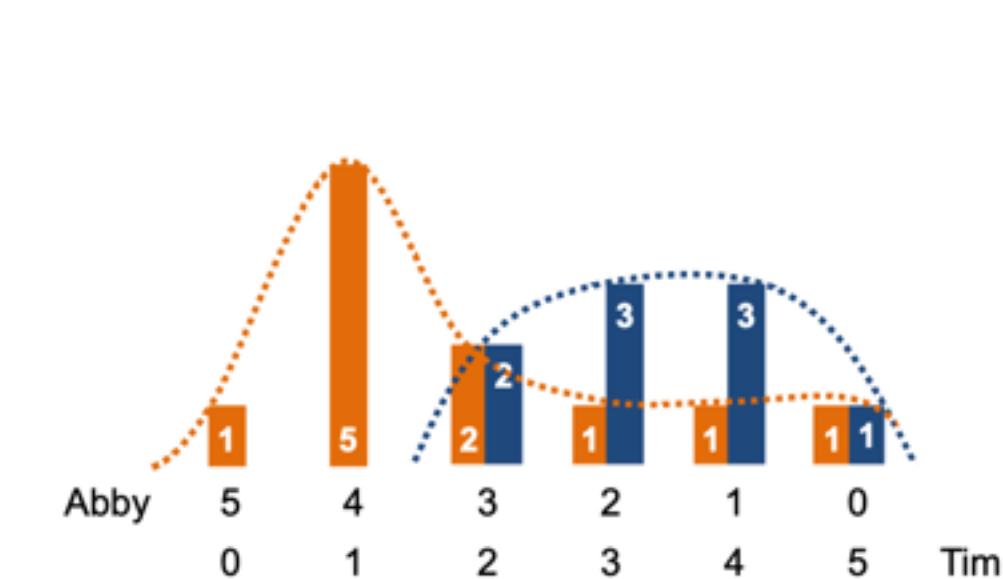


Figure 13. Y-axis: Counts of the 20 men and women participants by their facet values. (Same as Figure 2 but broken out by gender.) Orange: women, blue: men. X-axis: Abby=Abby Facets, Tim=Tim Facets. Example: the left bar says that the only participant with 5 Abby facets (0 Tim facets) was a woman; the right pair of bars says that one man and one woman had 5 Tim facets (0 Abby facets).

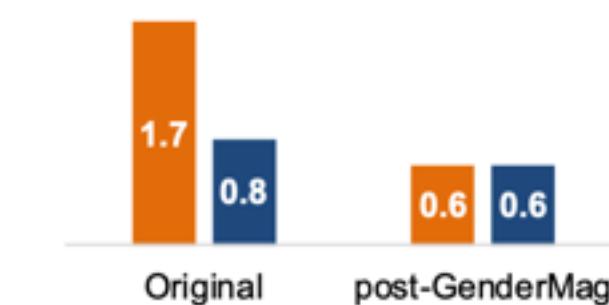
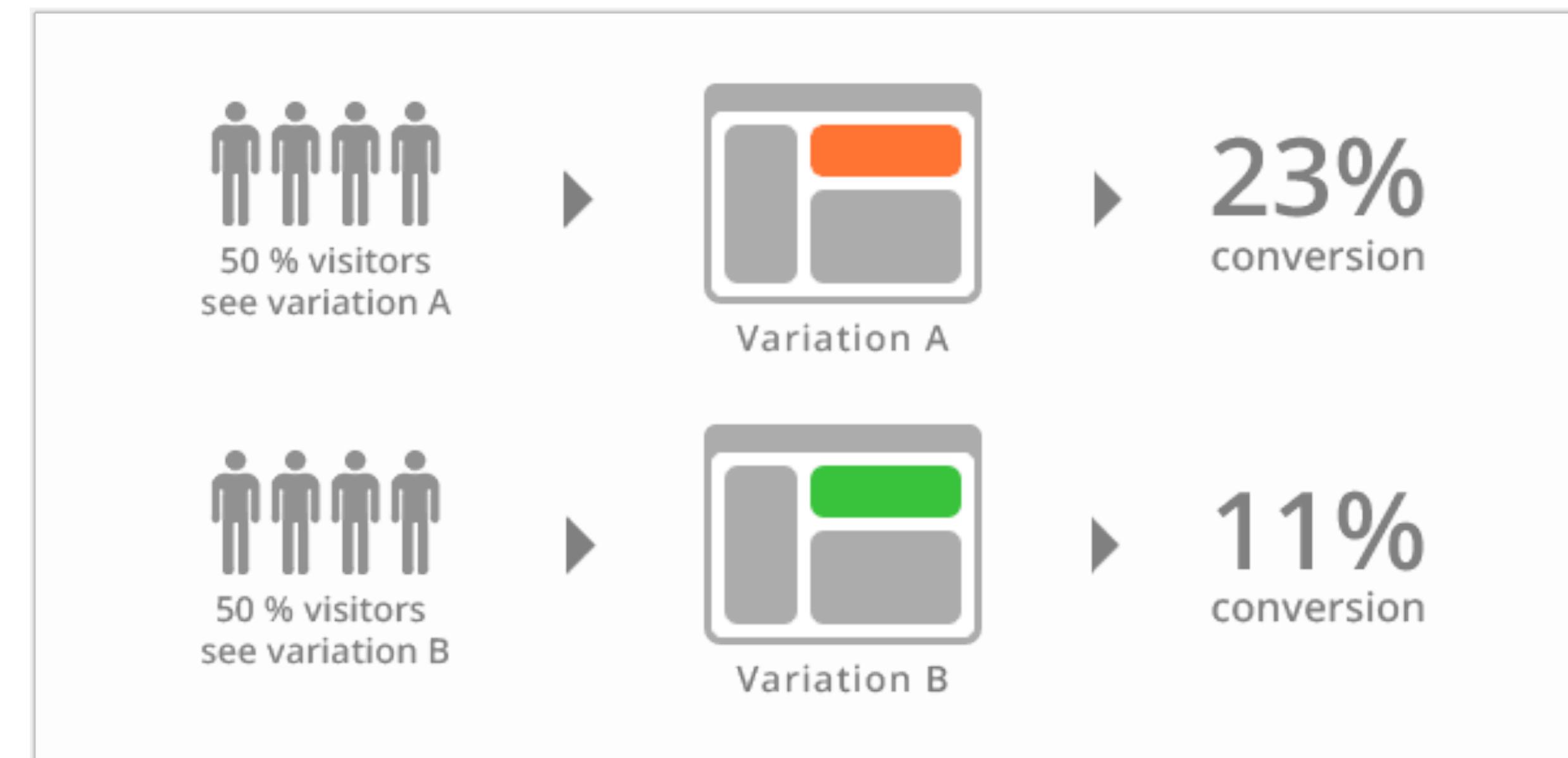


Figure 14. Average number of action failures per person by gender identification (orange: women, blue: men). In the Original version, women's action failure rates were over twice as high as men's; with the post-GenderMag redesign, all failure rates went down, and the gender gap disappeared.

Generalized Oracles: Human Evaluation

A/B Testing

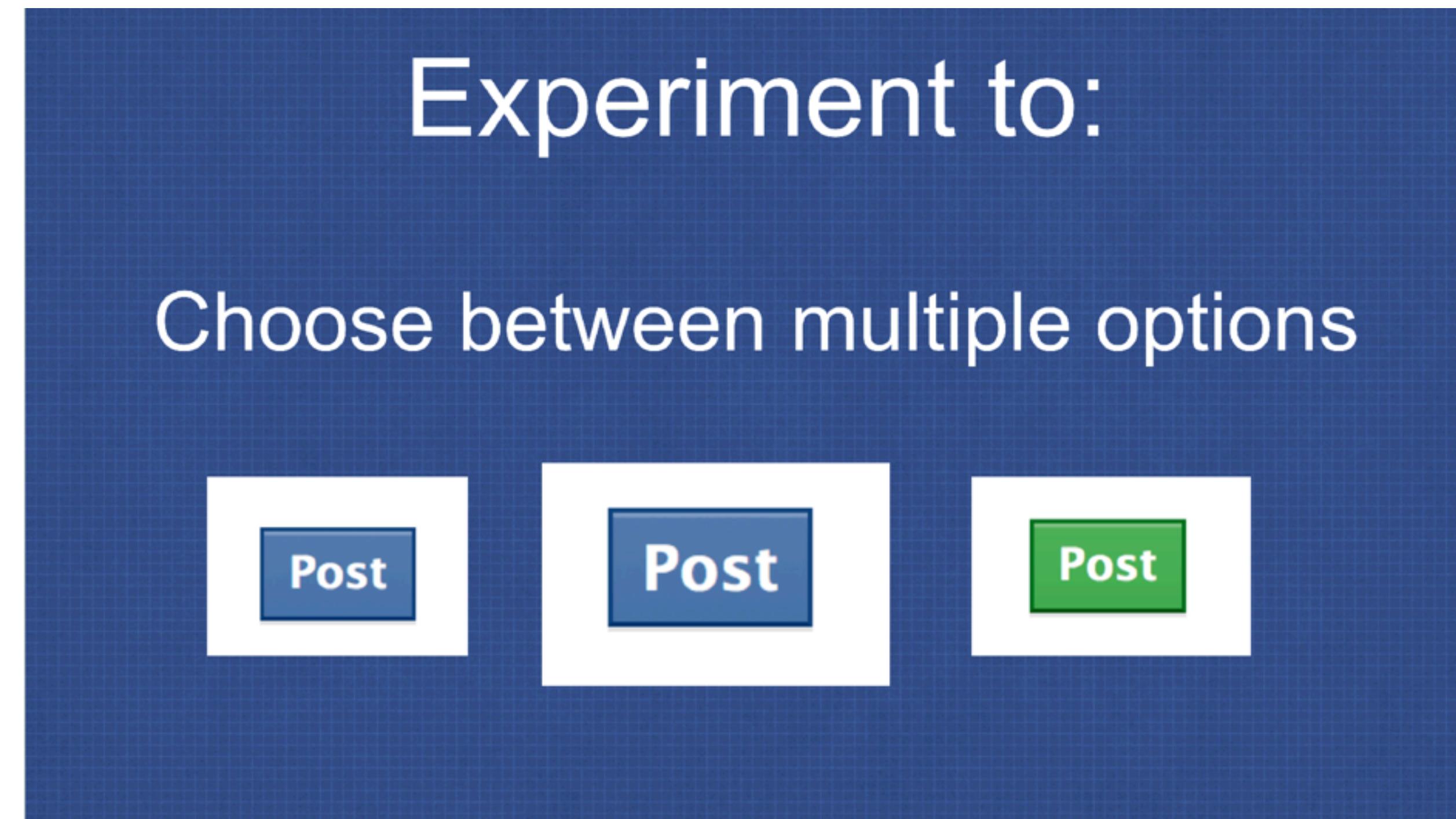
- Ways to test new features for usability, popularity, performance without a focus group
- Show 50% of your site visitors version A, 50% version B, collect metrics on each, decide which is better



Generalized Oracles: Human Evaluation

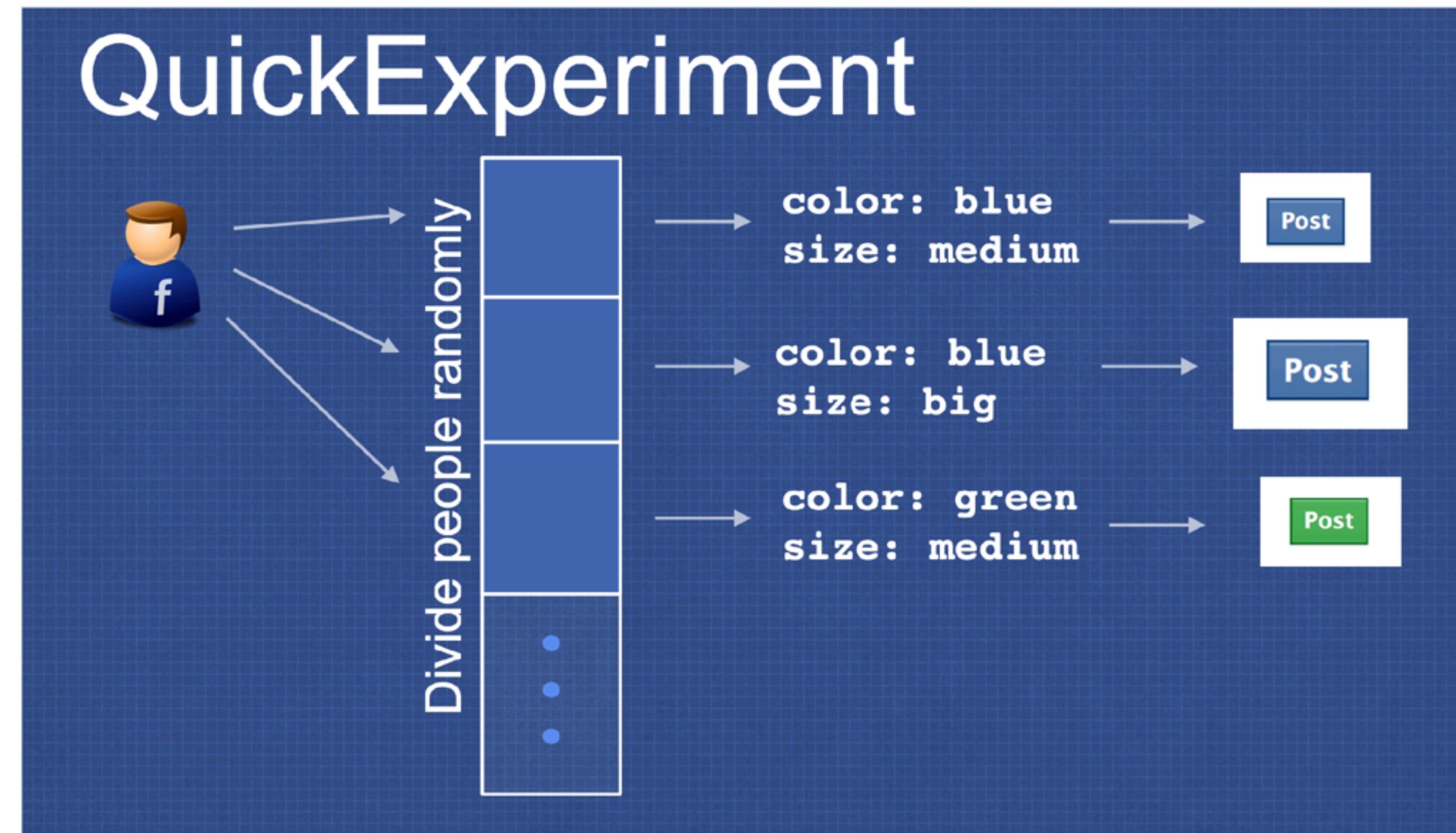
A/B Testing: PlanOut from Facebook (“N=10⁹ user study”)

- Used to test advertising strategies (and Facebook functionality)
- Segment audience and define KPIs, collect results



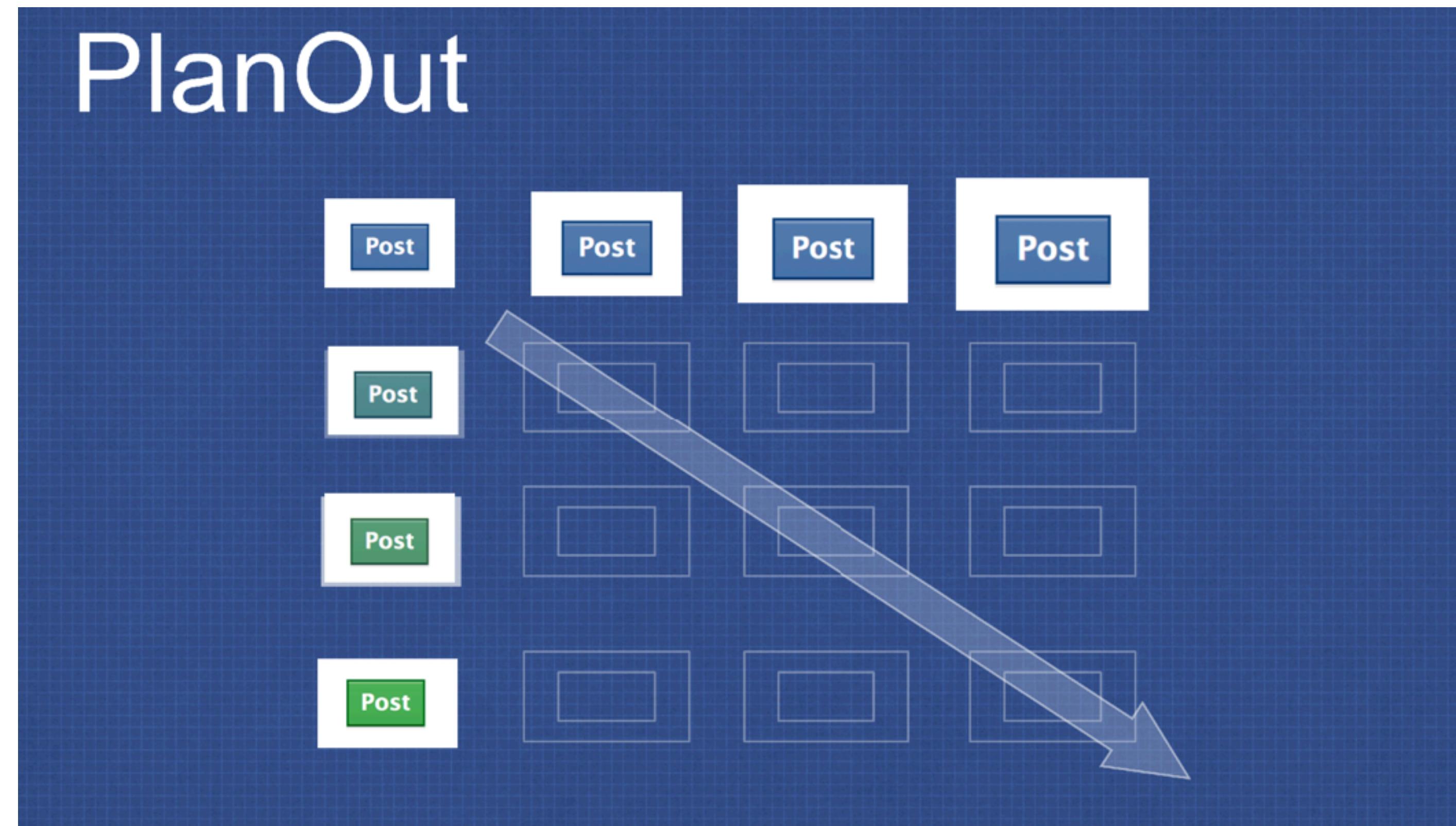
Generalized Oracles: Human Evaluation

A/B Testing: PlanOut from Facebook (“N=10⁹ user study”)



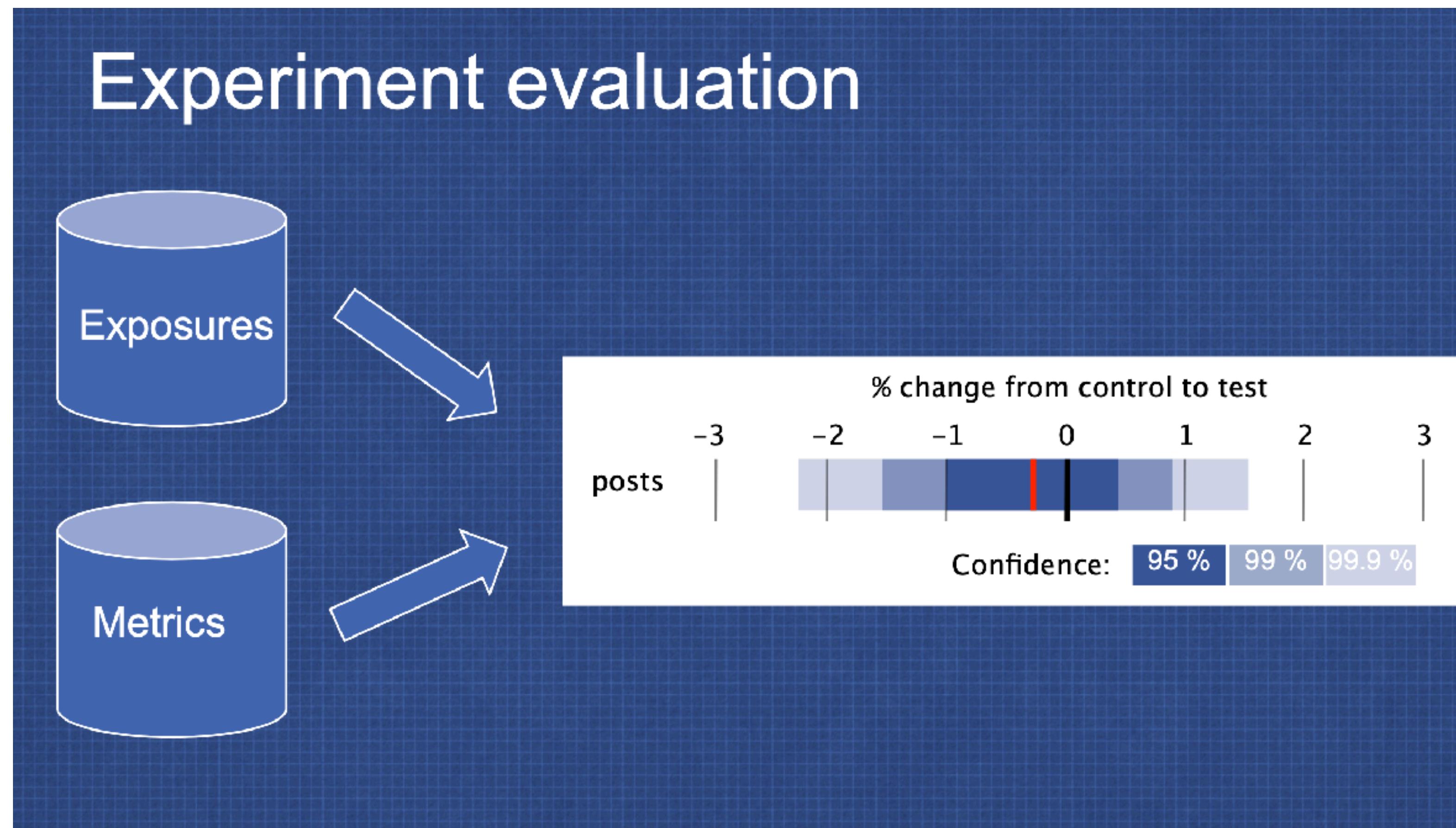
Generalized Oracles: Human Evaluation

A/B Testing: PlanOut from Facebook (“N=10⁹ user study”)



Generalized Oracles: Human Evaluation

A/B Testing: PlanOut from Facebook (“N=10⁹ user study”)



What is a “good” test suite?

Interpretation: Coverage of Requirements

- Enumerate all inputs
- Enumerate all behaviors
- Check system wrt all inputs and behaviors
- (Manually) create traceability links between requirements and their tests



Coverage of Requirements at Google

“The Beyoncé Rule”



What is a “good” test suite?

Interpretation: Coverage of statements or branches

- Intuition: if a test doesn’t execute a statement or branch, it sure isn’t checking its behavior
- Automatically determine which statements or branches are covered by a test

```
function magic(x: number, y: number) {  
  let z = 0;  
  if (x !== 0) { ✓ T1  
    z = x + 10;  
  } else { ✓ T2  
    z = 0;  
  }  
  if (y > 0) { ✓ T1  
    return y / z;  
  } else { ✓ T2  
    return x;  
  }  
}  
test("100% branch coverage", () => {  
  expect(magic(1, 22)).toBe(2); //T1  
  expect(magic(0, -10)).toBe(0); //T2  
});
```

What is a “good” test suite?

Interpretation: Other code-coverage metrics

- MCDC
 - Like branch coverage, but requires every condition in each if statement is taken and shown to affect decision outcomes independently
 - Required for safety-critical avionics, automotive & space software
- Path coverage
 - Has every possible route through the program been taken?
 - General limitations?

Does Higher Branch Coverage Imply More Faults Found?

- “there is a moderate to very high correlation between the effectiveness of a test suite and the number of test methods it contains”
- “there is a moderate to high correlation between the effectiveness and the coverage of a test suite when the influence of suite size is ignored”
- “the correlation between coverage and effectiveness drops when suite size is controlled for. After this drop, the correlation typically ranges from low to moderate, meaning it is not generally safe to assume that effectiveness is correlated with coverage”

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{linozem,rholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are rare in practice, so the results may not generalize to typical test suites.

We have extended these studies by evaluating the relationship between test suite size, coverage, and effectiveness for large Java programs. Our study is the largest to date in the literature: we generated 31,000 test suites for five systems consisting of up to 724,000 lines of source code. We measured the statement coverage, decision coverage, and modified condition coverage of these suites and used mutation testing to evaluate their fault detection effectiveness.

We found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Our results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging;
D.2.8 [Software Engineering]: Metrics—product metrics

General Terms

Measurement

Keywords

Coverage, test suite effectiveness, test suite quality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](http://permissions.acm.org).
Copyright is held by the author(s). Publication rights licensed to ACM.
ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/05
<http://dx.doi.org/10.1145/2568225.2568271>

1. INTRODUCTION

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

Unfortunately, these studies do not agree on the strength of the relationship between these test suite characteristics. In addition, three issues with the studies make it difficult to generalize their results. First, some of the studies did not control for the size of the suite. Since coverage is increased by adding code to existing test cases or by adding new test cases to the suite, the coverage of a test suite is correlated with its size. It is therefore not clear that coverage is related to effectiveness independently of the number of test cases in the suite. Second, all but one of the studies used small or synthetic programs, making it unclear that their results hold for the large programs typical of industry. Third, many of the studies only compared adequate suites; that is, suites that fully satisfied a particular coverage criterion. Since adequate test suites are rare in practice, the results of these studies may not generalize to more realistic test suites.

This paper presents a new study of the relationship between test suite size, coverage and effectiveness. We answer the following research questions for large Java programs:

RESEARCH QUESTION 1. *Is the effectiveness of a test suite correlated with the number of test cases in the suite?*

RESEARCH QUESTION 2. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage and/or modified condition coverage when the number of test cases in the suite is held constant?*

RESEARCH QUESTION 3. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage and/or modified condition coverage when the number of test cases in the suite is held constant?*

The paper makes the following contributions:

- A comprehensive survey of previous studies that investigated the relationship between coverage and effectiveness (Section 2 and accompanying online material).

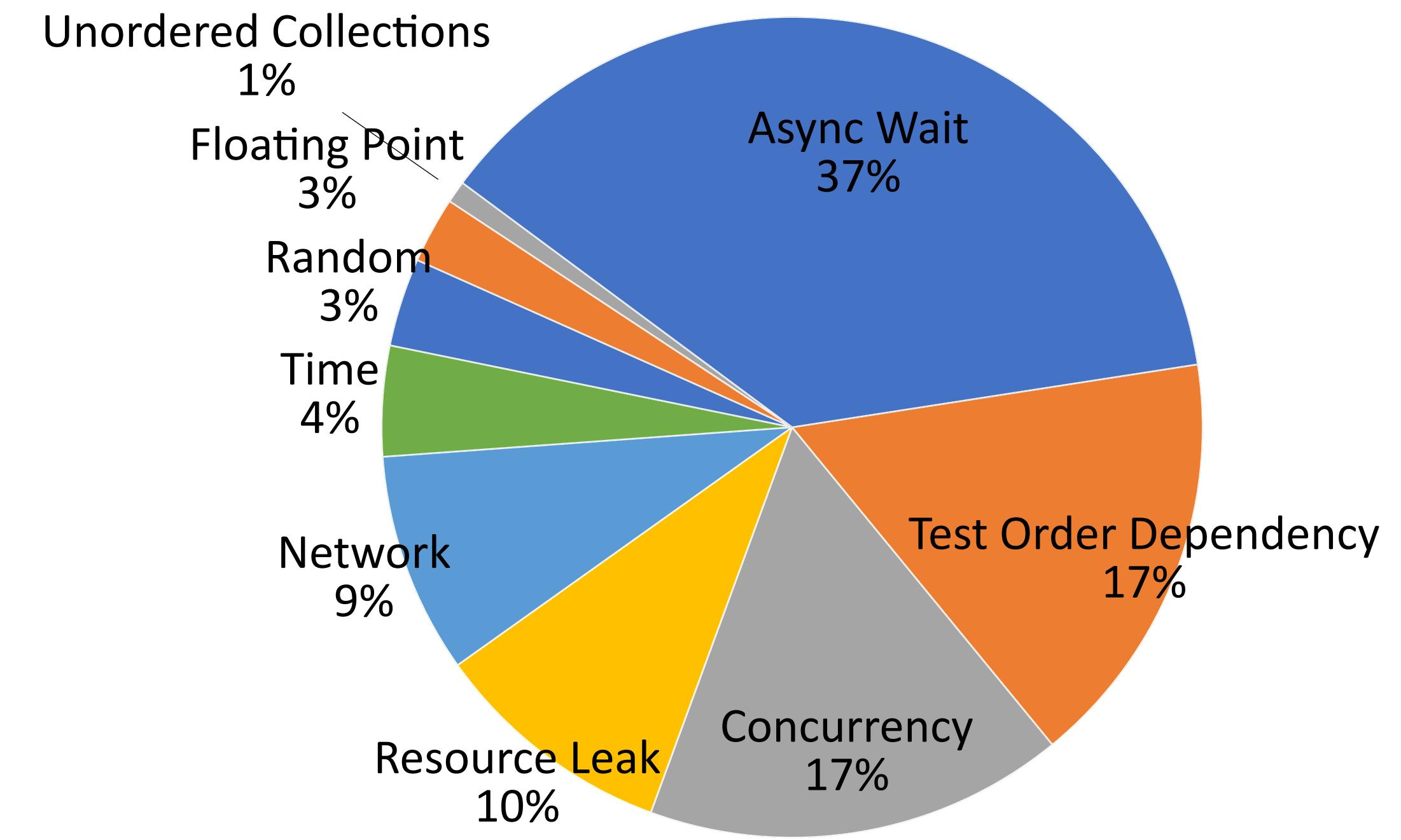
What makes a good test?

More than just coverage and oracles

- Tests should be hermetic: include all information and dependencies necessary to run the test
- Tests should be clear: improves debugging later on
- Tests should be scoped as small as possible: faster and more reliable
- Tests should make calls against public APIs

Good Tests are Not Flaky

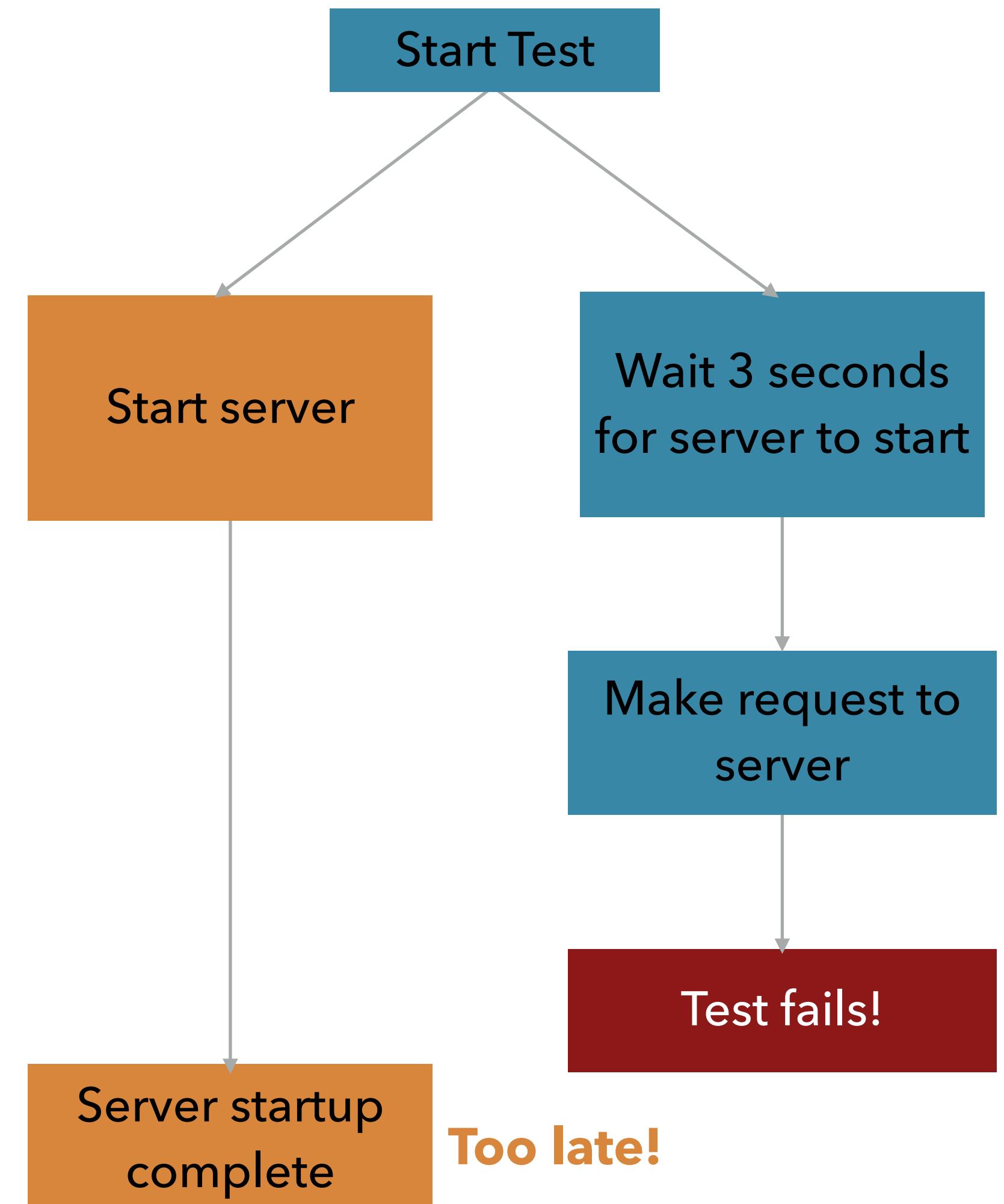
- Flaky test failures are false alarms
- Tests that are hermetic defend against “test order dependency” - failures due to tests running in other orders
- Most common cause of flaky test failures: “async wait” - tests that expect some asynchronous action to occur within a timeout
- Good tests avoid relying on timing



[Luo et al, FSE 2014 “An empirical analysis of flaky tests”]

Flaky Test Example: Async/Wait

- Most common root cause of flakiness
- Difficult to avoid, but consider:
 - Have more “small” tests that don’t require concurrency
 - Ensure sufficient resources available for running tests
 - Embed reasonable error detection to classify test failures as likely to be “flaky” vs true failures



What is a “good” test suite?

Interpretation: It has strong oracles

- Strong oracles should fail the test if the behavior is incorrect
- How to evaluate the strength of oracles?
- Strawman - “Seeded Faults”:
 - Create N variations of the codebase, each with a single manually-written defect
 - Evaluate the number of defects detected by test suite
 - Test suite is “good” if it finds all of the bugs you can think of

Mutation Analysis Tests the Tests

Idea: What if many (real) bugs could be represented by a single, one-line “mutation” to the program?

```
public contains(location: PlayerLocation): boolean {
    return (
        location.x + PLAYER_SPRITE_WIDTH / 2 > this._x &&
        location.x - PLAYER_SPRITE_WIDTH / 2 < this._x + this._width &&
        location.y + PLAYER_SPRITE_HEIGHT / 2 > this._y &&
        location.y - PLAYER_SPRITE_HEIGHT / 2 < this._y + this._height
    );
}
```

Correct code for “Contains” check in [Covey.Town](#)

```
public contains(location: PlayerLocation): boolean {
    return (
        location.x + PLAYER_SPRITE_WIDTH / 2 < this._x &&
        location.x - PLAYER_SPRITE_WIDTH / 2 < this._x + this._width &&
        location.y + PLAYER_SPRITE_HEIGHT / 2 > this._y &&
        location.y - PLAYER_SPRITE_HEIGHT / 2 < this._y + this._height
    );
}
```

Mutated (and buggy) code for “Contains” check in [Covey.Town](#)

Mutation Analysis Tests the Tests

- Automatically mutates SUT to create mutants, each a single change to the code
- Runs each test on each mutant, until finding that a mutant is detected by a test
- Can be a time-consuming process to run, but fully automated

Mutation Report Shows Undetected Mutants

- Mutants “detected” are bugs that are found
- Mutants “undetected” might be bugs, or could be equivalent to original program (requires a human to tell)

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	90.30%	90.30	121	13	0	0	0	0	121	13	134
TS ConversationArea.ts	76.92%	76.92	10	3	0	0	0	0	10	3	13
TS InteractableArea.ts	97.01%	97.01	65	2	0	0	0	0	65	2	67
TS Town.ts	85.00%	85.00	34	6	0	0	0	0	34	6	40
TS ViewingArea.ts	85.71%	85.71	12	2	0	0	0	0	12	2	14

```
public overlaps(otherInteractable: InteractableArea): boolean {  
    const toRectPoints = ({ _x, _y, _width, _height }: InteractableArea) => ({ x1: _x - PLAYER_SPRITE_WIDTH / 2, y1: _y - PLAYER_SPRITE_HEIGHT / 2, x2: _x + _width / 2, y2: _y + _height / 2 } as Rect);  
    const rect1 = toRectPoints(this);  
    const rect2 = toRectPoints(otherInteractable);  
    const noOverlap = rect1.x1 >= rect2.x2 || rect1.y1 >= rect2.y2 || rect1.x2 <= rect2.x1 || rect1.y2 <= rect2.y1;  
    return !noOverlap;  
}
```

Use Mutation Analysis While Writing Tests

- When you feel “done” writing tests, run a mutation analysis
- Inspect undetected mutants, and try to strengthen tests to detect those mutants

```
< > ⚡ Killed (65) ✓ Survived (2)
132
133     public overlaps(otherInteractable: InteractableArea): boolean {
134         const toRectPoints = ({ _x, _y, _width, _height }: InteractableArea) => ({ x1: _x
135         const rect1 = toRectPoints(this);
136         const rect2 = toRectPoints(otherInteractable);
137 -         const noOverlap = rect1.x1 >= rect2.x2●
138 +         const noOverlap = rect1.x1 > rect2.x2
139             || rect2.x1 >= rect1.x2 || rect1.y1 >= rect2.y2 || rect2.y1 >= rect1.y2; ●
140     }
141 }
```

Detailed mutation report for “overlaps” method - two mutants were not detected!

Undetected Mutants May Not Be Bugs

- Unfortunately, we can not automatically tell if an undetected mutant is a bug or not

```
265  public initializeFromMap(map: ITiledMap) {  
266    const objectLayer = map.layers.find(eachLayer => eachLayer.name  
267 -    if (!objectLayer) {●●  
268 -      throw new Error(`Unable to find objects layer in map`);●  
269 -    }  
270 +    if (!objectLayer) {}  
271      const viewingAreas = objectLayer.objects  
272        .filter(eachObject => eachObject.type === 'ViewingArea')  
273        .map(eachViewingAreaObject => ViewingArea.fromMapObject(eac
```

This mutant is *equivalent* to the original program: Even without this check for undefined, an error is still thrown when the undefined layer is dereferenced on the following line

```
62  public static fromMapObject(mapObject: ITiledMapObject, broadcas  
63    const { name, width, height } = mapObject;  
64    if (!width || !height) {●  
65 -      throw new Error(`Malformed viewing area ${name}`);●  
66 +      throw new Error(``);  
67    }  
68    const rect: BoundingBox = { x: mapObject.x, y: mapObject.y, w  
69  }
```

This mutant is *equivalent* to the original program: Even though the error message changed, the specification doesn't indicate what error message should be thrown.

Roadmap

- Thursday:
 - Are mutants a valid substitute for real faults?
 - Can automation help developers write assertions?
- Next week:
 - How to generate inputs?
 - Project proposal brainstorming
- 2 weeks:
 - Very large case study: generating inputs, property testing
 - Project proposal discussions
- 3 weeks: Continuous integration and test suite maintenance

What makes a bad test: Flaky Tests

Why do Google's testing infrastructure team hate “Large” tests?

- How do we (reliably, repeatedly, cheaply) execute a test that:
 - Changes some global variables?
 - Changes the state of a database?
 - Executes stock trades?
 - Connects to remote servers?

Flaky Tests

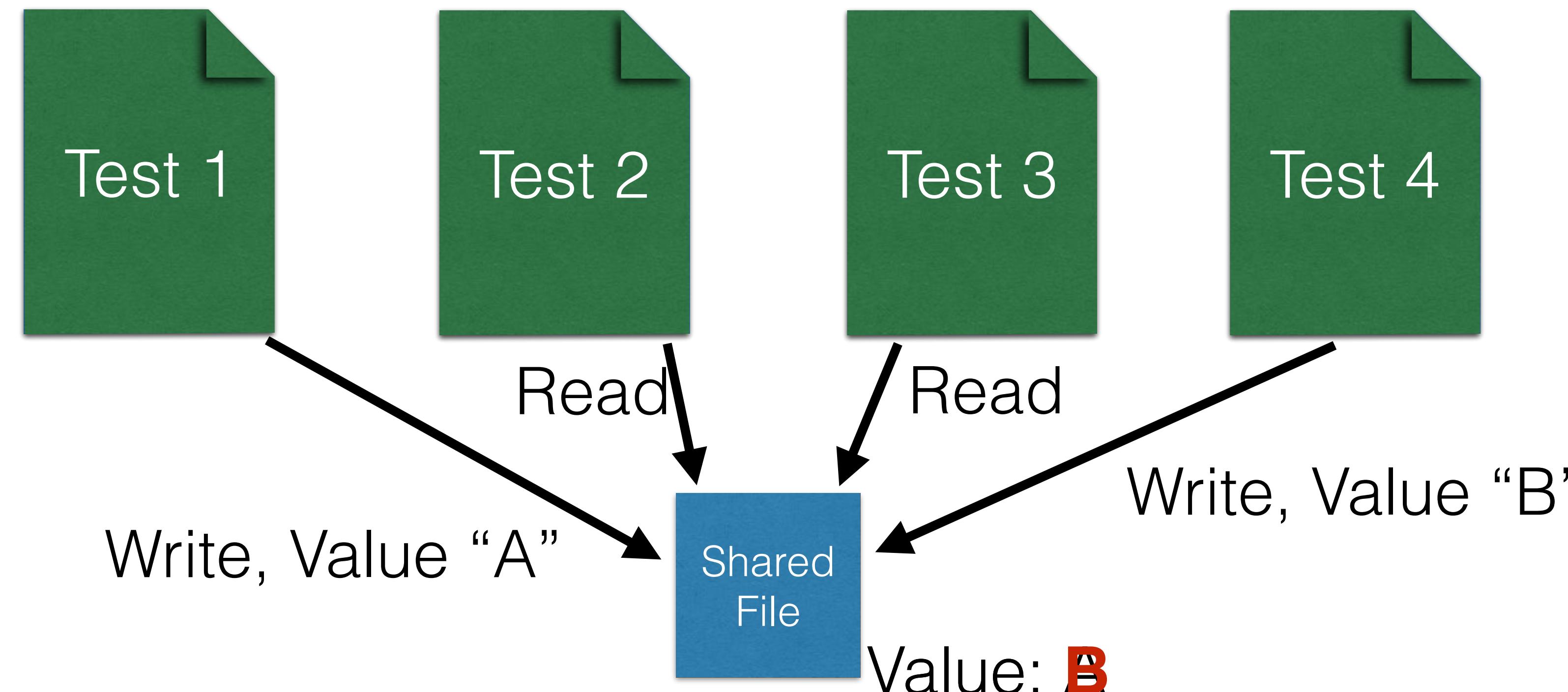
An anti-pattern in testing

- Google: 16% of all automated tests are flaky
- Microsoft: 5% of Windows & Dynamics CRM tests are flaky
- Facebook: “Assume all tests are flaky”
- Most developers: flaky tests are a nuisance!



Flaky Tests

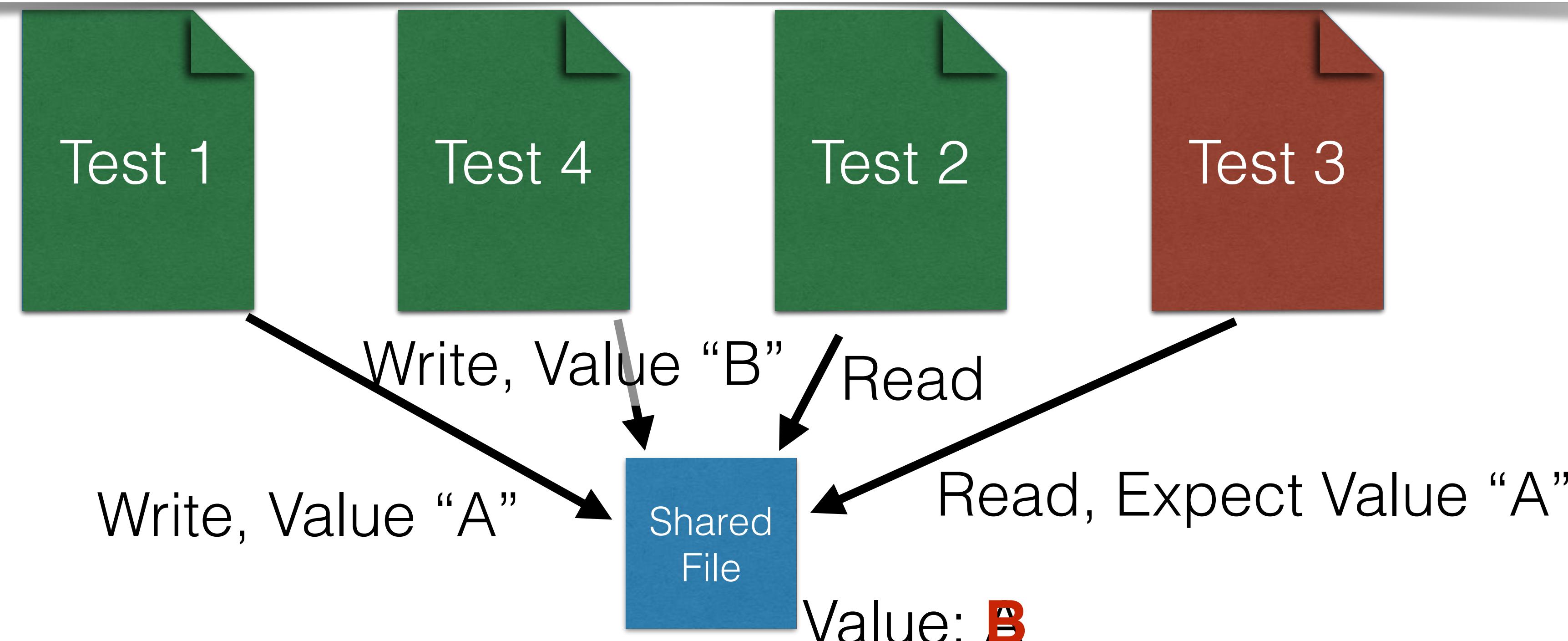
Test Order Dependencies



Flaky Tests

Test Order Dependencies

A flaky test: outcome of Test 3 changed, but the code hasn't changed!



Flaky Tests & Test Order Dependencies

Touch global variables or database?

Option 1

```
let myVar = 5;
describe('test with dependency', function() {
  before(() => {
    // runs once before the first test in this block
    myVar = 10;
  });

  it("is a terrible test", ()=>{
    //do lots of stuff
    myVar = 5;
    //do lots of stuff
    expect(myVar).to.be(5);
  });
  after(() => {
    // runs once after the last test in this block
    myVar = 10;
  });
});
```

Setup, teardown methods

Fast, but “compliance appliance”

Option 2

Test 1

is a terrible test

Test 2

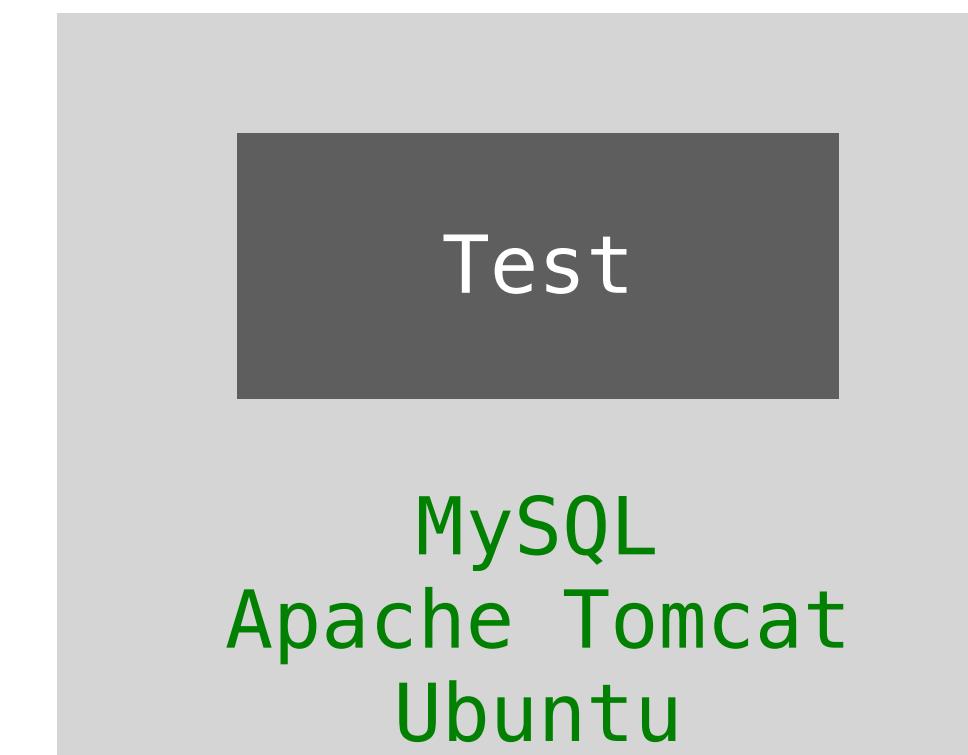
Isolate each test in a new process
(or container)

Slow, but “non-compliance appliance”

Flaky Tests & Test Order Dependencies

System tests at scale

- Relying on engineers to develop and maintain reliable setup/teardown results in unreliable tests
- Without isolation, can't run multiple tests concurrently
- Common solution: system tests run in entirely isolated environments



Test (running in a newly provisioned VM)

Flaky Tests Overall

A problem we're stuck with?

- Reduce the scope of a test: small tests aren't flaky
- Remove timed waits, increase timeouts: reduce flaky failures?
- Make tests more understandable: can you tell if a failure is flaky or not?
- Mitigate with reruns, but this increases test cost

